University of Amsterdam
Programming Research Group

```
DATA DIVISION.

  WORKING-STORAGE SECTION.

    01.  DSJDFRR-IJ192390-SDJGEHJ2.
         02.  UI8T-G5KM-N034D                PIC 99/99/99.
         02.  HJML97G-KNBVFD45-7-HG          PIC XXXXXX.
         02.  LK21-34-JSJKDJR-93894J         PIC :-).

    01.  DVK54-FJKFJK4390--23-OKR2349IO      PIC 9999.

    01.  HDRISO0934834R3NMJRQRRUJJJ49        PIC 9(2).
```

Reengineering COBOL software implies specification of the underlying dialects

Mark van den Brand
Alex Sellink
Chris Verhoef

Report P9702 Februari 1997

University of Amsterdam

Department of Computer Science

Programming Research Group

Reengineering COBOL software implies specification of the underlying dialects

Mark van den Brand

Alex Sellink

Chris Verhoef

Report P9702 Februari 1997

M.G.J. van den Brand

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7593
e-mail: markvdb@fwi.uva.nl

M.P.A. Sellink

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7583
e-mail: alex@fwi.uva.nl

C. Verhoef

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7581
e-mail: x@fwi.uva.nl

Universiteit van Amsterdam, 1997

Reengineering COBOL software implies specification of the underlying dialects

Mark van den Brand, Alex Sellink, Chris Verhoef *

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

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

Abstract

We argue that maintenance and reengineering tools need to have a thorough knowledge of the language that the code is written in. More specifically, for the family of COBOL languages we present a general method to define COBOL dialects that are based on the actual code that has to be reengineered or maintained. Subsequently, we give some typical examples of maintenance and reengineering tools that have been specified on top of such a COBOL grammar in order to show that our approach is useful and leads to accurate and relatively simple maintenance and reengineering tools.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring; D.2.m [**Software Engineering**]: Miscellaneous—Rapid prototyping

Additional Key Words and Phrases: Reengineering, System renovation, COBOL

1 Introduction

There is a constant need for updating and renovating business-critical software systems for many and diverse reasons: business requirements change, technological infrastructure is modernized, the government changes laws, or the third millennium approaches, to mention a few. So, in the area of software engineering the subject of reengineering becomes more and more important. Reengineering is the analysis and renovation of software, see, for instance, [CC90] for an introductory paper and [BKV97] for an annotated bibliography on this subject. To aid in the analysis and renovation of software it is crucial to have tool support. We strongly believe that such tools largely benefit from having knowledge of the language they have to analyze or renovate. In other words, tools to aid in reengineering should be based on the grammar of the language they intend to process.

*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.

As most of the business-critical software is written in COBOL, we will present a general method describing how to obtain a COBOL grammar, given a number of COBOL programs. Moreover, we explain how to reduce the size of this grammar significantly by means of, as we call it, *unification of production rules*. Finally, we show that tools that are based on such a compact language definition are useful in reengineering the code written in that language. Thus, the challenge of specifying reengineering tools lies in the (compact) specification of the underlying language. When it comes to defining syntax is is obviously better to use a parser generator, like Lex+Yacc than to write a parser by hand. Therefore, it is not surprising that, e.g., Software Refinery [Rea92], which is based on generic programming language technology incorporates a number of reengineering tools (for more connections between reengineering and generic language technology we refer to [BKV96]).

