Program Structures for Non-proper Programs

R. C. Tausworthe
DSN Data Systems Development Section

Canonic structured programming forms the basis of an attractive software design and production methodology applicable to proper programs (programs having but one entry point and one exit point). Programs developed using this methodology tend to be easier to organize, understand, modify, and manage than are unstructured programs. However, there are notable examples in which programs either are inherently non-proper (usually, with more than one exit, rather than more than one entry), or else suffer when forced to be structured. This article addresses ways of extending the concept of structured programming to cover such cases; it is a report of an ongoing research activity to examine potential Deep Space Network software development standards.

I. Introduction

Canonic program control-logic structures, such as sequence, DOWHILE, and IFTHENELSE proposed by Böhm and Jacopini (Ref. 1) and others (Refs. 2, 3) form the basis of an attractive software design and production methodology, known as "structured programming," applicable to *proper* programs – those that have one point of entry and one exit point. Such programs developed using top-down modular, hierachic structured programming techniques tend to be easier to organize, understand, modify, and manage, especially when the canonic set includes other simple extensions of the three above, such as WHILEDO and CASE (Fig. 1).

However, there are typical cases where the strict adherence to the "one-entry-one-exit" rule for a program or program module is a *hindrance*, rather than a *help*, to effective software development. Structure for the sake of structure should not overrule structure for the sake of clarity.

One notable example of such counter-productivity, occurs when one is designing a program that is capable of detecting the existence of situations for which further processing in the current mode is either useless or unnecessary. Often, in such cases, the most desired, most logical, and most clearly understood course is to divert program control to a recovery mode or back to the user/operator for subsequent decision making and manual operations (Fig. 2).

The alternative to programming such abnormal exits of a module is to introduce structure flags as necessary to force these exits to the normal exit point; but then this flag has to be tested each time a "normal" action in the program comes up for execution. If an abnormal condition has occurred, the normal action must be bypassed (Fig. 3). Bypassing is necessary until an appropriate nesting level is reached that the appropriate recovery procedure can be invoked in a properly structured way. This not only introduces a clutter of excess, distracting detail to slow down the programmer, but it also creates a somewhat larger, slower program. Hence, besides interfering with programmer effectiveness, strict adherence to canonic proper-program structures has caused the program itself to suffer.

It is also the case in many of the higher level languages that some statements can cause unavoidable, automatic branching to prespecified or default program locations when certain conditions occur. For example, in FORTRAN and PL/1, executing the file-input statement can result in normal input (the program continues at the next statement), an end-of-file condition (the program branches to a prespecified statement), or a file-error condition (the program branches to a separately specified statement). True "structured programming" (using canonic structures) is thus not possible whenever such statements appear.

II. Criteria for Structuring Multi-Exit Modules

The context of structured programming obviously needs to be extended, in such cases, to include constructs that fit the language and that will tend to increase design productivity and program efficiency. But great care must be taken in extending the basic set of structures, not to undo (or potentially undo) the progress that canonic structures have contributed to software development. Mills' (Ref. 3) proof of the correctness theorem depends on the "one-entry-one-exit" character of programs. Permitting modules to have multiple exits (or entries) can, therefore, be a very dangerous policy unless that policy is limited to justifiable situations where correctness is not impaired. I shall judge candidate structures to augment the canonic set relative to the following criteria:

(1) The top-down development and readability of the program design must not be impaired by the extended structures.

(2) The hierarchic, modular form of the program must be maintained using the extended structures.

(3) Program clarity and assessment of correctness on an individual module basis must not be jeopardized.

(4) The situations under which an alternate exit of a module is permissible must be limited to special situations where the need is clear and desirable, or unavoidable.

(5) The new structures must conform to the same codability conventions used for the canonic set, such as modular indentation of lines of code, easily identifiable entry and exit points, and clear connectivity of program modules.

III. Structures for Multi-Exit Modules

Iterations and nestings of canonically structured proper program modules always result in proper programs. Whenever a branching (one entry, multiple exit) node appears in a structure, there also appears a collecting node and one or more process nodes within the structure so arranged that the global view again has only one entry and one exit.

The extension of this philosophy to modules having multiple exits suggests the simple extension to structured programs found in Fig. 4.

The entire structure is a proper module, although module *A* obviously is not. However, if the function *A* has been stated explicitly enough that the two exit conditions are determinable, based on entry conditions to *A*, then proof of correctness is conceptually the same as for an IFTHENELSE structure. I shall use the convention found in Fig. 5 to denote and emphasize the condition for that other exit. The condition or event *c* under which the exit occurs is directly displayed for more clarity and better understanding.

When there are more than two exits, these can be accommodated by another configuration, analogous to the CASE structure in Fig. 6.

The box *A* in Fig. 6 represents, for example, the way end-of-file and file-error conditions are actually treated in programming languages such as FORTRAN and PL/1. Using the configuration shown permits file input modules in such languages to take a structured appearance not otherwise achievable.

Normally, I draw the collecting node of CASE and IFTHENELSE constructs directly under the bottom vertex of the decision symbol. However, the exits in Figs. 5 and 6 are unusual exits from a module, so I do not. Normal flow is straight down.

Looping structures can similarly be extended by this technique, as shown in the four configurations in Fig. 7. Structures (a) and (d) in Fig. 7 represent examples of programs which endlessly process streams of input data until the data quality falls below a specified condition *c*, at which time, some alternate procedure is invoked. Structure (b) represents a program *A* in which *c* senses an abnormal condition; *B* is a recovery module which initializes *A* for another try. The structure (c) would find application, for example, when information is being inserted at a terminal by *A* for processing by *B*. If *c* detects an error, the program returns to *A* for correct input; otherwise, it continues.

