Proceedings of BCS Software Engineering Symposium, Belfast 1976

COBOL

M A Jackson
Michael Jackson Systems Limited
5 Scot Grove
Pinner, Middlesex HA5 4RT, UK

Abstract

COBOL is the most widely used programming language for data processing applications. It possesses certain important basic virtues, which should not be underestimated. It also has important defects:

(a) excessive machine dependence

(b) syntactic anomalies at every level

(c) very poor facilities for defining new primitives

(d) absence of a coherent underlying philosophy for program design.

The unsuitability of COBOL for so-called 'structured coding' is a superficial matter on which, in any case, today's conventional wisdom is faulty.

The COBOL user can do little to mitigate the effects of (a) and (b), but much to overcome (c) and (d). Some software techniques are discussed, and some methodological approaches, especially in relation to (d).

A warning is issued against premature abandonment of COBOL. The fundamentals of data processing systems are briefly explored; the conclusion offered is that we do not yet know enough to design a language for data processing whose virtues would be sufficiently great and sufficiently certain to justify its widespread adoption in place of COBOL.

Introduction

I would like to begin by saying how delighted I was when I received Professor Hoare's invitation to speak at this symposium on Software Engineering. It would be an intellectual *concours d'elegance*, a gathering of massed mental horsepower. And I would be there. And then I saw that I was to speak about COBOL. Imagine my feelings: invited to the ball on condition that I wore my kitchen rags, that I came in my pumpkin!

That night I dreamed a terrible dream. The concours d'elegance was in full swing. Crowds gathered to admire the sleek models on the Algol stand — the famous original Algol 60, which won so many prizes in spite of having no doors for entry and exit, its descendants such as the beautifully balanced Pascal tourer and the versatile overland Simula 67 de villa; even vehicles which shared only the famous name attracted respectful attention. Nor were admirers lacking for the 900 hp PL/I all-purpose limousine, with its incredibly complicated control panel and 500-page driver's manual. But for our exhibits on the COBOL stand the spectators had only derision: the three-wheeled Required COBOL 61, with optional Elective features; COBOL 65, the new model whose announcement passed completely unnoticed; the most recent development — the five-wheeled, eleven-cylindered ANS COBOL; none attracted even a glimmer of admiration — not even from spectators who had owned nothing else all their motoring lives.

Of course I felt better in the morning. One always does. And much of my fear vanished when I looked carefully at the plan for the symposium: the only programming languages mentioned by name were COBOL and Fortran. Clearly the occasion was going to be less of a concours d'elegance and more of an old crocks' race. Certainly I could expect a more sympathetic

audience: perhaps even one of those heartwarming occasions when one gets stuck on a particularly steep DECLARATIVE section and someone comes forward with a spare MOVE CORRESPONDING and vanishes before one has a chance to thank him properly.

So I decided that I would come here today and exhibit the COBOL All-weather Gents' Programming Vehicle for your inspection. Then I would indicate some of the ways in which its users can, and do, get a very high mileage out of it. Finally, I would say something about the possibility of trading it in for a newer, more modern, product, and suggest what I think are the crucial issues we must consider.

The COBOL Language

COBOL's greatest virtue is its simplicity: not the simplicity of a deep and far-ranging abstraction, but the simplicity of a Government-inspired committee which knows what it wants and goes straight for it.

In 1959, most data processing systems consisted of small programs operating on serial files held on magnetic tape; most of these programs were written in machine language or in a very primitive symbolic assembly language, often by inexperienced programmers. Programs were, inevitably, expensive to build and usually very hard to understand. The fundamentals of the COBOL solution were these:

- The language should be as close as possible to natural English.
- Executable program text (the PROCEDURE DIVISION) would be written in sentences and paragraphs, with punctuation by commas, semi-colons and full stops. Operations would be self-explanatory, such as ADD CREDIT TO BALANCE.
- Data declarations should be segregated from procedure in a separate DATA DIVISION. Data should be given a simple hierarchical structure, the elementary data objects being of problem-oriented, not of machine-oriented types.
- Machine-dependent characteristics of a program and of its data files should be isolated in the ENVIRONMENT DIVISION. This would localise the changes necessary to move a program from one machine to another.