To emphasize the importance of tools based on language definitions we give a small example. Suppose that we need a tool that analyzes COBOL programs by looking for date-related variables and that this tool is solely based on lexical scanning. This implies that the tool will list comments containing the patterns it is looking for, as well. To reduce the number of so-called false-positives it is sensible to modify this tool so that listing of comments will be suppressed. In fact, the tool now contains knowledge of the language it is analyzing. Still many false-positives will be found for various reasons. This can mostly be solved in an ad-hoc manner by adding extra knowledge of the language. This procedure has to be reiterated for a tool that needs to perform another task. In fact, most of the time a tool developer is busy with adding knowledge of the language to the tool instead of implementing the actual reengineering or maintenance task of the tool. So in our opinion it is a better idea to approach this in a more structured manner, by specifying COBOL syntax and subsequently specifying tools on top of that syntax. Which brings us to the matter of defining COBOL syntax. There is a myriad of COBOL dialects. We will discuss in this paper a general method describing how to obtain a practical COBOL grammar that recognizes a particular dialect. Then we discuss how tools can be specified that aid in reengineering and maintenance tasks. At this point the reader may observe that although putting more knowledge in a tool implies less false-positives, the processing overhead will increase. However, since legacy systems are notoriously large, the reduction of false-positives is a serious matter. In the end it pays off to use a tool that has extensive knowledge of the language. Apart from that it can even be the case that a tool that has knowledge of the language improves its performance. For instance, a tool that returns the `PROGRAM-ID` of a given COBOL program only needs to process the `IDENTIFICATION DIVISION` whereas a tool that has no knowledge of the language has to process the entire program to find the `PROGRAM-ID` (if it finds it at all, see Section 5.1 for more details).

Having motivated our work, we will give an impression of it. In Figure 1 we depicted how to obtain a practical grammar that is based on preprocessed code. We briefly discuss the various phases of that process.

First, we will discuss the left-hand side of Figure 1. The starting point is COBOL code that needs to be analyzed. In our case this was a mix of two COBOL dialects provided by a (financial) company. To be able to analyze and change such code, it is important to minimize effort. This implies that the code has to be preprocessed. We sketch this approach: first we strip the COBOL code

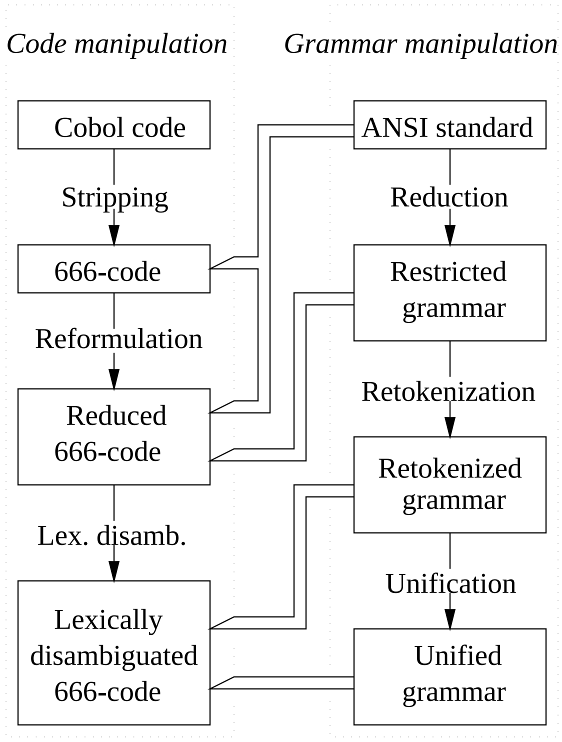

Figure 1: Code and grammar manipulations.

so that only relevant columns remain, thus obtaining so-called 666-code[1]. Then we reformulate this code a bit by performing some global substitutions. For instance, we substitute `THRU` for `THROUGH`. Then we lexically disambiguate the thus obtained reduced 666-code, for instance, to be able to lexically distinguish the comment marker * from the multiplication operator *. The result is lexically disambiguated 666-code.

The right-hand side of Figure 1 deals with manipulations on the grammar. The open arrows indicate that the grammar in question understands the code the arrow points at. We have an ANSI COBOL grammar at our disposal [Man96], it understands 666-code and reduced 666-code. We reduce the ANSI-grammar by the removal of redundant tokens and we remove all the constructs that do not even occur in the code, for instance, the END PROGAM HEADER did not occur in the code. Thus we obtain a restricted grammar that understands reduced 666-code. Then we retokenize the grammar so that it will recognize the lexically disambiguated 666-code. Finally, we unify the grammar. Roughly, this means that we combine context-free rules into less rules. The advantage of this approach is that the grammar will be reduced in size, which leads to a better performance of the generated parser. Of course, the unified grammar will recognize more programs than the retokenized one, it may even recognize incorrect programs. But since the original code is compilable, this causes no

[1] We call it 666-code since the `perl(1)` expression `substr($_,6,66)` is responsible for stripping the original code.

problem (see Section 4.4 for more details).

