

Measuring commercial program suite complexity

D.J. Leigh & E. Halliday

*School of Computing, Staffordshire University,
Stafford, ST18 0AD, UK*

ABSTRACT

COBOL is a very widely-used computer language, even in the 1990s. Its age is reflected in the ease with which poor initial programming practices are possible, even though the language does allow correct structuring. This may be contrasted with those languages of more modern derivation, where encapsulation, abstraction and data-hiding capabilities encourage correct approach. A large burden of maintenance also comes with older and less well-documented program suites. As more and more modifications are made, the "spaghetti" nature of its code can become emphasised.

To avoid this difficulty, monitoring of the complexity of the code can give early warning of troubles to come. Complexity measurement has generally been undertaken in languages such as FORTRAN, as being more readily available, and more "scientific" in their approach. A similar approach to a non-"scientific" language has not, so far as is known, been attempted. The apparently free-form nature of COBOL can, however, be analysed efficiently, and the measurements applied throughout suites of programs.

The structure of programs written in ANSI Standard COBOL (1985) ² is described. The measurement of complexity for individual programs written in COBOL is discussed. The alternatives available for appropriate evaluation of complexity are investigated. The extension of complexity measurements to a suite of associated programs is described. The requirements of meaningful results are evaluated, with the practical and commercial uses to which they may be put. The application of these metrics to an existing program suite are described, with the effects arising from their use. The conclusions relating to their effectiveness, and further refinements to be made, are given.

376 Software Quality Management

INTRODUCTION

One of the requirements in well-engineered programs and suites of programs is that they be maintainable. (This term is here assumed to refer to any changes which are to be made to a program, from whatever source. Changes in users' requirements lead to a need for changes in software as much as those arising from the detection of any errors in programs⁷.) So that changes to achieve the required effect may be made efficiently, the program code should be well-presented, and be structured in as simple a manner as possible.

This relates to many aspects of the program, some of which are to do with only its presentation: the layout, the use of embedded comments, the relevance of data-names, and so on. These should be objects of good design practice in the same way as all other, overall program production practices. These aspects are capable of measurement against appropriate metrics in an obvious way.

There are other measurements relating to good programming practice which are less to do with the appearance of a program, and much more to do with its intrinsic behaviour. These are typically gathered under the title of program complexity, and have reference to the ease with which a maintenance programmer may understand the implicit structure of the program.

It should not be assumed that poor practices are necessarily restricted to the initial production of a program, although they may be present from its inception. During any changes which are made to program text, poor layout, inappropriate names, and other unfortunate attributes may be introduced on any occasion. It is therefore in the interests of all efficient program management teams that the worth of the programs under their care are monitored on a continuing basis.

In terms of presentation alone, the application of simple tools may be sufficient (for example, the utility "cb" for C programs). Where the considerations relate to the complexity of the program, however, neither the measurement nor the correction of poor practices may be undertaken so easily. In the first place, it is necessary to define appropriate metrics, and then to evaluate them, before being able to make use of them. It is the purpose of this paper to describe the evaluation of a suitable structural metric in terms of the COBOL programming language, and to extend that description to suites of COBOL programs.

THE STRUCTURE OF COBOL PROGRAMS

COBOL programs are hierarchically structured². They consist of four *DIVISIONS*, each of which consists of one or more *SECTIONS*, defined either explicitly or implicitly. These are further refined into *paragraphs*, *sentences* and *statements*. These structures are made up, in turn, of *words* of one or more

characters. This hierarchy is supposed to reflect the structure of the English language.

IDENTIFICATION DIVISION.

...

program name, authorship, etc.

...

ENVIRONMENT DIVISION.

...

computer configuration, input-output definitions

...

DATA DIVISION.

...

data definitions for the program

...

PROCEDURE DIVISION.

...

program statements

...

Figure 1 - COBOL program structure

The first two *DIVISIONs* identify the program (*IDENTIFICATION DIVISION*) and describe its external interfaces (*ENVIRONMENT DIVISION*). In the great majority of COBOL programs, these serve only to define file interfaces and the control of components of the program. The third *DIVISION* defines the data structures to be used within the program (*DATA DIVISION*). In terms of complexity measurements, they contribute little significant information. The program statements are specified in the *PROCEDURE DIVISION*, which completes the program, and forms the basis for the evaluation of the associated complexity metrics. An outline of the program structure is given in figure 1.