IV. Hierarchic Expansion of Multi-Exit Modules

All of these configurations certainly satisfy the first four criteria for extensions to the basic proper structures, at least when viewed macroscopically, as Figs. 5, 6, and 7 are. But what happens when a multi-exit function (box) at one level expands (to a flowchart) at the next hierarchic design level?

The ANSI standard (Ref. 4) technique for denoting hierarchic flowchart expansion is by way of *striping* the box to be expanded, as shown in Fig. 8. The striped module is given a procedural name, *NAME*, a cross-reference identifier *x*, and a number *n*, on its current chart. If the current chart identifier is *m*, then the box can be uniquely identified as the Dewey-decimal number *m.n*, and this number can be used for cross-referencing when no ambiguity arises. When that is the case, *x* need not appear at the point of striping.

Using top-down hierarchic-expansion methodology, one starts the design of the module at the next level with a functional description of the module and the conditions under which the several exits occur. He then proceeds to design an algorithm to perform the intended action using the usual canonic structures. In addition, he perhaps finds occasion to use one or all of the configurations of Figs. 5, 6, and 7. At some point, then, he breaks away from proper program constructs, to divert the flow of control to the alternate module exit(s). He does this by replacing a box normally appearing in a structure by an exit symbol, as shown in Fig. 9.

The flowchart which results has one *normal* (structured) exit point, and one or more *extra-normal* (unstructured) exits. It is worthwhile pointing out again that the extra exits may derive from perfectly normal non-pathological events. For example, when reading data from a file, it is a very common practice to read until an end-of-file indication occurs. Hence, the alternate exit from a box labeled "input from file" taken when the end-of-file occurs cannot be said to be an "abnormal" event. I shall refer to it rather as a *paranormal* exit (*para* from the Greek meaning "beside"), to differentiate it from the (normal) exit taken after the more usual, stated function (reading elements from the file) has taken place, and from a truly *abnormal* exit (one in response to an abortive event).

Paranormal events thus lie between the normal and abnormal; they are the simple "alternate exits" which should be allowed in the software designer's bag to permit him, among other things, to create modules which can recover efficiently from minor failures in the program or from erroneous input data.

On the flowchart of a multi-exit module, several occurrences of each paranormal exit might appear, as depicted in Fig. 10. How does such a flowchart stand in relation to the criteria I gave earlier? To me, flow through the chart does not appear disorganized, nor do any of the first four criteria seem violated; some branches just terminate early, back to an activity defined at a previous hierarchic level, that's all. The expansion of a multi-exit symbol as a separate flowchart is thus not objectionable, in my opinion.

However, if a multi-exit chart such as that in Fig. 10 were to replace its flowchart symbol at the previous level in the hierarchy, the new expanded chart would have crossing flowlines. A simplified case of this is illustrated in Fig. 11.

Non-planar flowcharts are particularly annoying to anyone trying to understand a program, because crossing flowlines detract from readability, reduce clarity and understanding, impair assessment of correctness, and attack the program organization generally. Flowcharts with on-page connectors to avoid the crossings are no better. Programming conventions which can lead to such difficulties are of questionable utility and are clearly a violation of the criteria I stated earlier.

The violation comes as the result of substituting the flowchart with paranormal exits back in place of the simple box at the earlier level. Neither of the flowcharts —that with the multi-exit box, nor its expansion at the next design level—is objectionable on a separate basis. For example, there is no objection in Fig. 10 being the next-level embodiment of box *A* in Fig. 6. But, there is objection to substituting Fig. 10 for box *A* in Fig. 6, because then the flowlines become jumbled.

The exit points of canonic structures, coded or flowcharted, are readily located, because they invariably either appear at the bottom, or else result as the immediate consequence of the loop test, at the top. Logical flow in nested structures having exists somewhere in the middle is naturally going to be harder to read and follow, even if the flowchart remains planar. Hence even if flowlines don't become jumbled as one flowchart replaces its box at the preceding level, the resulting chart is very apt to be less readible, because of the lack of uniformity in substructure exit locations.

These objections are somewhat at variance with the way canonically structured flowcharts at one hierarchic level can replace a striped symbol at the preceding level without violating the criteria given earlier. The way to avert such difficulty is clearly not to redraw flowcharts at one level, substituting flowcharts from the next level for multi-exit striped modules. Fortunately, this restriction is superficial in a top-down *design*, because flowcharts are developed *from* striped symbols, rather than vice-versa.

V. Coding Multi-Exit Structures

I have not addressed how the structures stand in relation to the fifth criterion (codability). Obviously, there are times when the coded procedure corresponding to a striped-module flowchart might need to appear directly in-line for speed efficiency, rather than a coded call to the procedure. In canonic structures, this presents no problem, but in multi-exit structures, there is again apt to be a problem identifying the connectivity of the code. Moreover, if it were deemed objectionable to do such replacement of flowcharts for striped multi-exit modules on a 2-dimensional medium, it seems to me even more objectionable to allow substitution of multi-exit code for procedure calls in the program, a linear medium.

For readability, the following modular coding formats are useful to implement the permissible extended program structures.