In practice, the above code and grammar manipulations will be carried out in parallel. In order to explain our approach we treat those transformations as conceptual entities.

1.1 Organization of the Paper

In Section 2 we discuss how to obtain COBOL definitions in general. In Section 3 we will elaborately discuss the code manipulation part of Figure 1 where we translate COBOL code into lexically disambiguated 666-code that suits our reengineering purposes. The grammar manipulation part of Figure 1 will be discussed in Section 4. In this section we discuss a general method to define the syntax of COBOL. In section 5, we discuss the specification of tools based on the COBOL syntax definition. In Section 6 we give our conclusions.

1.2 Related Work

In [BFKO93] and [BFK$^+$94] a formal semantics of a subset of COBOL is presented based on denotational semantics. A reverse engineering tool is based on these ideas which can be used to reverse engineer COBOL74 programs. This tool is based on a COBOL grammar consisting of 170 complex rules in BNF notation, based on this definition the formal semantics of the control statements is defined. These control statements are isolated in a new language *MiCo*. We give a general method describing how to obtain such grammars, we have specified such grammars, and we specified tools based on such grammars.

In [NM93] experiences with building a remodularization tool for COBOL using Software Refinery is discussed. We used a system that is similar to theirs to develop tools for COBOL.

In [CCL96] a method is proposed to algebraically specify tools to aid in reengineering and maintenance. Their approach is to base the tools on a database that contains knowledge of the code to be maintained. We stored the knowledge of the code in a language definition of that code. They use traditional parser generation techniques (Lex+Yacc) to generate parsers for specific languages before the actual analysis can take place. In our case we use more sophisticated incremental parser generation techniques. See Section 2 for more comparisons and details.

2 Towards Defining COBOL

In this section we will discuss how to define syntax in general and what techniques we used to define COBOL dialects. Then we discuss an ambiguity problem of COBOL that can not be solved easily.

First we enumerate the various COBOL dialects and their compilers.

- COBOL68, COBOL74, COBOL85. The first, second and third version of COBOL. These languages were released in 1968, 1974, and 1985 respectively.

- UNISYS COBOL, OS/VS COBOL, COBOL 370, Microfocus COBOL. These are software products: COBOL compilers, their corresponding COBOL dialect, and tools to support programming that particular COBOL dialect.

2.1 Formalisms to Define Syntax

We briefly discuss a number of formalisms in which it is possible to define the syntax of a programming language. They are, EAG [Kos91], Lex+Yacc [Joh75, LS86], Refine [Rea92], and ASF+SDF [BHK89, HHKR89]. First we will shortly characterize them and then we discuss their merits and shortcomings in the light of defining a COBOL syntax. We refer to the surveys [SK95] and [Wat91] for a general discussion on the above (and other) formalisms to define the syntax and semantics of programming languages.

EAG EAG stands for Extended Affix Grammar. It is an attribute grammar formalism based on two-level van Wijngaarden grammars [WMP+76], it allows semantic-directed parsing and imposes no restrictions on the grammar. We are not aware of a COBOL definition in EAG.

Lex+Yacc Lex+Yacc are well-known formalisms to generate scanners and parsers. They have been used to define COBOL grammars.

Refine Refine is a formalism that is supported by a commercial system to develop reengineering tools, it is based on research activities in the area of knowledge-based systems and programming environments performed at Kestrel. There exists a Refine/COBOL environment to analyze and modify legacy COBOL code.

ASF+SDF ASF+SDF stands for algebraic specification formalism and syntax definition formalism. It is an algebraic formalism with user definable syntax, it is modular and imposes no restrictions on defining a grammar. A programming environment supports this formalism and from a language definition, a complete programming environment, including a parser, can be generated. ASF+SDF has been used to define COBOL grammars.

Now we discuss relevant properties of the above formalisms in the light of defining COBOL syntax.

