Structured programming in COBOL—the current options

J. M. Triance

Computation Department, University of Manchester, Institute of Science and Technology, Sackville Street, Manchester M60 1QD

The problems of writing structured programs in COBOL as defined by the Current Standard and by CODASYL are examined. Four versions of structured COBOL supported by preprocessors are presented and some criteria for judging preprocessors are proposed.

(Received March 1979; amended April 1979 and July 1979)

The principles of structured programming are now widely understood and applied. Central to the concept are the three basic program constructs: sequence, selection and iteration. It is usually presupposed that the program will be divided in communicating processes (or modules) as a consequence of the program design method (ref. top-down programming (Stevens, Myers and Constantine, 1974)) and inversion (Jackson, 1975). To some people structured programming also involves basing the program structure on the problem structure. This can lead to backtracking and a need to resolve structure clashes (Jackson, 1975). This paper makes no attempt to evaluate the different forms of structured programming but instead, for commonly held views of the technique, it examines the problems of using COBOL as the programming language.

The motivation for applying structured programming techniques is to produce programs which are easy to read and maintain. It is possible to design programs using these techniques and then translate the design into COBOL. But if, as is normally the case, the COBOL source listing forms a major part of the documentation it is important that the program's structure is readily discernible from the source listing. This cannot easily be achieved in current versions of COBOL, which were in general specified when the term 'structured programming' meant nothing to the COBOL community. In particular IF's cannot be used to implement all forms of nested selections, there is no construct for selections with more than two alternatives, there is no construct specifically designed

IF C1
  IF C2 S1
  ELSE S2'
END-IF
S3.

Fig. 1 Use of END-IF

approach 1:
  IF NOT C1 GO TO P1
  IF C2 S1
  ELSE S2.
  S3.
P1.
approach 2:
  IF C1 PERFORM P2.
    rest of in-line code
P2.
  IF C2 S1
  ELSE S2.
  S3.
approach 3:
  IF C1
    IF C2 S1 S3
    ELSE S2 S3.

Fig. 2 Equivalent coding in ANS 74 COBOL

for iterations, no construct for backtracking and the features for communicating processes are inadequate for some variations of structured programming.

CODASYL (1978) has recognised some of these problems and its solutions are presented in this paper. Since there will be a few years delay before these enhancements are generally available to COBOL programmers this paper also investigates alternative ways of implementing the new structures. Equivalent forms in American National Standard 1974 COBOL (ANSI, 1974) are presented. Four versions of structured COBOL are also investigated. They are Michael Jackson's Schematic Logic (1975; 1970; Triance and Yow, 1979), IBM's SCOBOL (1970), ADR's MetaCOBOL (1970) and Software Science's CASTLE (1970). All of these are supported by COBOL preprocessors.

Each feature of structured COBOL is considered separately.

Explicit scope terminators

END-IF has been introduced to CODASYL COBOL for the purpose of terminating the scope of a nested IF statement. This is needed when a statement (S3 in Fig. 1) which is within the scope of an outer IF statement is to be executed after each branch of the nested IF.

In this and subsequent extracts of coding C1 and C2 represent conditions and S1, S2 and S3 represent a sequence of imperative statements. P1, P2, etc. will be used for procedure names.

The equivalent coding in ANS 74 COBOL is shown in Fig. 2. If the coding is produced by automatic or semi-mechanical translation from a structured design language (such as schematic logic (Jackson, 1975) or pseudo-code (IBM, 1970)) then it will probably resemble approach 1. This coding in many respects is the closest to Fig. 1 but its correspondence to the original structure is obscure. Approach 2 will in many cases be undesirable because it enforces the wide separation of the nested IF from the outer IF. Approach 3 will normally be 'readable' but the repetition of S3 involves extra coding (it could be several lines) with the added opportunity for errors. In some cases of further levels of nesting this approach would
- ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE with SIZE ERROR Phrase
- CALL with ON EXCEPTION Phrase
- DELETE, START and REWRITE with INVALID KEY Phrase
- EVALUATE
- IF
- READ with AT END or INVALID KEY Phrase
- RECEIVE with NO DATA Phrase
- RETURN
- SEARCH
- STRING and UNSTRING with the ON OVERFLOW Phrase
- WRITE with INVALID KEY or END-OF-PAGE Phrase