All data storage for variables within a COBOL program is defined within the *DATA DIVISION*, as a single-level storage area, the size of which can be defined at compilation time. In some implementations, it is possible to define dynamic structures, but implementors have sufficient information supplied by each COBOL program to render such "advanced" techniques transparent, so that the overall requirements can be specified to the operating system when the program is loaded for execution.

Within every COBOL program, COBOL words are divided into two classes: those defined by the language, and those defined by the programmer. There are over three hundred of the former, and they may not be redefined by the programmer. They can, therefore, be used as fixed elements when analysing the structure of a COBOL program. (The words given in upper-case letters in

378 Software Quality Management

figure 1 are COBOL-defined words of this nature.) In particular, the use of a subset of these found within the *PROCEDURE DIVISION* - the COBOL "verbs" - allows efficient recognition of program structures within legal COBOL programs.

The English-like nature of a COBOL program is further accentuated by the use of fullstops, commas and semicolons as punctuation markings. The fullstop is used as a sentence delimiter, while the others are essentially for purposes of enhancing readability. An example of a fragment of program code, taken from the *PROCEDURE DIVISION* of a COBOL program, is given in figure 2. Its presentational characteristics, with upper-case letters throughout, the use of hyphenation to construct identifiers, and that new statements commence at the beginning of a new line, is typical of the style adopted by most COBOL programmers. Since these are not language requirements, however, use cannot be made of them when analysing COBOL programs.

```
...  
ADD 1 TO WB-LINE-COUNT;  
IF WB-LINE-COUNT > WD-PAGE-LIMIT,  
    PERFORM P58-NEW-PAGE-ROUTINE.  
...
```

Figure 2 - Fragment of COBOL program code

Each COBOL statement within the *PROCEDURE DIVISION* commences with a COBOL verb - examples given in figure 2 are *ADD*, *IF* and *PERFORM*. Each statement terminates at the next COBOL verb, or at the end of a COBOL sentence (which is marked by a fullstop). Thus, the statement structure is determined to an overwhelming extent by the COBOL verbs alone. There is, however, a subclassification of the COBOL statements into two: the *imperative* and the *conditional*. Because each statement contains precisely one verb, these verbs are similarly classified into *imperative* and *conditional*. In the example of figure 2, *ADD* and *PERFORM* are *imperative* verbs, and *IF* is a *conditional* verb.

```
...  
ADD 1 TO WF-PERSONNEL;  
ADD WG-PAYMENT TO WC-TOTAL,  
    ON SIZE ERROR PERFORM P42-RECOVER-POFLO.  
...
```

Figure 3 - Alternative uses of the *ADD* verb

It is possible for a given verb to be either *imperative* or *conditional* according to its context. If the action performed may lead to more than one conclusion,

according to the circumstances, the verb is necessarily *conditional*, even though it may appear *imperative* under normal interpretations of its actions. An example of this is given in figure 3: the first example of the *ADD* verb is *imperative*, and the second *conditional*. (The inclusion of the *SIZE ERROR* clause has effected the change.)

MEASUREMENT OF COMPLEXITY

As referred to above, the goal of every program designer should be to keep program structure as simple as possible, whilst providing the functionality required by the problem specification. Some straightforward measures have been proposed, which have the advantages of ease of evaluation within a program or a suite, such as lines of code delivered, or total number of user-defined names employed. Whilst these metrics have been widely employed, sometimes in a contributory fashion (see, for example, the COCOMO method), they are of lesser worth than alternatives. There is usually an implicit requirement that metrics in commercial applications be easy to evaluate; otherwise they will not be employed. (The two examples given form part of a metric proposed by Halstead referred to below.)

In general considerations of program structure, not dependent on the language employed, programs may be characterised as consisting of paths joined at nodes. The paths can only be traversed in one direction, that of sequential execution. Each path has a single entry point and a single exit point, and a node has one or more paths entering and leaving it. Hence the structure of a program is isomorphic with a directed graph. The number of branching paths through a program provides a measure of its complexity from the point of view of its verification using "white box" testing. From a theoretical aspect, a program may be completely tested if every path has been exercised and every node has been traversed. Although there may also be a requirement that, for complete testing, every node must also be traversed by every pair of (entry, exit) paths which pass through it, this is felt to be superfluous. This allows complexity to be viewed in terms of verification (from testing). To achieve appropriate levels of verification each path is to be traversed, in the course of which each node will be visited.