Capitalized words in the formats below identify macros for control structures to be translated into whatever language is being used to write the program. The italic symbols identify programming language strings: *c* is a condition (event) which causes the paranormal cessation of the procedure called by statement *s* or of the statement *s* itself; the statements $s_1, \cdots, s_n$ and $s_m, \cdots, s_p$ are nested modules of canonic and extended structures. The skeleton flowline annotations are added merely for readability. Translation of the formats above into programming language statements can be done manually or by an appropriate preprocessor, such as the CRISP processor (Ref. 5).

```
(1) IF NO c DURING s

    :->THEN  s1
    :        s2
    :        .
    :        .
    :        sn
    :        END

    :->ELSE  sm
             sm+1
             .
             .
             sp
             ENDBLOCK

    If c=c1, c2 ..., cn
    then use CASE
    structures below
    for ELSE module,
    with ENDBLOCK
    for final END:

        CASE c1:sm
             sm+1
             .
             .
             sp
             END

(2) WHILE NO c DURING s
    ↑   s1
    ↑   s2
    ↑   :
    ↑   sn
    ←←REPEAT

(3) UNTIL NO c DURING s
    ↑   s1
    ↑   s2
    ↑   :
    ↑   sn
    ←←REPEAT
```

```
(4) LOOP
    ↑   $s_1$
    ↑   $s_2$
    ↑   ⋮
    ↑   $s_n$
    ←←REPEAT IF $c$ DURING $s$
```

```
(5) LOOP
    ↑   $s_1$
    ↑   $s_2$
    ↑   ⋮
    ↑   $s_n$
    ←←REPEAT UNLESS $c$ DURING $s$
```

VI. Abnormal Terminations of Structured Programs

The multi-exit structures discussed so far will extend structured-programming techniques to cases where programming to handle events using canonic structures could prove counter-productive. However, there are *abnormal* contingencies encountered during a top-down design that may not have been fully identified at earlier levels as part of a module's normal function. Yet, in order for the program to perform correctly, the abnormal situations must be dealt with, and hopefully, *not* by redesigning the previous levels.

For example, it may be known intuitively ahead of time that some arithmetic operations can result in overflow-errors under certain (perhaps unknown) input conditions. But it may not be knowable, until an actual algorithm is designed, just where the overflows will occur, or what the input conditions that cause them will be.

In other cases, there may be knowable, specifiable contingencies which represent abnormal departures from the program's normal functionings, which the program must respond to (or recover from). A decision table drawn up for this program would likely classify such abnormal conditions into the "ELSE-rule" category—all cases not specifically defined by the program's intended behavior under normal, error-free input.

In some cases, recovery procedures can be instituted by the program itself; in others, operator intervention may be required. Different types of abnormalities will conceptually require entirely separate recovery procedures. For example, a program which generates a report from several files may conceivably be asked to complete the report because some identifiable parts of the report may yet be useful, even though one of the files continues to be read occasionally in error. However, in the same program, execution may be halted and control returned to the operator if one of the files cannot be found.

Abnormal exits to many unstriped modules are often overlooked because the abnormal exit is implied in the code for that module. A flowchart box labeled "A=B+C" would, for example, be coded in FORTRAN as "A=B+C"; but if A and B are large enough, an overflow trap automatically kicks the control to some error-handling procedure. Yet these connections are seldom put on the flowchart. Indeed, if such implicit actions were required to be drawn onto a flowchart, as in Fig. 12, few "structured programs" would exist. And imagine all the confusion trying to follow all the jumbled lines!

A similar statement holds concerning abnormal terminations of striped modules. In order for us to be able to design and program using what *appear* to be structured programming techniques, it is usually necessary for us to suppress the flowchart connections for abnormal situations, at least down to that design level where an abnormal event is sensed explicitly and an explicit branch to the recovery procedure appears. But if program modules—unstriped, as well as striped—may have abnormal contingencies whose connections may not appear in an explicit form at a given design level, then program response can only be fully and readily assessed if the conventions for suppressing the connections are easily remembered, fully understood, and rigorously adhered to.

Of course, it may be entirely possible that a program can invoke a recovery procedure and return to normal processing in a purely structured way. Such cases, even though induced by abnormal events, nevertheless can be handled by the normal and paranormal-exit structures already discussed. It is the others that must be covered by the convention.

The rule for displaying abnormal terminations which, to me, seems most in keeping with the first four criteria given earlier is the following: *Flowlines corresponding to abnormal terminations exiting from modules may be omitted at all hierarchic levels beyond that at which the recovery module appears on a structured flowchart and which also shows the flowline connection from the parent module which contains the nested submodule(s) from which the abnormal exit is made.*

Figure 13 depicts a chart at which a particular abnormal termination first appears. The recovery procedure appears as a module (here named RECOVERY) executed whenever the abnormal error event occurs in later levels. The exploded views of striped submodules of *B* being aborted do not show either the *error* condition or the module termination symbol labeled "RECOVERY" unless there is an explicit need to do so (e.g., when *error* is actually tested as an unstriped module), or unless showing them contributes to readability, understandability, assessment of correctness, etc. As the latter of these represents an optional case, the abnormal exit can appear merely as a comment, as shown in Fig. 14.

VII. Labeling Flowchart Exits

There is obviously a need for correct and consistent labeling of the exit terminals of a module flowchart, so that the reader can tell immediately and with certainty whether it is a normal, paranormal, or abnormal subprogram exit, or a subroutine return. Further, he must be able to locate the procedure next to be executed following the exit easily and unambiguously.