Modularity The formalisms EAG, Refine, and Lex+Yacc do not support modularity. Especially for a huge language like COBOL with so many dialects, modularity is essential for developing language definitions for various dialects and to increase grammar reuse. ASF+SDF does support the necessary modularity.

Parsing techniques The formalisms Refine, and Lex+Yacc are based on LR-parsing and while developing a context-free grammar the user may be confronted with shift-reduce and reduce-reduce conflicts. This can be an enormous problem when developing a definition for COBOL. In particular, when an existing definition is extended or modified to deal with a new dialect.

Arbitrary grammars The formalisms EAG and ASF+SDF support the development of arbitrary context-free grammars, even ambiguous grammars. In addition, EAG supports semantic-directed parsing: semantic values, like the types of variables, can influence the parsing of a program. This can be an aid in disambiguating COBOL expressions (see Section 2.3 for more details).

2.2 Techniques: Motivation and Choice

Obviously, there is no best formalism to define COBOL syntax. Lex+Yacc is proven technology but has as a serious drawback its lack of modularity and its conflicts. Refine which is based on the same parser generation technology also has these problems. On EAGs we can be short. They are not an option since EAGs are not very well supported by tools.

We have chosen for ASF+SDF because we think modularity is essential when describing the syntax of COBOL and its myriad of dialects. Moreover, the ASF+SDF formalism is supported by a programming environment, called the ASF+SDF Meta-Environment [Kli93], which is based on lazy and incremental scanner and parser generation techniques allowing flexible prototyping of language definitions. Moreover, it supports rapid prototyping of tools based on the COBOL syntax, see Section 5 for details.

Another argument in favor of the ASF+SDF Meta-Environment is that it is quite easy to change the context-free grammar rules of company-specific versions of the language, or to define a context-free grammar for languages that resemble COBOL such as Telon [Com92] (a language used to generate screens of the user-interface of an application). Since there are many dialects of COBOL this is a serious argument in favor of ASF+SDF.

Refine and ASF+SDF are comparable: both systems provide facilities for the development of tools to analyze programs. An important difference is that Refine is a commercial product whereas the ASF+SDF Meta-Environment is an academic product and can be obtained for free for universities and research institutes. Another difference is that ASF+SDF is based on algebraic specifications whereas Refine has a Lisp-like dialect to write specifications.

To a certain extend it is possible to translate a context-free grammar in SDF to Lex+Yacc. Since SDF permits arbitrary grammars unlike Lex+Yacc this may induce some extra work in manually solving shift-reduce and reduce-reduce conflicts. Furthermore, it is possible to obtain C-code from an ASF+SDF specification. This can be done either automatically or manually, both these approaches are elaborately described in [Hil96]. These possibilities are for us also an argument to work with ASF+SDF.

2.3 Ambiguities in COBOL

There are various forms of ambiguity, many of them are created by associativity and priority conflicts in the context-free syntax rules. Associativity problems can be solved by introducing either brackets in the sentence to be parsed, or by modifying the context-free syntax rules. The associativity problems can be solved in SDF by extending the context-free syntax rules with the appropriate associativity attributes. Priority problems can also be solved by introducing brackets in the input sentence, or by giving an ordening on the context-free syntax rules that cause the priority conflicts.

```
00001  IDENTIFICATION DIVISION.
00002  PROGRAM-ID. XisYorZ.
00003  DATA DIVISION.
00004  WORKING-STORAGE SECTION.
00005  01 X PIC 9 VALUE 1.
00006  01 Y PIC 9 VALUE 2.
00007 * 88 Z VALUE 2.
00008  01 Z PIC 9 VALUE 2.
00009  PROCEDURE DIVISION.
00010  MAIN.
00011    IF X = Y OR Z
00012      DISPLAY "s"
00013    ELSE
00014      DISPLAY "f".
00015
00016
00017 *       X | 1 | 1 | 1 | 1 |
00018 *       Y | 2 | 2 | 2 | 2 |
00019 *       Z | 2 | 1 | 2 | 1 |
00020 *  -------|---|---|---|---|
00021 *  output | f | s | s | f |
00022 * comment | 7 | 7 | 8 | 8 |
```

Figure 2: Example COBOL program.