There have been a number of measures of program complexity proposed, originating with those of Halstead ⁵, McCabe ⁹ and Albrecht ¹. (A useful overview of these, and others, may be found in ¹¹.) The first and third of these are directly concerned with the source code of the program, and therefore may change between implementations, especially where the implementation language differs. The second corresponds more nearly to the path/node description given above. They have common features: earlier investigations of their utility suggest that software science (Halstead's metric) and cyclomatic complexity (McCabe's metric) are correlated with the implementation required to construct

380 Software Quality Management

a given piece of software. The former is concerned with its coding details, and the latter is better applied at the level of design ⁸.

There is a consequent reaction which suggests that software science is therefore likely to be superior when it is to be applied to program source code. This is not necessarily the case, because of the way in which it is applied. Card has stated that "... practitioners can safely ignore software science at present. It probably represents the wrong level of detail for most measurement applications ..." and continues "... many ... experts recommend the use of the cyclomatic representation to ensure adequate ... coverage". He goes on "... cyclomatic complexity does represent the right level of detail; designers and programmers do think in terms of decisions" ³.

The identification of cyclomatic complexity with the path/node description is well supported. As an example, "McCabe's cyclomatic complexity indicates the number of paths through a given piece of code" ⁴. Its computation in an efficient manner, from supplied source code, is an advantage. It may be derived from the implicit graphical representation of the subject program, and corresponds to the number of independent areas defined by the graph. This is described in ¹⁰ and is there shown to be quantified as:

$$v(G) = \text{number of decision statements} + 1$$

This is readily computable for programs from their source code as long as their structures can be analysed. In the case of COBOL programs, this is simply evaluated, without recourse to structural analysis in full detail.

MEANINGFUL RESULTS

To derive the cyclomatic complexity for a given COBOL program, its text must be analysed for the occurrence of *conditional* verbs. In an error-free program, this may be done by scanning the program text for keywords (including verbs), and performing a small number of simple calculations. (It must be emphasised that the program must conform to the ANSI standard and must be without errors; otherwise the computations may be erroneous themselves.)

Stated simply, a scan of the program is undertaken at the lexical or micro-syntactic level, and COBOL words are analysed as they are met. The analyser takes the form of a finite-state sequential machine, with counters for each key verb. No counting takes place until the sequence

PROCEDURE DIVISION

is encountered; until this point is reached, no word can act as a verb. Each verb is then counted as it occurs, as are other keywords which serve to convert *imperative* verbs into *conditional* verbs, as in the example shown in figure 3 above. The cyclomatic complexity of the program is then calculated by summing the number of *conditional* constructions (verbs or other keywords), starting from a 1-origin.

It is not feasible to construct a recogniser of this type using a context-insensitive mechanism, which might be (for example) a text editing package. This approach was investigated and discarded, when it was understood that the *character sequence* could be detected in legally constructed COBOL programs without acting as keywords. One example of such an occurrence is in the text string (*non-numeric literal*) "THIS IS NOT THE REAL PROCEDURE DIVISION". Consequently, it is necessary to identify which of the COBOL elements is actually being processed at any time, to ensure that system-defined words are the subject of the analysis.

The prototype of an analyser of this style was developed using the unix tools "lex" and "yacc". These were quickly discarded, because they led to added complexity of development. The authors' experience in developing system programs in general, and language processors in particular ⁶, suggested that a more straightforward implementation in the 'C' language would be feasible; which proved to be the case.

The development took place within three systems, for implementational ease. The initial approach was under the unix operating system, but was discarded following difficulties in transferring the subject files into the system at that time. The prototype code was transferred to a personal computer system for further development, which then took place as far as the working system for a single program, with testing completed to the satisfaction of the project proposer. This was then transferred to a DEC VAX 8800 system, which contained the COBOL source files to be analysed.

APPLICATION TO EXISTING PROGRAM SUITE

As indicated, this procedure was implemented as part of a project concerned with the measurement of a suite of existing COBOL programs. A system had been established during the preceding ten years for commercial processing within a computer system, which was now in the process of being replaced. An investigation was therefore undertaken to decide on the feasibility of redesigning the programs of the suite, for which purpose their fitness for maintenance was also to be assessed. The text of the programs was therefore supplied to the test site at Staffordshire University.