The conventions summarized in Fig. 15 (of which only a subset may actually be operable within a given system) contain an identifier within the terminal symbol, and in some cases, a module number designator which labels the point of continued activity. This number, denoted by *n* in the figure, can be optional whenever *n* is a module appearing on the same chart at the immediately preceding level as the current striped module being terminated, such as is true for cases (c) and (e) of the figure. It may be supplied to aid in locating the next procedure. The number is mandatory, however, for an abnormal exit to an unnamed procedure (case (f)) defined at an earlier level, or to a named procedure (case (g)) when there is more than one named abnormal procedure in the program. The former mandate is clearly one to identify program connectivity unambiguously; the latter is only for ease in locating the referenced procedure in the documentation.

VIII. Coding Module Extra-Normal Exits

A top-down program may be written, as I indicated earlier, in a format whereby each module has its entry at the top and a normal (structured) exit at the bottom. Any exits in between are either calls (transfers) to modular procedures (usually, but not always) farther down in the code, or extra-normal transfers to points within modules at previous design levels, always higher up in the code.

Calls can be classified by the data-space state upon initiation of the called procedure. For example, subroutine calls pass the return address and optional arguments to the subroutine procedure, often in a stack configuration. Coding for the normal exit (in the subroutine case, RETURN) reconfigures the data space for proper resumption of program execution. *The same consideration must be given to extra-normal exits.* (In the subroutine case, these exits must also unstack return addresses and arguments).

Abnormal terminations may transfer back through an arbitrary number of levels, all at once, to a program recovery procedure. Hence, any data-space assumptions in effect at the higher level must be restored prior to the transfer. Paranormal exits may likewise transfer back through a number of levels, but only one flowchart level at a time (although in an optimized object code listing, this could appear as a single jump after appropriate data-space recovery, as above).

Just as it facilitates flowchart readability and understandability to identify normal, paranormal, and abnormal exits separately (but consistently), it is likewise the case with the code corresponding to these exits. Unfortunately, most programming languages do not have separate branching statements for all the cases in Fig. 15. However, coding conventions may be adopted, either in the form of annotations or, better yet, macros, to effect and display the program module connectivity. Table 1 summarizes such a set of conventions.

IX. Conclusion

This paper has demonstrated that the concept of structured programming can be extended to multi-exit structures in a natural way. The methods preserve almost all of the advantages of structured programming: top-down development, hierarchic expansion, program modularity, and assessment of correctness. At the same time, they relax structural constraints to take advantage of more efficient program configurations than are possible with canonic structures.

Acknowledgment

I would like especially to thank Dr. James Layland for his review, comments, and helpful discussions during the preparation of this paper.

References

1. Böhm, C., and Jacopini, G., "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules," *Communications of the ACM*, Vol. 9, pp. 366-371, 1966.
2. Dijkstra, E. W., "Structured Programming," in *Software Engineering Techniques*, NATO Science Committee, edited by J. N. Burton and B. Randall. Kynoch Press, Birmingham, England, April 1970. Available from Scientific Affairs Div., NATO, Brussels, Belgium.
3. Mills, H. D., "Mathematical Foundations for Structured Programming," IBM Document FSC72-6012, Federal Systems Div., IBM, Gaithersburg, Md., February 1972.
4. "American National Standard Flowchart Symbols and Their Usage in Information Processing," ANSI-X3.5-1970, American National Standards Institute, Inc., New York, N.Y., Sept. 1, 1970.
5. Tausworthe, R. C., "Control Restricted Instructions for Structured Programming–CRISP," to appear in next issue.

Table 1. Module exit conventions

Type	Meaning
SYSTEM	Program termination, return control to system
STOP	Program termination, return control to operator
END *n*	Subprogram normal termination. Control transfers to module *n* at preceding level; *n* is specified optionally
RETURN	Subroutine normal termination. Control returns to calling module
EXIT *event* TO *n*	Paranormal exit. Control passes to *event* case, program module *n* at the preceding level; TO *n* specified optionally
ABORT TO *n*	Abnormal exit to module *n* at earlier level; *n* is mandatory
ABORT TO *ABNAME* AT *n*	Abnormal termination to module named *ABNAME*, numbered *n*; AT *n* is optional