However, there are ambiguities which are context-dependent. We will give a COBOL specific example: `X = Y OR Z` (see Figure 2). Depending on the status of `Z` the root of the parse tree is either `=` or `OR`. The program in Figure 2 prints `s`(success) or `f`(failure) depending on whether `Z` is a numeral or a predicate and depending on the values of `X`, `Y`, and `Z`; viz. the results in the table in the program. The conclusion is here that the parse tree depends on the status of `Z`. Note that the `*` on line 7 now forces `Z` to be a predicate but if the `*` is on line 8 then `Z` is a numeral.

This problem can be solved in SDF by defining a more general context-free syntax rule and by adding an extra pass to adjust the constructed parse tree.

Note that this problem will not occur when we would use EAG because of the facility of semantic-directed parsing. The other approaches that we discussed in Section 2.1 do have this problem, but we think it is harmless since we do not need this type of information in reengineering. At least we did not encounter this problem in real-world code.

3 Preprocessing of COBOL programs

As is well-known, COBOL is a huge language. One of the problems in specifying COBOL syntax is that many syntactic variations with exactly the same semantics occur in the language. We also know that the reduction of the size of a COBOL definition leads to better performance, and it makes the specification of tools more easy. So, it is a good idea to preprocess COBOL code by taking away *that* syntactic freedom that can be dealt with easily in a preprocessor. However, other transformations, like the use of periods instead of scope terminators are not typical preprocessing tasks. For such tasks it is better to specify a tool on top of the COBOL syntax that takes care of such transformations. Usually, such tools will facilitate maintenance and reengineering of COBOL code. For instance, the transformation of implicit to explicit scope terminators (see Section 5.3 for details).

We will devote the rest of this section to a pointwise discussion of a few typical preprocessing tasks.

Stripping The first steps in the preprocessing phase correspond with the stripping in Figure 1.

- Elimination of obsolete language constructs. Namely, the line numbering (columns 1–6) and comments (columns 73–80) can be omitted, thus obtaining 666-code. Since the line numbers and comments may occur on every location that permits a line break, the context-free grammar explodes if we would incorporate them. Therefore, we omit them, which is harmless since line numbers and comments do not have a semantics.

- Elimination of the continuation mark `-` in the C-Area (this is column 7). The reason is that otherwise you could have continuation marks on arbitrary locations in the context-free grammar. To avoid explosion of the grammar, we remove them. Note that this is harmless since continuation marks do not have a semantics.

Reformulation The second phase corresponds to the reformulation phase in Figure 1. This phase leads to reduced 666-code. Note that we reduce the number of syntactic possibilities without changing the semantics of the code.

- Elimination of syntactic variation to prematurily reduce many alternatives in production rules of the context-free grammar. For example, the construct `VARYING [IN] [SIZE] [FROM]` (where the text brackets stand for optional keywords) has eight alternatives. We substitute for all variations just `VARYING`. Another example is to transform `(>|GREATER) [OR] (=|EQUAL)` to `>=` thus reducing from eight to one alternative (with `(A|B)` we mean the choice between `A` and `B`).

- Transformation from mixed-case keywords to upper-case keywords. This leads to a simplification of the grammar.

Lexical disambiguation The last phase corresponds to the lexical disambiguation phase in Figure 1. After that phase we have lexically disambiguated 666-code.

- Elimination of lexical ambiguities. One of the problems that we see in COBOL is the overload of symbols with a different meaning depending on the context. This can cause parse problems. For example the comment marker in COBOL is `*` or `/` but these symbols are also arithmetic operations. Only in the C-Area they indicate comments. We substitute for the comment signs a special symbol so that we are able to distinguish it from the operations. The reason for this is technical: most scanning techniques are oriented from left to right whereas for this problem we would prefer a bidirectional scanning mechanism.

Our preprocessor is a one page `perl(1)` script [WS91]. Translation original code to lexically disambiguated 666-code takes about 1 second per 5 KLOC.