The source code was transferred into a set of sequential files within a directory available to the DEC VAX 8800 system referred to above. The facilities available to the analysing program were then extended so as to access each of the source files in turn, and to process them. In this way, a complete report on the program suite was prepared automatically: cyclomatic complexity values for each program were collected for the subsequent analysis.

382 Software Quality Management

There was a wide variation in the values of this metric returned, from a minimum of 3 to a maximum of 1377, within a suite of 203 programs. This reflected, in part, the variation in the sizes of the programs under consideration: from 659 lines to 8044 lines. (This latter size metric is very similar to that evaluated by Albrecht referred to above.) Since no account was taken in the respective sizes of the PROCEDURE DIVISION in each case, this may be misleading.

In an attempt to return a figure more relevant to the maintenance complexity expected to be involved for a change to be made to any selected program, a relative complexity measurement was evaluated. This was a function of the cyclomatic complexity and the number of lines in the PROCEDURE DIVISION (referred to as "rcc"), and returned values in the range 0.3 to 17.1. The programs associated with these metrics were inspected, with particular reference to those representing the extreme values. The program size metric was not felt to be representative of the complexity associated with maintenance, and the unconverted cyclomatic complexity metric was only felt to be of limited use, in the case where substantial changes were to be made to the program under consideration. The rcc metric was felt to represent the effort required to make simple updating changes to the programs being maintained most nearly.

IMPLEMENTATION CONSIDERATIONS

A number of facilities were added to the analyser to simplify the processing of the results. These fell into three groups: those for COBOL line processing; those for program identification; and those for special reserved word processing. The first of these classified each line component into one of twelve categories, to simplify further processing. This enabled all comment lines and similar constructions to be discarded without further actions having to be taken. At the same time, all potentially relevant words were marked for further specific consideration.

During the processing of each program to evaluate the metrics described, it was necessary to track the results automatically. For this purpose, the name associated with each was extracted directly during the processing of the source file, from the PROGRAM-ID paragraph. This required no more than a very simple modification to the sequential machine referred to above. A similar modification allowed analysis of all external programs called by the source file presently being analysed, so that an hierarchical program structure within the suite could be constructed (for other purposes). The provision of these further facilities enabled useful information to be extracted for non-complexity purposes with little added effort beyond that already used to construct the original analyser.

CONCLUSIONS

The initial analysis of complexity measurements enabled the selection of a metric which was commercially acceptable for a specific requirement. This was stated in terms which allowed the efficient evaluation of a modification of a metric accepted in many other cases. The source programs under consideration were written in COBOL, to which this metric has not been widely applied. The calculation of this modified metric, referred to as rcc, was efficiently and automatically carried out for a non-trivial suite of programs. Other results of commercial interest were also made available during this process.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the help given by Professor Colin Theaker during the preparation of this paper, and that of Ted Bailey and Dave Brunskill during the inception and the operation of the project from which it came.

REFERENCES

1. Albrecht A.J. and Gaffney Jr. J.E. 'Software function source lines of code, and development effort prediction: a software science validation' *IEEE Transactions on Software Engineering* SE-9(6), pp639 ff (1983)
2. American National Standard ANSI X3.23-1985. ISO 1989-1985
3. Card D.N. with Glass R.L. *Measuring Software Design Quality*, Prentice Hall 1990
4. Grady R.B. and Caswell D.L. *Software metrics: Establishing a Company-wide program*, Prentice Hall 1987
5. Halstead M.H. *Elements of Software Science*, Elsevier 1977
6. Leigh D.J., Naylor J.T., Hopkinson J. and Rowley M.J. *COBOL for OS4000*, GECCL 1981
7. Leigh D.J., Hatter D.J. and Newton R.W. *Software: design, implementation and support*, Paradigm 1987
8. Macro A. and Buxton J. *The Craft of Software Engineering*, Addison-Wesley 1987
9. McCabe T.J. 'A Complexity Measure', *IEEE Transactions on Software Engineering* SE-2(4), pp308 ff (1981)
10. McCabe T.J. and Butler C.W. 'Design Complexity Measurement and Testing', *Comm. A.C.M.* Vol 32, No 12 (1989)
11. Navlakha J.K. 'A Survey of System Complexity Metrics', *Comp. J.* Vol 30, No 5 (1987)