Of course, it was all far too simple, but it worked surprisingly well. COBOL was easy to learn and easy to apply to small, well-defined programs. It produced an immediate improvement in program simplicity and readability. It was self-evidently a 'high-level' language. These benefits had to be large, because the early compilers were appalling: users suffered from endless limitations and compiler bugs and, above all, from very high compilation and execution costs. Quite a small program could take three hours to compile and would then run at one tenth of the speed of a well-written machine-language version. But these difficulties could be expected to diminish in time. Other difficulties, due to defects inherent in the language and in manufacturers' views of it, gradually became apparent.

Machine-Dependence

The isolation of machine-dependent characteristics in the Environment Division was so badly thought out that it failed entirely. To define a magnetic-tape file of 80-character card-images, the user would write:

```
ENVIRONMENT DIVISION.
..
SELECT CDFILE ASSIGN TO MAGNETIC-TAPE.
..
DATA DIVISION.
FILE SECTION.
FD CDFILE, DATA RECORDS ARE CARD-IMAGE.
01 CARD-IMAGE PICTURE X(80).
```

in which the SELECT-sentence specifies the machine-dependent information, while the entries in the File Section specify the structure of the data records. However, if further or different

information about the file is to be specified, we find that its allocation to Select sentence or FD entry is at best mysterious and at worst perverted:

- several records to be stored in each tape block (FD)
- standard header labels (FD)
- each tape block prefixed by a length indicator inaccessible to the user program (FD)
- file accessed purely sequentially (SELECT)
- name of record-key item for file accessed by key (SELECT).

In effect the Select sentence and the FD entry partition the information quite arbitrarily and to no purpose. Machine independence and portability are frustrated.

Machine independence was frustrated also in the elementary data item formats. Starting from a sound basic distinction between DISPLAY and COMPUTATIONAL formats, compiler designers were invited to define special formats that could be efficiently represented on particular machines. For example, current IBM compilers recognise COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2, COMPUTATIONAL-3 and COMPUTATIONAL-4, all of which are described, along with their machine representations, in the Language Specification. The user is thus encouraged to know, and to take advantage of, the exact representation of data in the machine. For example, a common device to obtain an unsigned packed decimal item for use in a file record where space is at a premium is to write:

```
02 FIELD-A PICTURE S9(5) COMPUTATIONAL-3.
02 FILLER  REDEFINES FIELD-A.
  03 FIELD-B     PICTURE XX.
  03 FILLER      PICTURE X.
```

in which FIELD-B contains the 4 senior decimal digits its of FIELD-A. By adding 10 to FIELD-A we effectively add 1 to FIELD-B Such techniques are widespread in COBOL installations.

Syntactic Anomalies

COBOL syntax has a number of anomalies. One of the most painful is the absence of terminating symbols for most constructs. In general, a COBOL construct is terminated by the beginning of another construct or the end of the program text. There are therefore some awkward restrictions on the formation of compound constructs. The most famous of these is the difficulty of forming nested IF statements. There being no END-IF terminator, the statement

```
IF V = W
   IF X = Y MOVE R TO S
            MOVE T TO U
   MOVE V TO W
```

does not mean what the indentation of the text seems to imply. The statement MOVE V TO W is subject to the condition X = Y, there being no way to indicate the end of the scope of that condition. A direct result of this anomaly is a curious classification of executable statements into *imperative* and *conditional*, together with a set of irksome restrictions on the use of conditional statements. Another result is a deep distrust of nested IF statements by COBOL programmers, even when they can be written clearly and unambiguously. Many installations have written standards prohibiting the use of nested IF statements altogether.

At a different level, there are anomalies in the COPY statement, used to introduce pre-written COBOL text into a program. Here the anomalies arise from a quite gratuitous specialisation of the statement. Instead of a general definition, allowing arbitrary text to be COPYed, specific cases are enumerated in which the statement may be used. These specific cases, inevitably, contain some subtle traps, cause much difficulty to the conscientious implementor, and turn out to exclude many potentially valuable uses of the COPY statement.

Defining New Primitives

The COPY statement is virtually the only mechanism for defining new primitives — if it can be classed as such. COBOL does not even officially possess a parameterised procedure call

mechanism. However, most manufacturers have now implemented a CALL statement with data items (not files), called by reference, as parameters. CALLed subprograms are separately compiled from the CALLing program.

Design Philosophy

The originators of COBOL, like the originators of most programming languages, gave very little thought to the question 'how should a program be designed?' Perhaps they believed that programs would always be small and would therefore not require any conscious design technique.

