Reflections on the evolution of COBOL

Alex Sellink [*], Chris Verhoef [†]

University of Amsterdam, Programming Research Group
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands

alex@wins.uva.nl, x@wins.uva.nl

Newbie language designer error #17:
Overgeneralizing from experience with natural languages.
English makes a terrible programming language.

Abstract

We are interested in the evolution of COBOL software throughout the years. We observe that the original design decisions of COBOL caused a style of programming that is nowadays undesirable. We give an example of a year 2000 solution that causes problems in the year 2000 due to the original design of COBOL. We briefly mention that solutions are available in the scientific literature.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring;

Additional Key Words and Phrases: Reengineering, System renovation, Interface reengineering, Control flow normalization, Language migration, COBOL.

1 Introduction

Today's Internet originates from a project initiated by the American Department of Defence. Years ago, this innovative Department was also involved in the initial phase of the development of COBOL. Various computer suppliers and users created in the late fifties the *Conference on Data System Languages*, abbreviated CODASYL. The objectives of CODASYL were to create a non-proprietary language that could serve as a universal communication means that would solve the problems of device dependent languages with all their different dialects. The CODASYL conference is one of the early — if not the earliest — examples of an attempt to prevent incompatibility problems in software by

[*] Alex Sellink was sponsored in part by bank ABN Amro, software house DPFinance, and the Dutch Ministry of Economical Affairs via the Senter Project #ITU95017 "SOS Resolver".

[†] Chris Verhoef was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO), project *Interactive tools for program understanding*, 612-33-002.

means of standardization. The goal of CODASYL was to develop a programming language with the following properties:

- based on English Language (so that it could be easily read);
- geared towards administrative programming;
- maintainable;
- usable on every hardware platform.

After a year the first proposal for a programming language referred to as COBOL-60 emerged. Although this language had many of the above properties incorporated there was still a problem with device independency. After the emerge of compiler technology, these problems were solved. During the sixties several development extensions were proposed to COBOL and in 1968 the *American National Standards Institute* (ANSI) published a COBOL standard. In 1970 this standard was approved by the *International Organization for Standardization* (ISO).

Since then, the ongoing development of COBOL lead to new COBOL standards in 1974 and 1985. Recently, an object oriented version of COBOL is developed. Whether or not al these standards meet the requirements as formulated by CODASYL is a question that we try to answer in the next section.

2 The Design Objectives in Retrospect

A COBOL program consists of four divisions. We can compare them to chapters in natural language. Each division consists of sections, the sections contain paragraphs, the paragraphs in turn contain sentences, and the sentences contain statements. These concepts are prominent consequences of the aim to resemble a natural language. Since COBOL sentences — as the following code example shows — indeed show a close correspondence to natural language we can conclude that the designers of COBOL succeeded in fulfilling the first design property.

```
IF X EQUALS 3 OR 4
   ADD 1 TO X
   MULTIPLY Y BY 2
   MOVE SPACES TO S.
```

The massive usage of COBOL in the administrative sector proves that the second design property "geared towards administrative programming" is also fulfilled.

As for the maintainability of COBOL we can conclude that this objective is not accomplished. Numerous publications on this subject and the myriad of so-called legacy systems are our witness.

Although COBOL is available on virtually every hardware platform we do not think that the fourth objective is reached. The initial idea was that all these different platforms should make use of exactly the same language. Currently, however, there are many different dialects in use. Each with their own special features, extensions etc.

Let us mention some dialects:

- ANSI-74 COBOL
- ANSI-85 COBOL
- IBM OS/VS COBOL
- IBM COBOL/II
- IBM COBOL SAA
- Unix COBOL X/Open
- MicroFocus COBOL
- Microsoft COBOL
- Ryan McFarland RM/COBOL
- Ryan McFarland RM/COBOL-85
- DOSVS COBOL
- UNIVAC COBOL
- Fujistu COBOL

3 Reading versus Parsing

In the previous section we argued that CODASYL succeeded in fulfilling two of the four design requirements, thus answering the question we formulated in the end of the introduction. Another, even more interesting, question that naturally arises is whether or not the original design requirements are still considered valuable after 30 years of experience in software development. For three of the four design requirements we can give a positive answer to this question. Only the first requirement "based on English language" is not considered a desirable design requirement nowadays. Of course, a syntax that is easy to read is preferred above one that is not. However, the assumption that this property can be met by means of having a close resemblance to natural language is not considered correct anymore. The point is that there is an essential difference between conventional text (as can be found in books, papers, etc.) and source text that serves as input for a computer program. The logical structure of the latter one is far more complex. We argue that this should have consequences for the syntax one chooses: in general we can distinguish two different activities both having plain text as input. These two activities are *reading* and *parsing*. When the syntax for some language is developed one should have in mind whether text in this new language should be easy to read, easy to parse, or both. In the early days of programming the answer to this question roughly sounded as follows:

> *Humans read text and computers parse text. Since life should be made easy for humans — and not necessarily for computers — reading is more important than parsing.*

Nowadays we think that the situation is not that simple. To explain this let us first take a look at a COBOL sentence that is slightly more complicated than the previous one, and review its readability. The following example is taken from [Volmac82].

```
IF condition-1
    statement-1-1
    IF condition-2
        IF condition-3
            statement-3-1
           ELSE statement-3-2
       ELSE statement-2-2.
```

In our opinion the above COBOL sentence is hard to read. The Volmac Groep has the same opinion. We quote from [Volmac82]:

> Because their logic is difficult to follow, nested `IF` statements should wherever possible be avoided in a COBOL program.

Thus, we can safely conclude that sentences with a complex logical structure are difficult to read — even if the syntax is close to natural language. A second important observation is that whenever a human is forced to read a sentence with a logical structure that is too complex to comprehend he or she will try to parse that sentence. Thus breaking the information into smaller pieces that are easier to understand. Combining these two observations we find that there might be at least one argument why computer programs (that usually have a complex logical structure) should not only be easy to read but also be easy to parse. The alternative is to try to avoid complex logical structures (like nested `IF`-statements) in computer programs. In [Volmac82] this alternative is advocated. We quote:

> Often a series of simple `IF` statements can be used in place of the nested `IF` statement.

We display the alternative they propose for the nested `IF` sentences given on page 4. The label `R1014` is a fresh label of a newly created empty paragraph:

```
    IF condition-1
           statement-1-1
      ELSE GO TO R1041.
    IF condition-2
           statement-2-2
           GO TO R1041.
    IF condition-3
           statement-3-1
      ELSE statement-3-2.
R1014.
```

Obviously, the price that is paid now is the introduction of two unstructured jump instructions. Apparently, this was not considered as undesirable as it is considered today. Presently, most companies oblige their programmers to program without using unstructured jump instructions, in order to lower the costs of software maintenance. If we don't want to use the alternative for the nested `IF` sentence on page 4, as proposed by [Volmac82], we have to accept that source text for computers is logically too complex to comprehend and must be parsed. Thus, we must see to it that parsing becomes as easy as possible. Obviously, the complexity of parsing is immediately related to the complexity of the grammar in question. Thus, syntax for programming languages must have grammar rules that are as simple as possible. This can be reached by adding all kinds of markers to reveal the logical structure. Scope terminator symbols are examples of such markers. Obviously, in a language that should have a close resemblance to natural language (old COBOL dialects) such markers are not present. The dramatic consequences for the complexity of the grammar is well illustrated by the following quotation[1] from [ANSI 85, loc. cit. p. IV-40] explaining the scoping rules of statements:

> When statements are nested within other statements which allow optional conditional phrases, any optional conditional phrase encountered is considered to be the next phrase of the nearest preceding unterminated statement with which that phrase is permitted to be associated according to the general format and the syntax rules for that statement, but with which no such phrase has already been associated.

About 20 years ago, one started to realise that such complex semantical properties should be eliminated from the grammar, at the cost of a syntax that is less close to natural language. In 1985, when the new COBOL standard was published [ANSI 85], one of the important changes was that the initial design decision to have a "natural language syntax" as much as possible was departed. Amongst the 21 major changes incorporated in the COBOL specifications within the CODASYL Journal of Development 1978 was the inclusion of 19 explicit scope terminators. We quote from [ANSI 85, loc. cit. p. XVIII-7]:

> The inclusion of additional facilities to support structured programming, including implicit and explicit terminators to delimit the scope of statements and the `CONTINUE` statement.

Due to downwards compatibility issues, the old paradigm could not be completely departed. Implicitly closed `IF`-constructions are still valid syntax in COBOL 85. Thus, both the code example given on page 4 and the following code fragment are correct COBOL 85-syntax:

```
IF condition-1
```

[1]This quotation also serves as a excellent example illustrating that even pure english natural language must be parsed in order to detect its meaning, if the sentence in question has a logical complexity that is sufficiently high.

```
        statement-1-1
        IF condition-2
            IF condition-3
                statement-3-1
            ELSE
                statement-3-2
            END-IF
        ELSE
            statement-2-2
        END-IF
    END-IF.
```

The presence of explicit scope terminators `END-IF` not only improves the readability of this sentence, but it also — as these things are related — simplifies the structure of the underlying grammar. Apart from the "readability issue", simple grammars have another important advantage. This latter advantage is illuminated in the next section.