**Fig. 5** CODASYL COBOL conditional statements

```
evaluate rec-code
  when 1 s1
  when 2 s2
  when other s3
end-evaluate
```

**Fig. 6** An EVALUATE statement

```
evaluate true
  when rec-code = 1 s1
  when rec-code = 2 s2
  when other s3
end-evaluate
```

**Fig. 7** An alternative form of EVALUATE

EVALUATE offers many more options including the specification of any COBOL condition after each key word WHEN. Each condition is evaluated to determine whether the sequence of statements immediately following is to be executed. Fig. 7 shows the statement in Fig. 6 translated into this form.

A better idea of the full power of EVALUATE can be obtained by using it to implement a decision table. Fig. 8 shows a simple decision table and the equivalent EVALUATE statement. As can be seen the Evaluate statement is simply the decision table turned through 90 degrees with the hyphen replaced by ANY, Y by TRUE, N by FALSE and ELSE by OTHER. The question of the suitability of EVALUATE for implementing decision tables is outside the scope of this paper.

In current COBOL there is no multi-branch multi-join statement so the programmer must use multiple IF statements, or GO TO with the DEPENDING option. The example from Figs. 6 and 7 is implemented in Fig. 9 using these two approaches. Neither of these implementations is self-documenting as a multi-way selection. Furthermore they both depend on GO TO's to achieve a common end point. This is a possible source of errors.

It is true that GO TO's can be avoided in the first approach by using nested IF's. But since this employs a nested solution to an unnested problem it reduces readability.

The second approach depends on the selection criteria being

<table>
<thead>
<tr>
<th>Conditions</th>
<th>1</th>
<th>2</th>
<th>ELSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEX</td>
<td></td>
<td>MALE</td>
<td>FEMALE</td>
</tr>
<tr>
<td>AGE &gt; 60</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>AGE &gt; 65</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 8** A more complex EVALUATE statement

Multi-way selections
CODASYL COBOL now possesses a multi-way branch statement for use in situations where the two-way branch offered by IF is unsuitable. It is the EVALUATE statement illustrated in Fig. 6. When the data item REC-CODE is equal to 1 the sequence of statements S1 is executed, when it is equal to 2 S2 is executed and in all other cases S3 is executed.

**Fig. 9** Multi-way selections in ANS74 COBOL
Schematic Logic:
SELECT REC-CODE = 1
  S1
OR REC-CODE = 2
  S2
OR
  S3
END

SCOBOL:
EVALUATE REC-CODE
CASE 1
  S1
CASE 2
  S2
CASE OTHER
  S3
END-EVALUATE

MetaCOBOL:
SELECT FIRST ACTION
WHEN REC-CODE = 1
  S1
WHEN REC-CODE = 2
  S2
WHEN NONE ARE SELECTED
  S3
ENDSELECT

CASTLE Format 1:
SELECT REC-CODE
CASE 1
  S1
CASE 2
  S2
OTHER
  S3
CSELECT

CASTLE Format 2:
IF REC-CODE = 1
  S1
ELSE IF REC-CODE = 2
  S2
ELSE
  S3
CIF

Fig. 10 Multi-way selections supported by preprocessors

expressed as integral values in a limited range. An alternative, which overcomes this problem in some cases, is to replace the GO TO DEPENDING by a normal GO TO and preset it to the required procedure-name by means of an ALTER verb. However, this verb is generally regarded as the least readable of all COBOL verbs and is due to be deleted from the next standard. Programmers are thus well advised to avoid ALTER.

The preprocessors under discussion all provide multi-way selection statements. Their formats are illustrated in Fig. 10 using the example from Fig. 6. In more recent versions of Schematic Logic OR is replaced by ALT to avoid any confusion with OR in COBOL conditions.

In SCOBOL there is an alternative format which permits the specification of COBOL conditions with each CASE so the selection is not limited to integral values of a data item. On the other hand the CASTLE SELECT statement takes the more limited form. Thus their second format must be used for the more general case. This format is merely a consolidation of COBOL's nested IF with ELSE and IF concatenated.