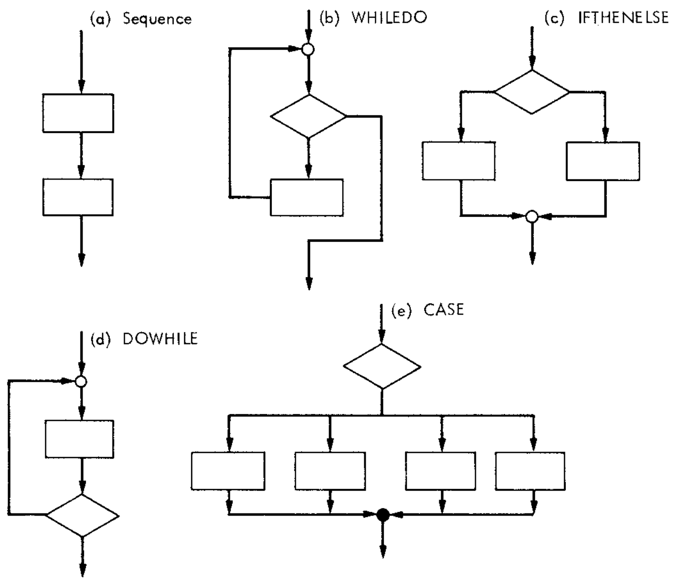

Fig. 1. Canonic program structures

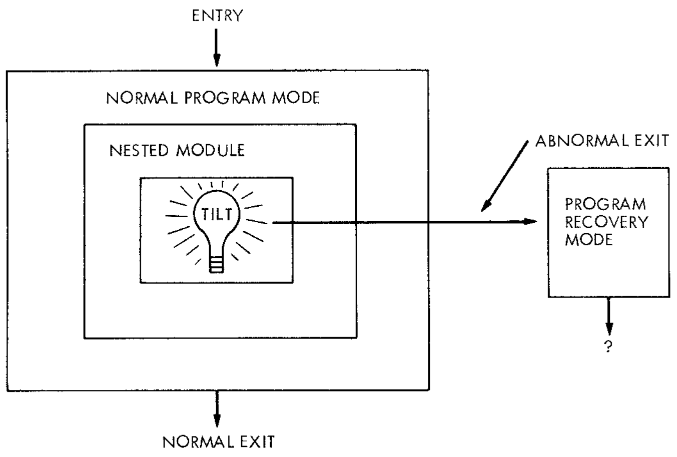

Fig. 2. Abnormal exit from a Nested Structured Program

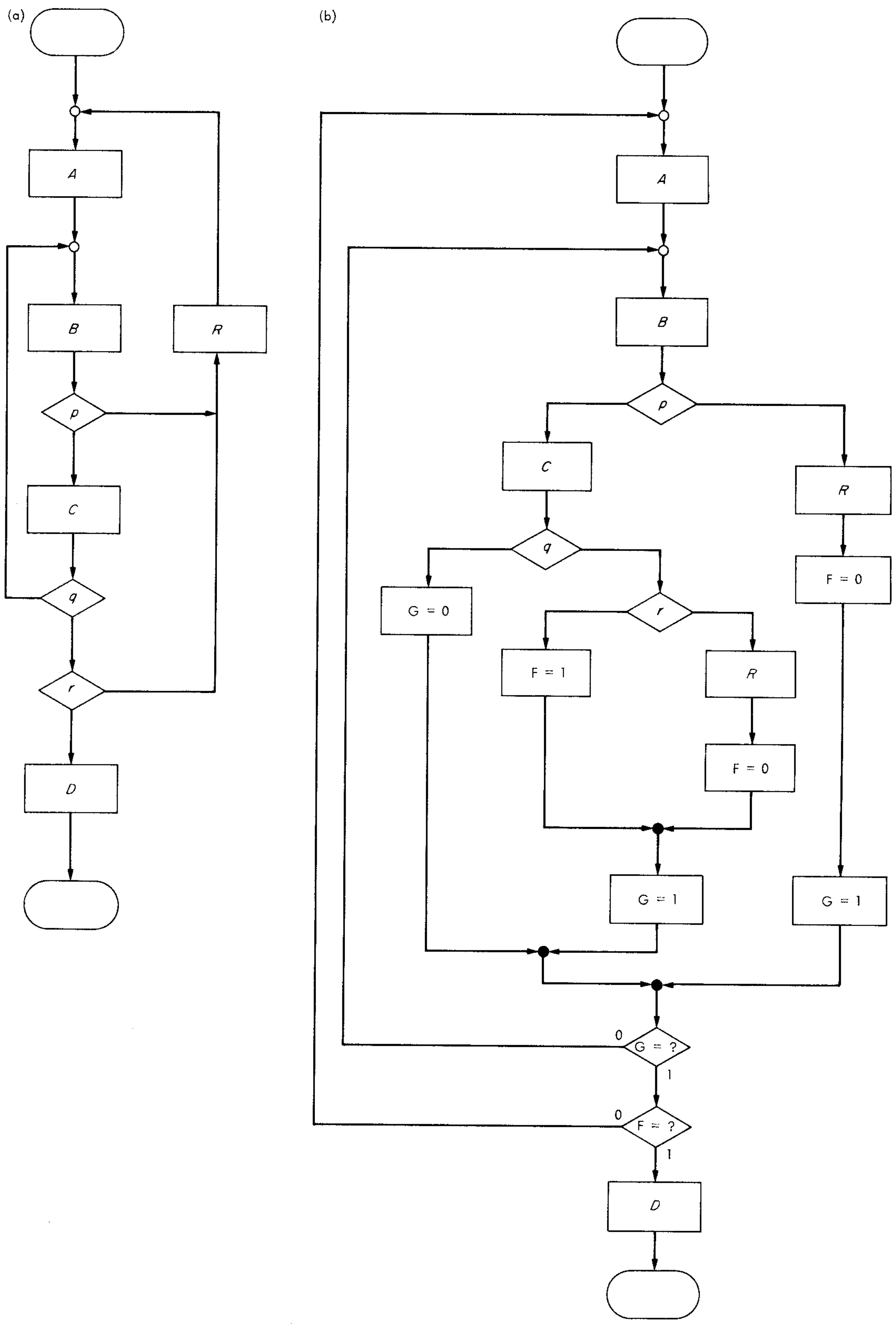

Fig. 3. Abnormal condition (a) unstructured program in which *p* and *r* are tests which indicate further execution is useless; *R* is recovery module which then initiates program restart, (b) structured form of (a)