Certainly this seems to be the feeling behind the COBOL approach to the question of the scope of data names. All data in a COBOL program is global: there is no way of associating a data object with a piece of procedure text or of preventing any other procedure text from referring to it. In a large program this can create the same kind of nightmare as Fortran COMMON, in which values of variables are mysteriously changed, as if by an unseen penman. But COBOL, unlike Fortran (and unlike Algol 60), does provide the means for defining data structures, which is surely of infinitely greater importance than locality of data reference.

Within the Procedure Division, we need to distinguish the facilities for textual structure from those for control structure. Facilities for textual structure are:

- Statements may (subject to the anomalies concerning imperative and conditional statements) be concatenated into sentences. Some nesting is possible.
- Sentences may be concatenated into paragraphs.
- Paragraphs may be concatenated into Sections.

Facilities for control structure are:

- A paragraph or section may be executed out-of-line by a PERFORM statement.
- A PERFORM statement may specify iterative execution, as in

  ```
  PERFORM PARA-1 UNTIL X = 0.
  ```

  There are other, more elaborate, iterative PERFORM statements, much like the Algol FOR statement or the Fortran or PL/I DO statement.
- A GO TO statement may GO TO the beginning of a paragraph or section.
- A GO TO DEPENDING statement may choose any of several destinations according to the value of an integer variable, as in

  ```
  GO TO P1, P2, P3, P4 DEPENDING ON X.
  ```

This is an unbeautiful, and even slightly ridiculous, set of facilities. But it is perfectly adequate for non-recursive programs. One of today's sadder sights is the Gadarene rush to condemn COBOL because it effectively prevents textual nesting and therefore 'is unstructured'. Largely for the historical reason that the origins of structured programming are in Algol-like languages, structured programming has become closely identified with the use of compound, textually nested, program statements. But, of course, if one considers that stepwise refinement, or top-down design, is the correct approach one could hardly do better than to use the PERFORM statement and avoid textual nesting altogether. In such a program as:

```
CALCULATE-PAY.
    PERFORM CALCULATE-OVERTIME-HOURS.
    COMPUTE GROSS - BASIC + (OVERTIME-HOURS * OVERTIME-RATE).
    PERFORM CALCULATE-DEDUCTIONS.
    COMPUTE NET - GROSS - DEDUCTIONS.
CALCULATE-OVERTIME-HOURS.
    SUBTRACT 40 FROM HOURS-WORKED GIVING OVERTIME-HOURS.
```

the refinement steps, crude though they are, are very clear. Further, everything has a name. Only bigotry could lead one to prefer Algol or PL/I in this respect.

USING COBOL

Many COBOL programmers are happy to use COBOL as it is — indeed, many use only a subset, avoiding the PERFORM statement or some of its variants (too hard to understand), nested IF statements (too easy to get wrong), compound conditional expressions (obscure and often wrongly compiled unless fully parenthesised), and even subscripted variables (too sophisticated). Used in such a way, the language has very limited power and facilities. It is a clear indication of the simplicity of much COBOL programming that many programmers can complete several years of work under such restrictions. For more demanding Applications, or more critical users, there are various ways in which the language can be extended and made into a better tool for system development.

Use of CALL Statement

One obvious approach is to provide a procedure library in the form of a set of subprograms which can be CALLed by the user program. This is the approach taken by some database systems, such as IBM's IMS. A major advantage of this technique is that compatibility is easily achieved and enforced among a variety of problem-oriented languages: IMS itself is written in Assembler Language, and interfaces are provided for user programs written in Assembler and PL/I as well as for COBOL.

Language Extensions

A different approach is to extend the COBOL language by the addition of new syntax for each new facility. This approach has been used extensively in successive versions of COBOL. For example, the ANS COBOL table-handling feature provides a SEARCH verb for serial and binary table look-up:

```
SEARCH TABLE-ENTRY VARYING INDEX-1
  AT END DISPLAY 'NOT FOUND'
  WHEN ENTRY-KEY (INDEX.-1) = ARGUMENT
    DISPLAY 'FOUND', ENTRY-VALUE (INDEX-1).
```

specifies serial search from the current value of INDEX-1, while

```
SEARCH ALL TABLE-ENTRY
  AT END DISPLAY 'NOT FOUND'
  WHEN ENTRY-KEY (INDEX-1) = ARGUMENT
    DISPLAY 'FOUND', ENTRY-VALUE (INDEX-1).
```

specifies a binary search over the whole table. An IBM extension for limited string handling provides concatenation and deconcatenation of strings, as in:

```
STRING FIELD-1 DELIMITED BY '*'
       FIELD-2 DELIMITED BY SIZE
       INTO FIELD-3 WITH POINTER SUBSCRIPT-2
       ON OVERFLOW PERFORM EXCESS-LENGTH-PROCEDURE.
```