4 Defining COBOL Syntax

In this section we will discuss a general method to obtain a definition of a COBOL syntax that is based on code that has to be maintained or reengineered. As a starting point we will work with an as complete as possible COBOL definition in SDF, which is based on the ANSI standard of COBOL85 [Ame85]. This definition contains approximately 1100 production rules but it is still not complete [Man96]. We will discuss how to obtain the ANSI grammar, and then we discuss how to obtain a specific grammar that recognizes the code that actually has to be maintained or reengineered.

4.1 An ANSI COBOL85 Grammar

The standard way to develop a language definition for an existing language is to base it on an official document containing at least the syntax rules. Developing the definition based on syntax rules results in a parser which accepts only syntactical correct programs. In this section we discuss the syntax definition of COBOL85 based on the ANSI standard [Ame85]. A more detailed description of this process is reported on in [Man96].

Ideally this process leads to a more or less complete formal definition of the language, however the amount of success depends on the complexity and size of the language. Although COBOL85 is a huge language, a considerable part of the language is defined in [Man96] along the lines that we discuss below. It consists of about 1100 production rules and it took an undergraduate student four months to specify these rules. As far as we know this is the largest specification of COBOL85. We are aware of [BFKO93] and [BFK+94] where a tool is described that is based on a COBOL grammar of 170 BNF rules.

Almost all context-free grammar rules of the COBOL85 syntax in [Ame85] are translated into rules in the Syntax Definition Formalism (SDF) [HHKR89]. After the definition of a considerable number of rules in SDF, a number of test programs were parsed by the generated parser to test the correctness and completeness of the context-free grammar for ANSI COBOL85. Here we benefited from the advantage of using the ASF+SDF Meta-Environment since it supports incremental parser generation from the context-free grammar.

Next, we describe the problems were encountered during the translation of the context-free grammar rules in [Ame85] into SDF rules. This is a nontrivial process for the following reasons. First of all the size of the ANSI document is huge. But more important is that the ANSI standard is not very well structured, for instance, there is no clear distinction between syntactic and semantic rules. The ANSI document distinguishes the following sets of rules: the so-called *general format* that gives the formal definition of the syntax, the *syntax rules* that add properties to the definitions in general format, and the *general rules* that define the semantics. The arbitrary order of optional constructs is often defined informally in the syntax rules instead of in the general format. The syntax rules sometimes define semantics whereas the general rules define syntactic constraints. Moreover, the syntax rules and general rules are scattered over the entire document.

In [Man96] care has been taken of these problems and although the resulting definition is not yet complete it has a (far more) better structure than the ANSI document and is, therefore, useful as an electronic reference manual. In fact, the grammar in [Man96] could very well serve as the starting point for grammar manipulations as depicted in Figure 1 leading to a more specific grammar that is geared towards recognizing specific COBOL code that has to be analyzed.

To give the reader an idea of the translation process we present an example. We discuss the translation of the `PICTURE` string environment from the ANSI standard into SDF. Since the complete SDF definition of the `PICTURE` string environment takes about 90 production rules we confine ourselves to showing how the following rule in general format is translated to SDF rules.

$$\left\{ \begin{array}{l} \underline{\texttt{PICTURE}} \\ \underline{\texttt{PIC}} \end{array} \right\} \texttt{ IS pic-string}$$

The intended interpretation of the above rule is as follows. Uppercase words represent keywords, lowercase words represent non-terminals. The curly braces indicate that one of the alternatives has to be chosen. Underlined keywords are obligatory as opposed to not underlined ones.

The translation to SDF rules for this construct is below.

```
context-free syntax
  PIC-LIT OPT-IS PIC-STRING -> PICTURE-DEF
  "PICTURE"                 -> PIC-LIT
  "PIC"                     -> PIC-LIT
                            -> OPT-IS
  "IS"                      -> OPT-IS
```

The meaning of the above rules is as follows. We mention that SDF is a BNF-like formalism, except that the left- and right-hand sides are switched. The alternatives of a syntax rule are those rules which have the same non-terminal at the right-hand side. `PICTURE-DEF`, `PIC-LIT`, `OPT-IS`, and `PIC-STRING` are non-terminals, `"PICTURE"`, `"PIC"`, and `"IS"` are keywords. The first alternative of `OPT-IS` is an empty alternative.