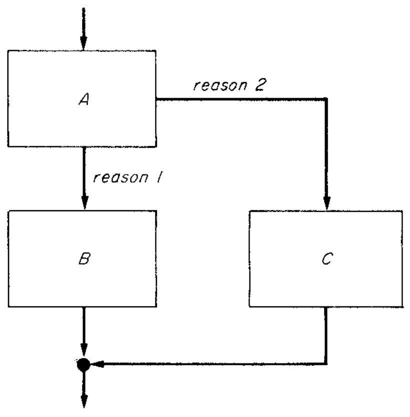

Fig. 4. Multiple exits configured into an IFTHENELSE-like structure

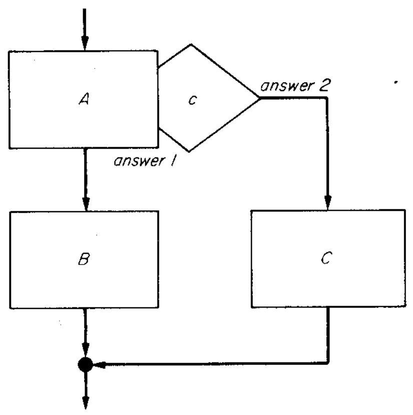

Fig. 5. Multi-exit program configuration with exit condition explicitly labeled

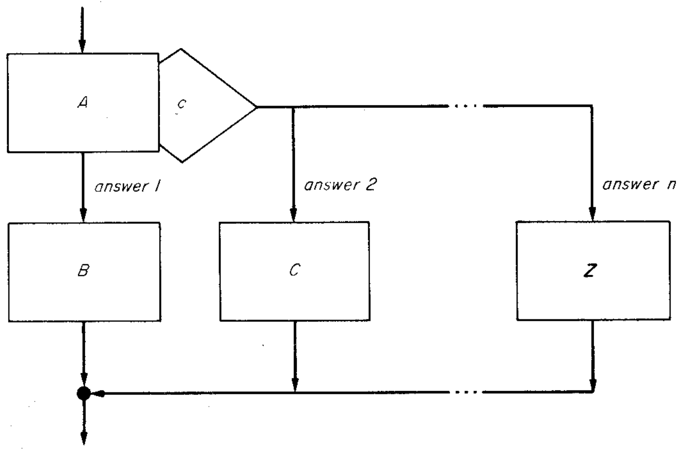

Fig. 6. Multi-exit CASE-like configuration with exit condition explicitly labeled

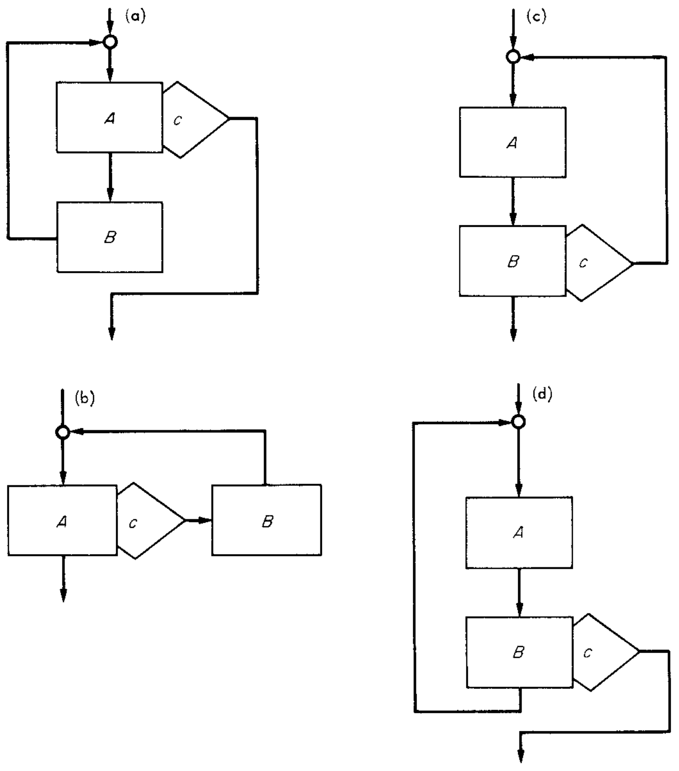

Fig. 7. Looping configuration with multi-exit structures

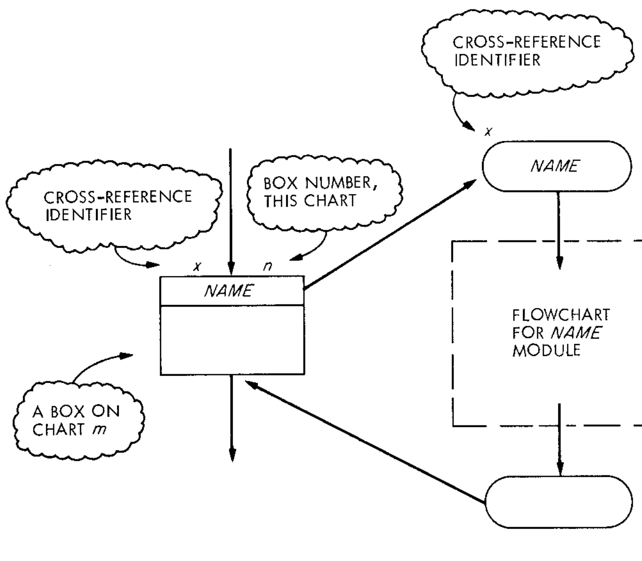
CROSS-REFERENCE IDENTIFIER
x
NAME
CROSS-REFERENCE IDENTIFIER
BOX NUMBER, THIS CHART
x
n
NAME
FLOWCHART FOR NAME MODULE
A BOX ON CHART m