The Codasyl Data Base Task Group proposals, in contrast to IMS, include a Data Manipulation Language to be embedded in COBOL as 'host' language: the user would write such statements as:

```
   FIND NEXT REC-A RECORD OF AREA-1 AREA
       SUPPRESS ALL CURRENCY UPDATES.
```

In general, this approach to the provision of new facilities seems full of dangers. The language is already large: the IBM Version 4 compiler already has well over 400 reserved words. Each new facility demands a substantial increase in the size and complexity of the compiler, and hence a substantial commitment on the part of the implementor. There is therefore an ever diminishing chance of compatibility between the facilities provided by one compiler and those provided by another. Worse, the facilities already implemented and proposed seem to achieve little cooperation, let alone synergy: two and two cost five, but make only three.

Preprocessors

A third approach is to provide new facilities in the form of a preprocessor. The user writes non-COBOL syntax, possibly embedded in a COBOL program; the preprocessor converts the whole into some kind of standard COBOL, which is then compiled by an ordinary COBOL compiler. This approach has been used to implement Decision Tables, to provide 'Structured Programming' constructs (such as DO WHILE, CASE, etc), and to generate programs according to simple predetermined patterns (such as report writing, input editing, file updating, etc).

Use of a preprocessor, of course, is an admission of defeat on somebody's part: it cannot be efficient to generate COBOL as an intermediate step in generating an object program. But it has a commercial attraction which is revealing. If you use a preprocessor, you may be able to obtain the benefits of an improved and more powerful language for writing your programs without formally committing yourself to anything beyond the standard COBOL which the preprocessor generates. The user can reasonably tell himself that his programs can be maintained by any COBOL programmer and compiled by any compiler. It may not be true, but it sounds convincing.

Methodologies

For many years now attention has been paid in data processing installations to achieving a standardised and disciplined way of writing COBOL. Undesirable elements of the language, such as the ALTER verb, have been proscribed; rules have been laid down for the construction of self-documenting data names and procedure names; limitations have been placed on the more dangerous uses of the PERFORM statement.

These efforts have received added impetus from the fashion for 'Structured Programming'. It is now generally accepted that programs should be clear and readable, using the limited set of control structures in a text which is neatly indented on the page. Program development should proceed 'top-down' or by 'stepwise refinement', possibly with an intermediate phase in which the program is written in an abstract programming language or pseudo-code.

The question arises: is not COBOL so primitive that we must abandon it if we are to obtain the benefits of Structured Programming? Is it not now clear that PL/I, or Algol 60, or Pascal is far better suited to structured programming, and must therefore be adopted without delay?

The question is horribly reminiscent of the Fortran/Algol question. The lack of structured programming control structures in Fortran is a trivial matter: they are easily simulated, if necessary by disciplined hand coding. What is far more serious is the lack of data structures, a lack which is, of course, shared by Algol. The question is simply off the point: without data structures both languages are very bad.

Data Processing Systems

We can, no doubt, agree that COBOL is a very bad language in many respects. But it is very widely used, and to change to another language would therefore be very expensive: we ought not to contemplate making such a change unless we are reasonably confident that the benefits would be very large and that another change would not be necessary for a long time. Such confidence would have to be based on a clear understanding of the nature of data processing systems and on a clear methodology for building them. The chosen language would then be seen to express that nature and support that methodology.

But such an understanding and methodology are lacking. They are certainly not provided by structured programming in the narrow sense in which the term is normally used. Structured programming, in that narrow sense, provides a way of writing programs whose execution is more easily deducible from their text; but it says nothing, about how those texts should be related to the problems they solve.

One possible view of data processing systems is the following. It is mentioned here not as an answer to our lack of understanding but as an illustration of the impact that such an understanding could have on a programming language.