If the Schematic Logic results from the application of the Jackson Method the alternatives will be mutually exclusive. No such assumptions are made with the MetaCOBOL SELECT. The format shown in Fig. 10 will ensure that only the action associated with the first true condition will be executed. It is also possible to specify LEADING ACTIONS to execute all the actions until a condition which is untrue is encountered and EVERY ACTION to execute every action for which the associated condition is true. It is possible to specify further actions which are to be executed when ANY of the conditions have been met or when ALL the conditions are true.

Iterations
CODASYL COBOL now contains an in-line looping construct. It is a variation of PERFORM illustrated in Fig. 11. The action of this statement is to test the condition Cl and if it is false execute the string of statements S1. This process is repeated until Cl becomes true whereupon control is transferred to the statement following END-PERFORM. The statements constituting S1 can be imperative statements as we know them or another inline PERFORM or any of the statements shown in Fig. 5 provided they are accompanied by the relevant scope delimiter (END-IF, END-EVALUATE, etc.).

In current implementations of COBOL there are two ways of coding iterations. They are illustrated in Fig. 12. Approach 1 becomes unreadable when S1 is a long string of statements (possible with nested iterations and selections). Both approaches are vulnerable to the shortcomings of the GO TO statement although the GO TO can be avoided in the second approach at the cost of placing S1 out-of-line (at the end of the Procedure Division).

If S1 represents another level of refinement in the top-down design of the program then it is, of course, desirable to place it out-of-line. The traditional out-of-line PERFORM thus has a continuing role and it will not be removed from COBOL in the foreseeable future.

The equivalents of this iteration in the preprocessor versions of COBOL are shown in Fig. 13. SCOBOL requires the con-
The Computer Journal Volume 23 Number 3

is true control is immediately transferred to the statement after each execution of SI the condition Cl is tested and if it is untrue, by S2 and, if C2 is untrue, by S3. Control would then transfer to the statement after END. If, however, C1 is true control is transferred to S4 immediately after S1. Similarly, if C2 is true control is transferred to S4 immediately after S2.

There is no equivalent feature in any of the other preprocessors or in CODASYL COBOL. An implementation of the example in ANS 74 COBOL is shown in Fig. 19.

Communicating processes
An important aspect of larger programs which are designed using structured programming constructs is that they are

**Fig. 17 QUIT in Schematic Logic**

**Fig. 18 Schematic Logic POSIT construct**

**Fig. 19 Implementation of POSIT in ANS 74 COBOL**
divided into modules. Each module represents one stage in the top-down design (or stepwise refinement) of a program. Great emphasis is placed on the independence of these modules (IBM, 1970; Stevens, Myers and Constantine, 1974) of these modules.

In ANS74 COBOL the modules can be paragraphs or sections within a program which are PERFORMed or separately compiled subprograms which are CALLeed. PERFORM offers the minimum of independence:

(a) no local data
(b) there is no parameter passing mechanism
(c) modules can physically overlap
(d) it is possible to branch from the middle of one module to the middle of another
(e) return links to the calling module can be corrupted when overlaying

CALL, on the other hand, offers almost a maximum of independence:

(f) all data is local
(g) the only communication of data is via parameters
(h) each module is a separate program
(i) control can only be transferred to another module by CALL or by EXIT PROGRAM which returns control to calling program.

CALL thus has a number of advantages over PERFORM but in some implementations it is extravagant on computer resources. There is also the overhead for the programmer who must code all four divisions for each module and must define the parameters in both the calling and the called program. This overhead is removed in CODASYL COBOL which makes all divisions optional with the exception of the Identification Division (a minimum of two lines) and permits Global Data. This latter feature while making modules easier to write is a major threat to module independence.

Parameters in a CALL statement in CODASYL COBOL may be specified as being passed ‘by content’. This means that such parameters in the calling program cannot be altered by the called program. This saves the programmer having to take a copy of a parameter before it is passed in those cases when he wishes to protect the parameter from corruption by the called program. To this extent the feature aids module independence.