Fig. 8. Hierarchic expansion of striped flowchart symbol

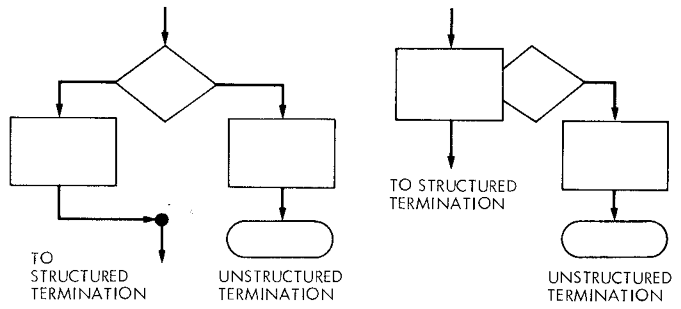

Fig. 9. Modes of generating multiple exits in otherwise structural programs

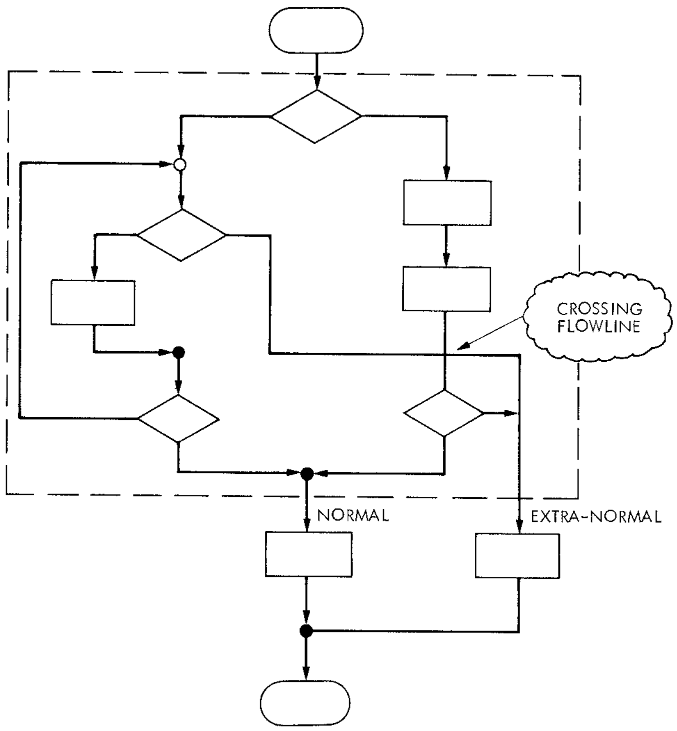

Fig. 11. Crossing flowlines can appear when the flowchart of a multi-exit box replaces the box