A data processing system models, or simulates, a part of the real world. In that world there are numerous independent entities, interacting one with another These entities may be grouped into equivalence classes by suitable abstractions: each equivalence class is modelled by a process within the system , and each entity of that class by an activation of that process. In the real world, the entities interact by the output of one providing stimulus to another; within. the system, the results of one process instance become transaction input to another process instance. The activation records, or state-vectors, of the processes are the 'master records' in the system.

If we were to adopt the above view of data processing systems, we would need a programming language in which long-term processes could be conveniently created, activated, suspended and destroyed, in which the state-vectors of such processes could be easily defined and manipulated. Such needs are not met by any of the programming languages in common use today. It is awkward and inconvenient to use COBOL to implement a system from such a viewpoint, but it would be no less awkward and inconvenient to use PL/I or Pascal.

An Ideal

One might suppose that we should therefore introduce into our programming language new constructs designed to implement long-term processes.

I think that this would be a most serious error The main reason for our spectacular lack of progress in data processing during the past fifteen years has been precisely the perpetration of such errors. Our unfortunate, though justifiable, concentration on efficiency has led us to stress the implementation rather than the concept, the specific rather than the general, the machine rather than the problem.

We should instead be thinking about how to specify a process without prejudicing its implementation. An interrupt handler for a hardware device, a conversational program, a long-term process of the kind suggested above — all these are processes; but their common nature is hopelessly obscured by the widely differing details of their implementations. These details include some apparently high-level considerations: is the object we are concerned with a main program, a subroutine, an interrupt-driven program, a coroutine, a task? And they include some obviously low-level considerations: what is the syntax of a 'read' statement? how do you set and restore the program status? what is the standard format of the parameter list? We should be concerned to specify individual processes without paying attention to these considerations.

It seems sufficiently general for data processing applications to regard a process as a main program which writes and reads serial files. The input files, collated by time, provide the transaction input; the serial output files are the process's contribution to the transaction input of other processes. Communication between processes is therefore achieved by serial 'write' and 'read' operations. Synchronisation is maintained by the physical ordering of the records within one file and by the collating of the transaction input to each process.

A programming language to support such a view could be very simple because of the small number and the generality of the concepts involved. Indeed, we already have a precompiler which handles processes communicating by serial files: it is capable of generating those processes in various implementations of the kinds mentioned above. For example, if two processes A and B are specified, communicating by a serial file which A writes and B reads, the precompiler is capable of producing two main programs A and B with the serial file on magnetic tape, or a main program A invoking B as a subroutine, or a main. program B invoking A as a subroutine.

Clearly, a programming language at the highest level of generality envisaged here is not a complete system development tool. Having specified the individual processes in such a language we must further specify the implementation. The most important implementation questions are these:

(a) What is the algorithm for determining which process (or processes) should be activated next?

(b) Given the answer to question (a), how should the processes be implemented? In particular, how should their state vectors (activation records) be handled?

(c) Given the answers to questions (a) and (b), is it desirable to dismember processes and/or state vectors, recombining them into new hybrid forms? If so, how?

In answering question (a), we may, for example, choose between a batch and an on-line solution. In answering question (b), we may choose between serial and direct-access files, and may consider such implementations as index-sequential, hierarchical and chained organisations. In answering question (c) we may form transaction-processing modules by taking procedure coding from several processes, or may form database segments by taking parts from several state vectors.

Ideally, these implementation choices are independent of the specification of the individual processes. Furthermore, while it is almost certainly impossible to make the choices automatically, it seems possible to carry them out automatically. The program transformations discussed by Burstall and Darlington (Some Transformations for Developing Recursive Programs, in Proceedings of 1975 Conference on Reliable Software) may be of exactly the kind needed for dismembering and recombining processes.

We may therefore imagine system development to consist of two phases. In the first phase, individual processes are identified and programmed in a suitable languages this corresponds to what we may today call system specification. In the second phase, implementation choices are made and carried out; this corresponds to most of what the system designer does today, and to the optimisation aspects of programming.

The role of COBOL, if it has a role in this scheme of things, clearly lies in the first phase. Equally clearly, COBOL is a very unsuitable language for the first phase. All one can usefully say is that the other contenders are no better except in minor particulars. The crucial defect which they all share is their hopeless confusion of the problem with the machine. It cannot be a step forward to devise or adopt yet another language which has this same defect.