The Jackson method requires a different kind of module—designed to communicate with other modules via sequential files. Thus data is passed from one module by a write statement and received in another by a read. The execution of these modules can be synchronised so that as soon as a record has been written by one module that module is suspended and the other module is invoked so that it can read the record. When this second module has processed the record it can return control to the original module and wait for another record to be written. When implemented this way only one record is in existence at any time and it can thus be kept in central storage—it is in fact passed from one module to another like a parameter. Indeed these communicating processes can be implemented by replacing the WRITE by a CALL statement and the READ by an EXIT PROGRAM. Additional coding is however required to ensure that the READ is fully simulated. This is done by ensuring that on re-entering the subprogram control is immediately transferred to the statement following the most recently executed EXIT PROGRAM. A comprehensive description and justification of this mechanism, known as inversion, is given in Jackson’s book (1975). The generation of this somewhat ungainly code is automated by Jackson’s preprocessor (1970).

There is a problem that ANS74 is ambiguous and CODASYL COBOL is clearly unhelpful in maintaining subprograms in their last used state. CODASYL states that all PERFORM links must be initialised whenever EXIT PROGRAM is executed. Thus, any active PERFORM’s are corrupted as a result of executing the coding which simulates a READ statement. The solution usually adopted is to avoid PERFORM in inverted programs although some current implementations do not have this problem.

CODASYL have plans to introduce asynchronous processing using the telecommunication verbs RECEIVE and SEND rather than READ and WRITE. It is likely that inversion will be implementable in terms of this facility although it will provide more power than inversion requires and is thus likely to involve additional overheads. Because of the delay inherent in the COBOL development process this facility will not be generally available in the next decade.

The user’s current options

One option open to users is to persist with their current compilers until the CODASYL features are implemented. These features are not, however, likely to become generally available until the late 1980’s although it is likely that some compiler writers will regard structured programming constructs to be a special case and treat them with greater urgency.

For those users who are not willing to wait the only other option is to use a preprocessor. The relative merits of the four versions of structured COBOL which have been discussed can be summarised as follows:

Schematic Logic
This is the obvious choice for users of the Jackson method: the best known preprocessor, JSP-COBOL, supports inversion and backtracking as well as the three basic structures. Other users of schematic logic might wish to confine themselves to the three basic structures. JSP-COBOL is available for IBM 370-style architecture machines, Univac 1100 and ICL 1900 and the smaller 2900’s.

SCOBOL
This is the closest to CODASYL COBOL and thus offers advantages in forward compatibility. It is available on IBM equipment.

MetaCOBOL
This is a macro processor with the structured programming constructs being supplied as prewritten macros. Users can amend the existing macros or write new ones to enhance the version of structured COBOL offered. However, writing macros of this type is by no means trivial. MetaCOBOL is available on IBM equipment.

CASTLE
This offers a wide choice of constructs including three for iterations (WHILE, REPEAT and LOOP). Each user installation would be well advised to choose a limited subset of constructs if one can be found to suit their purposes. CASTLE is available on the larger IBM ranges, Univac 1100 and ICL 1900 and the smaller 2900’s.

There is a reluctance on the part of some users to use preprocessors. This is often the result of experiences with ill-conceived or badly written ones. Before acquiring any preprocessor it is thus worth checking it against the criteria in the following section.

Criteria for preprocessors

Provision of required facilities
All the preprocessors discussed in this paper support additional
features. Some of these will be useful; others might even be counter-productive.

Extensions forward compatible
Wherever possible the extensions supported by a preprocessor should be compatible with ANS COBOL or CODASYL COBOL. This will save conversion when the required extensions are eventually supported directly by the COBOL compiler.

This requirement will sometimes be waived when the user takes a conscious decision to depart from ANS COBOL or CODASYL COBOL because they are deemed to be inadequate.

Extensions fully supported
The preprocessor should apply complete syntax checking to the extensions and produce meaningful error messages for any errors.

No unreasonable restrictions on COBOL
Most preprocessors place restrictions on the COBOL which they will accept. Two examples which are likely to cause only slight inconvenience are the requirement that each COBOL verb is the first word on a line and the introduction of additional reserved words. Most preprocessors reserve a whole class of words for generated names. These might be all words with some unusual prefix, such as ZZWXX, or all those containing two consecutive hyphens or all those over 25 characters long. Many preprocessors do not publicise the particular restriction and simply rely on the unlikelihood of anyone transgressing it. Transgressions of any of these restrictions should be reported by the preprocessor.