4 Code Restructuring

Although new COBOL dialects are as much as possible upwards compatible[2] with respect to older dialects, the owners of COBOL software usually want to adapt their code to new standards as soon as these new standards have become sufficiently stable. In other words: the adaptations carried out by the compatibility tools are not satisfactory in some sense. This is due to the fact that compatibility tools solve the incompatibility problems with the least possible effort. Typically, a tool for upgrading COBOL 74 to COBOL 85 does not add scope terminators or reformulate patterns where an `EVALUATE`-statement (a case construction, not present in COBOL 74) could be used. This is hardly surprising. In order to introduce these new concepts in old source code one has to trace patterns that give rise to usage of new features. Location of such patterns is far more complicated than the kind of patterns that are currently traced by the compatibility tools, and require deep knowledge of the underlying grammar. The currently existing compatibility tools do not have this deep grammar knowledge built in. Obviously, the development of tools that perform non-trivial transformations on source code is easier on languages that have a simple grammar. Thus, another important advantage of simple grammar rules is the positive effect that it has on the complexity of the supporting tool environment. The impact of this should not be underestimated. There is a strong need for tools that perform the kind of non-trivial transformations we mentioned. From reliable sources we understood that a large Dutch financial company deployed a programmer (the poor chap) for one year to add explicit scope terminators into a large COBOL application by hand. In [BSV97b, BSV97a]

[2]Moreover, tools that (almost) automatically convert your code to the new standard are supplied with the release of new compilers. An example of a collection of such tools is CCCA facility of IBM.

it is explained how this task could have been performed in a few hours rather than in a year.

We give an example of a transformation which is directly related to the evolution of the COBOL language. Consider the following code fragment taken from [LL89, loc. cit. p. 129]:

```
IF BALANCE > ZERO
    IF BALANCE > 100
       COMPUTE FINANCE-CHARTS = BALANCE * .015
       ADD FINANCE-CHARTS TO BALANCE
    ELSE
       MOVE 1.50 TO FINANCE-CHARTS
       ADD FINANCE-CHARTS TO BALANCE
ELSE
    MOVE ZERO TO FINANCE-CHARTS.
```

Note that `ADD FINANCE-CHARTS TO BALANCE` occurs in both the `THEN` and `ELSE` branch. The reason for this is that a conditional construction is forced to be the last construction in blocks like sentences or — as in the example case — the statements in the `THEN`-part of another conditional construction. Tis is due to the implicit scope termination rules. Natural would be to remove the `ADD` statement from both the `THEN` and the `ELSE` branch and place it outside the scope of the innermost `IF`. However, since no explicit scope terminators are present in COBOL 85 conditional constructions must be ended by an `ELSE` matching a heigher `IF` or a seperator period. Both options are unsatisfactory in the case of the innermost `IF`. If we choose the first option — as done in the example — we have to copy the `ADD` statement because it does not make sense to place it anywhere after termination of the innermost conditional construction, and if we chosse the second option we have to deal with the problem that the period also terminates the outermost `IF` construction, whereas `ADD FINANCE-CHARTS TO BALANCE` should be inside the scope of this outermost `IF` statement.

Using the COBOL 85 standard we can transform this into its natural form:

```
IF BALANCE > ZERO
    IF BALANCE > 100
       COMPUTE FINANCE-CHARTS = BALANCE * .015
    ELSE
       MOVE 1.50 TO FINANCE-CHARTS
    END-IF
    ADD FINANCE-CHARTS TO BALANCE
ELSE
    MOVE ZERO TO FINANCE-CHARTS
END-IF.
```

It is possible to automatically detect the old patterns and to replace them for the more natural pattern that avoids code duplication. See [BSV97b, BSV97a] for details.