In the above way the entire translation can be made and a major part of it has been carried out in [Man96]. The ensueing context-free grammar recognizes 666-code (see Figure 1).

4.2 A Restricted Grammar

Based on actual code of about 200 COBOL programs, a mixture of OS/VS COBOL and COBOL 370 programs, the SDF definition discussed in the previous section is adapted and reduced to a restricted grammar. The reason that we want a restricted grammar is to improve the performance of both the parser generator and generated parser. The performance is not optimal, since he production rules in the ANSI grammar have many alternatives (due to the syntactic freedom that COBOL provides), and these alternatives in turn have many members. This implies that the effort to construct the LR-table increases.

First we restrict the grammar by leaving out all those COBOL constructions from the ANSI definition that do not occur in the actual COBOL code. Next we carry out a reduction of the ANSI grammar which corresponds with the reformulation phase applied to the code, this yields a restricted grammar. The result is a restricted grammar that understands reduced 666-code.

4.3 A Retokenized Grammar

The next phase in the preprocessing was to lexically disambiguate the reduced 666-code. The reason to do this is not to optimize the performance but to solve some technical problems that deal with ambiguities. Hence, the grammar needs to be retokenized. To give an impression of the retokenization we treat a few typical aspects of it.

In COBOL a period (`.`) stands for at least three different things: a scope terminator, a `DECIMAL POINT`, or in a `PICTURE` it can be a formatting symbol (if `DECIMAL POINT` is set to `COMMA`). In order to distinguish a period that terminates a `PICTURE` we changed other two possibilities into a colon (`:`). Thus the grammar needs to understand colons.

To distinguish the comment markers `*` and `/` from the same tokens for the operations multiply and divide we changed comments into `#`. Hence, the grammar needs to understand that `#` indicates layout.

Certain scope delimiters can be dealt with more easily when they are renamed into a single symbol. The reason to do this is purely technical. We changed both `EXEC` and `END-EXEC` delimiters into a `$`. We changed the scope delimiters `==` in a `COPY` statement into `@`. So the grammar needs to be retokenized in order to understand these symbols.

4.4 A Unified COBOL Grammar

The grammar manipulations that we discussed thus far share the property that the grammar does not recognize incorrect programs. The restricted grammar can recognize less programs than the ANSI definition and the retokenized grammar parses a lexically disambiguated form of COBOL code (which can be seen as a dialect of COBOL). In this section we will discuss a grammar transformation that we call *unification*. It can parse incorrect programs. At this point we have to make an assumption on the code that we want to analyze: we assume that the code is correct, in the sense that it compiles. Fortunately, our point of departure is COBOL code that is compilable since it is our purpose to maintain or reengineer exactly such COBOL code. The reason that we want to unify the grammar is to further improve the performance of the generated parser. With the unification technique we can simplify the grammar drastically and improve the performance the parser even more drastically.

Next we will explain what unification is, and we will give an example of its use. The idea is that we take the least common multiple, to phrase it informally, of all the alternatives of a production rule, thus obtaining a rule that allows possibly more than is permitted by the compiler. The advantage of this approach is a reduction of the number of production rules that is quite dramatical. We illustrate our approach by giving an example. Just as with the ANSI definition we will show how we take care of the `PICTURE` string environment in the unified grammar. As an indication we give two (simplified) SDF rules originating from the ANSI grammar. They represent the rules for numeric, e.g. `9(6)`, and alphanumeric, e.g. `X(6)`, pictures:

```
[9]([1-9][0-9]*) -> PIC-STRING
[X]([1-9][0-9]*) -> PIC-STRING
```

We unify all production rules that define `PIC-STRING` (approximately 90 in the ANSI grammar) by allowing all the possible combinations of characters that a `PIC-STRING` could consist of. So, it sufficed to have a single production rule to define the complete `PICTURE` string in the unified grammar:

```
[0-9XxAa()pZzVvSszBCRD/,$+\-*:]+ -> PIC-STRING
```

The above rule expresses that one or more characters that are contained in the text brackets form a `PICTURE` string, called `PIC-STRING`. So the above unified rule recognizes *that* there is a `PICTURE` string, but not if it is correct. For example, it parses `:-)` but such a `PICTURE` string never occurs in compilable COBOL code. In fact, the unified grammar disentangles the structure of the COBOL program; which is exactly our intention.