The programmer’s use of the COPY facility might also be restricted. Any code which is placed on a COBOL library is copied at compile time thus placing it outside the control of the preprocessor. For the structured programming preprocessor this would be a particular problem for Procedure division coding. The ideal solution is for the preprocessor to support the COPY facility. Failing this the programmer would have to write the code for the library in unstructured COBOL or avoid the use of COPY in the Procedure Division.

Reference to original source listing sufficient
The extended version of COBOL offered by the preprocessor is the programmer’s source language. Looking at the code generated by the preprocessor is just as undesirable as looking at the object code generated by the compiler. It should thus be unnecessary. Three reasons why the programmer might need to refer to the generated code are:

(a) to understand the semantics of the extensions
(b) to interpret a compiler diagnostic message
(c) to interpret a run time diagnostic which refers to the generated code

The first problem does not arise if the preprocessor has adequate documentation. The second problem is removed when the compiler-produced error diagnostics can be interpreted by reference to the original listing but there appears to be no simple and comprehensive way of achieving this. An alternative approach, which carries a significant overhead, is for the preprocessor to perform a complete syntax check on the original source. This makes the compiler syntax checking redundant. The third problem arises when debugging software exists which relates the errors to the compiler’s input. Ideally the preprocessor should match this level of support for its input. Failing that the programmer can resort to providing their own tracing or monitoring with DISPLAY statements.

Moderate requirements of computer resources
There is an obvious overhead at compile time—often the preprocessor takes longer to run than the compiler. There is also a possibility that the generated program will be slower and occupy more storage than a program written directly in pure COBOL. In establishing whether these overheads are acceptable the likelihood of fewer test runs and greater programmer productivity resulting from the enhanced version of COBOL must be allowed for.

Compatibility with other systems
COBOL is supported on all moderately sized computer systems. This is not true of any of the preprocessor supported features. If portability is required the availability of the preprocessor on hardware and operating systems used currently (or planned) is an important factor.

Compatibility with other preprocessors
With the widespread use of preprocessors some users may find they need to run each source program through more than one preprocessor. This rather ridiculous situation can easily arise since some telecommunication systems and many data base systems are implemented by preprocessors. Such a proposition should be approached with severe caution for although preprocessor writers are obliged to provide compatibility with a particular manufacturer’s version of COBOL (and subsequent enhancements of it) they are unlikely to guarantee compatibility with other preprocessors. A better solution is to use a more comprehensive preprocessor, perhaps based on macros, which can handle all the desired extensions—provided that such a preprocessor can be found.

For installations which are unable to, or do not wish to acquire a preprocessor it is feasible to produce one internally. A large number of such projects have failed in the past because insufficient attention was paid to the design of the extensions and the usability of the preprocessor. In view of this the preceding checklist could usefully be applied to any such project.

Acknowledgements
The author is grateful to the referee and members of the BSI COBOL Working Group for their valuable comments on the first draft of the paper.

Conclusions
COBOL as defined in the current standard is not suitable for writing self-documenting structured programs. CODASYL has recognised the problems of coding selections and iterations and has enhanced COBOL accordingly. But these enhancements will not be available for a number of years. In the meantime, these control structures can be supported by preprocessor. This is a perfectly satisfactory short term solution provided that sufficient attention has been paid to the design and implementation of the preprocessor.

There are no enhancements in CODASYL COBOL for Jackson’s backtracking and inversion. For users of these features preprocessors appear to be the best solution for the foreseeable future.
Book reviews

Introducing the Electronic Office, by S. G. Price, 1979; 161 pages. (NCC, £8.50)

According to the preface 'this book is a scene-setter, exploring possibilities, benefits and problems, and outlining the products available today'. The book is divided into six chapters covering: environment and scope; what will the electronic office look like; products and services available; hindrances and challenges; implementation; future. There is a foreword by Clive Jenkins, a bibliography, a glossary of acronyms and an index.