Not all transformations on source code are related to the evolution of compilers. Also a design decision (2 digits for year variables), changing business requirements, or even political factors (the Euro) may give rise to transformation on source code. We end this article with an example of a so-called Year 2000 conversion. This example is particularly interesting to us because it is not just an example of a transformation on COBOL code, but also an illustration of "implicit scope termination grief". Due to the possibility of implicit scope termination, the incorrect replacement pattern does not lead to compiler errors and thus not to syntactically incorrect code. The example is communicated to us by Dr. Joseph Kisting. He is manager of the Competence Center Kalenderjahr 2000 of Debis Systemhaus. Debis is owned by Daimler-Benz InterServices and Cap Gemini Sogeti. We kindly acknowledge permission of Debis Systemhaus to publish the code fragment in question:

```
    IF LIEFER-JJ < 50
        MOVE 20 TO LIEFER-JH
    ELSE
        MOVE 19 TO LIEFER-JH

    MOVE BETRAG TO BUCHUNGSBETRAG
    MOVE  MENGE TO BUCHUNGSMENGE
    WRITE BUCHUNGSDATEI
        FROM BUCHUNGSSATZ.
B99.
    ...
```

The first four lines were produced by a Year 2000 solution tool. The last four lines are original code. A standard solution used by Y2K tools[3] is used here: for values for `LIEFER-JJ` smaller than `50` it puts in the variable `LIEFER-JH` the value `20` and for other years the value it assigns is `19` so that the complete calculation will be a Year 2000 compliant. This is in order, and the code *is* Year 2000 compliant. The code compiles and gives the good output before the Year 2000. However, as soon as `LIEFER-JJ` is smaller than `50` the original code is no longer evaluated since the separator period on the last line delimits the scope of the `IF`. The problem here is that the conditional statement that was produced by the Y2K tool did not include a scope terminator. Due to the separator period at the last line the original code became conditional. Of course, this is a bad Y2K tool. Still, the implicit scope terminator rules do not reveal the mistake, for instance with a compile error, since the separator period that first just ended a sentence now also serves as an implicit scope terminator. Here is a replacement pattern that would not cause a mistake in the above example:

```
IF LIEFER-JJ < 50
    MOVE 20 TO LIEFER-JH
ELSE
    MOVE 19 TO LIEFER-JH.
```

[3]Y2K tool stands for year 2 kilo tool, i.e., year 2000 tool.

Next we will show that using a separator period in the replacement pattern will not solve the problem in all cases. To illustrate this we will show that using the separator period as a scope terminator does not give correct results in all cases. Consider an original conditional code fragment below. The task of the Y2K tool is to change the 3rd line into a conditional one in order to take care of the year 2000. Let us suppose that the original code fragment of Debis Systemhaus was:

```
     IF BETRAG > 0
    MOVE BETRAG TO BUCHUNGSBETRAG
    MOVE  MENGE TO BUCHUNGSMENGE
    WRITE BUCHUNGSDATEI
        FROM BUCHUNGSSATZ.
B99.
    ...
```

We add the replacement pattern that we proposed above in the code fragment:

```
     IF BETRAG > 0
IF LIEFER-JJ < 50
MOVE 20 TO LIEFER-JH
ELSE
MOVE 19 TO LIEFER-JH.
    MOVE BETRAG TO BUCHUNGSBETRAG
    MOVE  MENGE TO BUCHUNGSMENGE
    WRITE BUCHUNGSDATEI
        FROM BUCHUNGSSATZ.
B99.
    ...
```

which places the original code outside the scope of the outermost `IF`. In fact, this implies that a solution to solve the Y2K problem using `IF` constructs relies on the use of explicit scope terminators. For, if the replacement pattern would have been

```
IF LIEFER-JJ < 50
    MOVE 20 TO LIEFER-JH
ELSE
    MOVE 19 TO LIEFER-JH
END-IF
```

the Y2K tool would have given a correct output in all cases. So we can conclude that an mportant part of Y2K solutions in COBOL is to take care of adding explicit scope terminators in order to prevent mistakes of automated conversions and in order to detect erroneous Y2K conversions. Dealing with this problem is nontrivial as pointed out earlier in this paper. But it can be done.

5 Conclusions

Concluding we can say that the initial design of COBOL that it should resemble a natural language as close as possible has lead to significant problems in COBOL software. The use of sophisticated restructuring technology is needed in order to improve the maintainability of such code. The Y2K problem puts restructuring of COBOL software in an actual context and accentuates the necessity to structure the COBOL systems.

References

[ANSI 85] American National Standards Institute, Inc. *Programming Language – COBOL*, ANSI X3.23–1985 edition, 1985.

[BSV97a] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *proceedings of the fourth working conference on reverse engineering*, pages 144–153, 1997.

[BSV97b] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In M.P.A. Sellink, editor, *Proceedings of the 2nd international workshop on the theory and practice of algebraic specifications*, 1997. To appear.

[LL89] J. Longhurst and A. Longhurst. *COBOL*. Prentice-Hall, 1989.

[Volmac82] Volmac Groep. *Volmac Standard COBOL*, second edition, 1982.