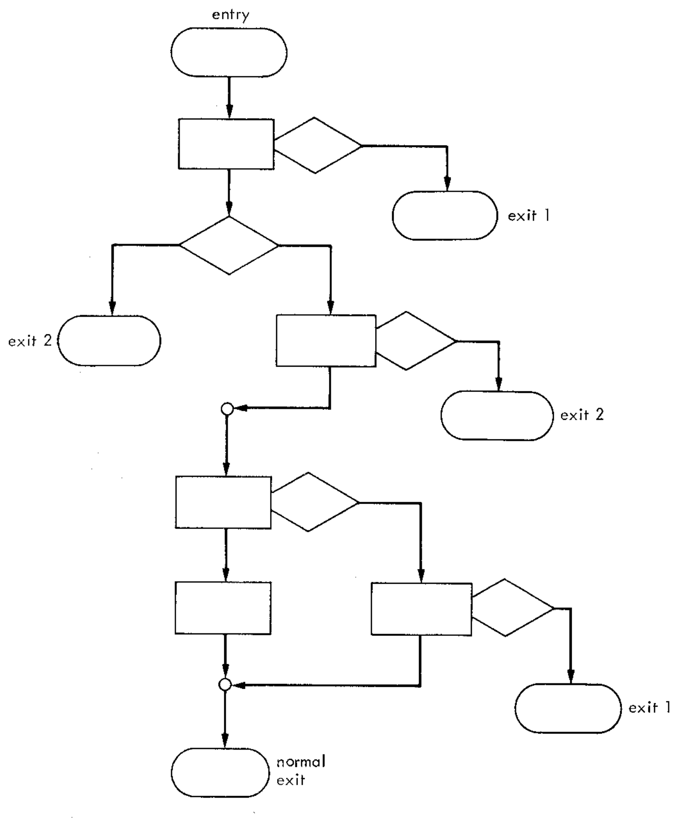

Fig. 10. Possible expansion of a module with two extra-normal exits

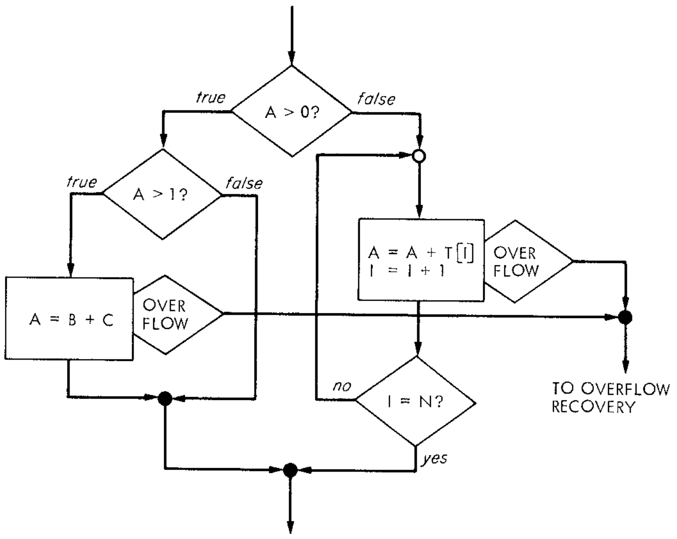

Fig. 12. Implicit abnormal contingencies in a simple "structured" program

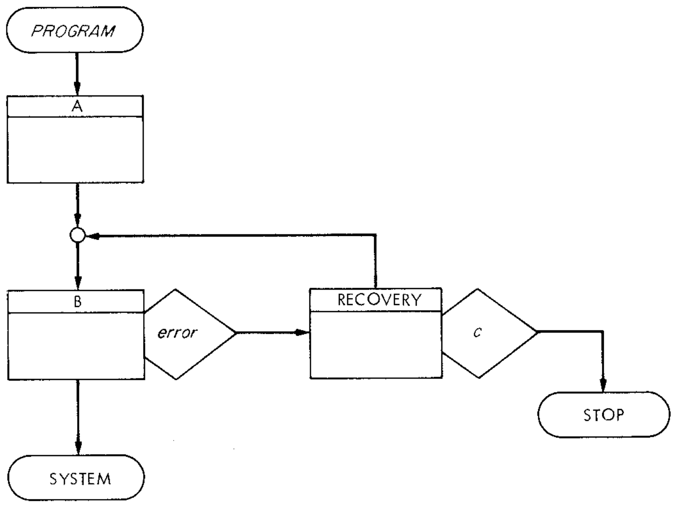

Fig. 13. A Program A THEN B, in which an occurrence of error during the execution of B initiates the RECOVERY procedure. If recovery under criterion c is possible, B is tried again; if recovery is not possible, control returns to the operator

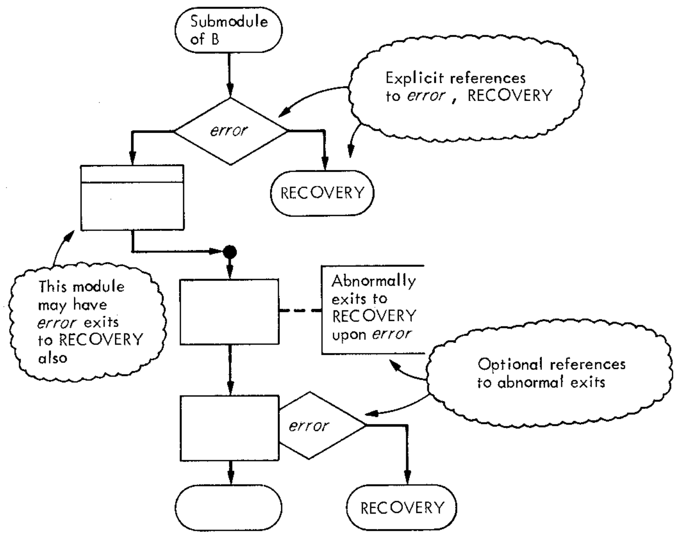
Submodule of B
Explicit references to *error*, RECOVERY
error
RECOVERY
This module may have *error* exits to RECOVERY also
Abnormally exits to RECOVERY upon *error*
Optional references to abnormal exits
error
RECOVERY

Fig. 14. Notation for abnormal module terminations at levels deeper than RECOVERY

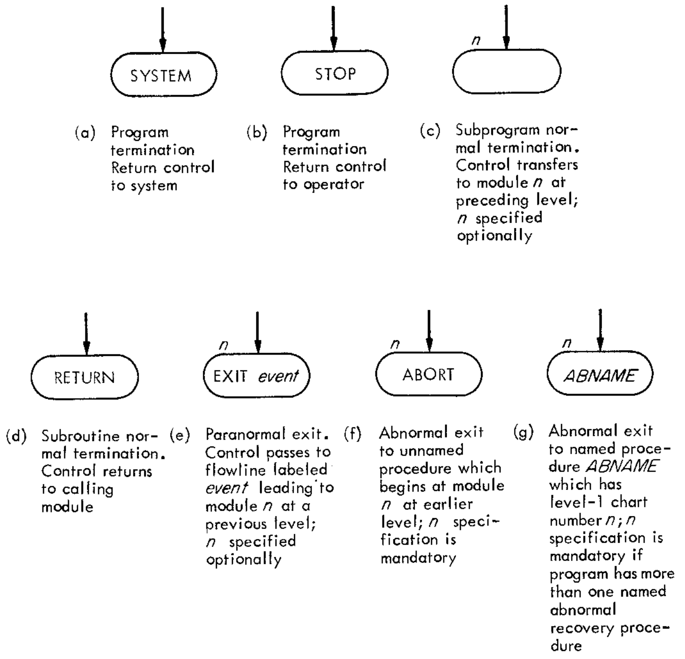
SYSTEM
STOP
n
(a) Program termination Return control to system
(b) Program termination Return control to operator
(c) Subprogram normal termination. Control transfers to module *n* at preceding level; *n* specified optionally
RETURN
n
EXIT *event*
n
ABORT
n
ABNAME
(d) Subroutine normal termination. Control returns to calling module
(e) Paranormal exit. Control passes to flowline labeled *event* leading to module *n* at a previous level; *n* specified optionally
(f) Abnormal exit to unnamed procedure which begins at module *n* at earlier level; *n* specification is mandatory
(g) Abnormal exit to named procedure *ABNAME* which has level-1 chart number *n*; *n* specification is mandatory if program has more than one named abnormal recovery procedure

Fig. 15. Module termination symbol annotation conventions