Overall this book is rather disappointing. Despite the claims made in the foreword and on the cover it is particularly weak on 'the significance of the people factors' of which discussion is both brief and lacking in conviction. The structure of the book could be improved by eliminating unnecessary repetition and by re-ordering some of the material. On occasion it reads as though it was written in sections and put together in a hurry. In commenting on current data processing systems, there is a tendency to suggest that little has been learnt and no improvements taken place since the mid-sixties, a tendency which experienced data processing practitioners may find irritating.

The author is in favour of the electronic office and its early implementation. His conclusion is 'Research and development investment in the office is not the accepted practice today and experimentation is expensive but the acquisition of experience now by experimentation will enable the full opportunities of tomorrow to be taken'. He is clear also on the likely costs and difficulties and on the need to verify the purposes to be achieved and to redefine requirements to be met, by the electronic office. Accepting the author's evidence on the latter would allow the reader to reach a quite different conclusion, i.e. verify and redefine as though for automation but do not proceed to implementation until a cost-effective electronic office is available.

Despite its limitations (and its price) for anyone interested in the role of the office, present and future, this book offers some useful information and poses some pertinent questions.

T. G. Gough (Harrogate)


This book is intended as a text book for a comprehensive course in software engineering or as a broad introduction to the subject. It assumes a knowledge of basic computer terminology. The introduction defines the scope of the subject and stresses the important roles of communication and creativity in this discipline. Section 1 deals with project management and is strongly oriented towards the methodologies of the Hughes Corporation by whom all the contributors are employed. A typical development cycle is discussed and related to an interesting energy/entropy model. Throughout there are frequent references to an up-to-date bibliography.

The next section on software design emphasises that this is a problem solving process and that creativity as well as technical expertise and experience are necessary to achieve good design. It discusses the dynamic nature of the problem and the roles of operating systems, hardware and data bases in the design process. There is much good material here about such matters as modularity and cost assessment using weighted matrices but the close association with the particular procedures of the corporation results in a long and somewhat complex discourse. To obtain the full benefit of the ideas discussed committed and diligent reading is necessary together with access to the references. Structured programming is dealt with next in some detail with frequent references to the well known texts. There are, I think, too many trivial examples and the repeated illustration of each construct discussed in a number of languages could well have been left as an exercise for the reader. Section 4 on validation and verification points out the varying interpretations of these terms. Software testing is discussed and the use of simulation and automation, all in a broad context. Testing is illustrated by an example using a test tool of some sophistication but one that is unlikely to be available to many readers in this country. Despite the brevity of this section it provides a useful survey and invokes a number of useful ideas. The section on privacy and security starts with a discussion of the problems and some approaches to their solutions. OS/360, Multics and Hydra are used to illustrate different systems. Methods of user authentication are discussed and illustrated. A fascinating section on the use of cryptography follows in what I felt to be excessive detail. An extension of the summary on privacy and security would have been more appropriate. The set of references to this section seemed to me less up-to-date than in the earlier chapters. The final section on the legal aspects of setting up and operating a software house under US law must obviously have limited relevance to the great number of readers in this country. It does however contain a number of useful checklists of general interest. A short appendix on education and curriculae seemed too brief to be of much use. A cheaper edition, it is worth pointing out, has been produced, is nicely bound and has few printing errors—honest value for money.

Bernard Coleman (Wolverhampton)


This is a report of a WG5.2 workshop held in France in 1976. The book is in three parts. The first and largest part is a selection from contributed papers and some very terse notes of the discussions on those papers. The second is a condensation of the working papers of subgroups, giving the subgroups' tentative recommendations on their allotted topics. The third part is the report of the workshop, containing guidelines and recommendations.

Many of the participants felt that talk of methodology in graphics, let alone talk of standardisation, was premature. Much of the workshop, and its most important result, was concerned with groping towards a conceptual basis on which useful abstractions could be built. 'The most important result is the recognition and acceptance of the need for a clear distinction between a graphics subsystem (called the core) and a modelling subsystem (called also the geometric subsystem)' (page 201).

The appeal of this book is not to those whose interest lies in software development methods or who seek an introductory overview of the field of computer graphics: it is to those who attended—or would have liked to attend—the workshop itself.

M. A. Jackson (London)