In this way we unified the retokenized COBOL grammar. This resulted in a definition of approximately 300 production rules. We tested this definition on a large collection of COBOL programs (about 200 programs, varying from 20 to 80.000 LOC). We estimate that the job can be done in about a month.

5 Tools Based on COBOL Definitions

In this section we will explain how to use a COBOL syntax to specify tools that aid in reengineering and maintenance tasks. We specify such tools on top of the COBOL syntax. This is done with the Algebraic Specification Formalism [BHK89] (the ASF part of ASF+SDF). With algebraic equations it is convenient to specify the functional behavior of a tool. The ASF+SDF Meta-Environment supports the rapid prototyping and the development of such a tool. The equations that specify it take as input COBOL code. They perform the specified task by focusing on that part of the code that has to be inspected and/or changed. We will first specify an example tool to make our approach clear and then we discuss some maintenance/reengineering tools. We will provide input/output example fragments to show the functionality of the tools.

5.1 Example Tool

We present a simple example to illustrate how to specify tools on top of a COBOL grammar. The task of our example tool `pid` is to extract from a given COBOL program its `PROGRAM-ID`. Since the `PROGRAM-ID` of a COBOL program is located in the `IDENTIFICATION DIVISION`, we can restrict our tool to only inspect the identification division:

```
pid(I-div E-div D-div P-div) = pid(I-div)
```

The variables `I-div`, `E-div`, `D-div`, and `P-div` stand for the four divisions in COBOL. The above equation expresses that the tool will only focus on the `IDENTIFICATION DIVISION`. The behavior of `pid` on that division is described as follows:

```
1 - 6 (lines truncated to fit 2 cols) 73 - 80
---------------------------------------------
00001  IDENTIFICATION DIVISION.      PROGRAM-
00002 / PROGRAM-ID. Poodle           ID.PROGR
00003 * PROGRAM-ID. Dachshund        AM-ID.PR
00004                            PROGROGRAM-ID
00005 -AM-ID. Pid-Bull.              .PROGRAM
```

Figure 3: Cryptic example program.

```
pid(IDENTIFICATION DIVISION.
      PROGRAM-ID. Id.
      Author Inst Wdate Cdate Remarks) = Id
```

The variables `Id`, `Author`, `Inst`, `Wdate`, `Cdate`, and `Remarks` stand for the possible fields in the identification division. This last equation expresses that for a given `IDENTIFICATION DIVISION` the `pid` tool returns the variable `Id` which is the `PROGRAM-ID`.

This trivially specified tool is capable of extracting the correct `PROGRAM-ID` in the cryptic COBOL program of Figure 3. The program compiles with the Microfocus COBOL compiler. Of course, it is an artificial program, but it shows that even though the keyword `PROGRAM-ID` does not appear literally in it, our simple tool extracts the name: `Pid-Bull`. This is due to the fact that the tool has knowledge of the language the program is written in. Needless to say that with `grep`-like tools the correct `PROGRAM-ID` would never have been found.

Now that we have an idea of how to conveniently specify tools, we will discuss some tools that have been specified to aid in maintenance and reengineering. We describe them briefly and provide examples of their input/output behavior to illustrate their functionality and power.

5.2 `out2in` Tool

In COBOL we can distinguish so-called *in-line* and *out-of-line* `PERFORM` statements. An in-line `PERFORM` is an iterative construct that is comparable to, for instance, while-loops in PASCAL. An out-of-line `PERFORM` is actually a procedure call. The `out2in` tool transforms an out-of-line `PERFORM` to an in-line one. This is useful since for most analysis tools it is often convenient that the physical order of the code equals the execution order. Of course, it is possible to re-implement this functionality in every tool again, but it is, in our opinion, a better idea to make a tool that performs this task, so other tools can use this tool if necessary.

In Figure 4 an example fragment and its transformed output is shown. We can see that the out-of-line construction at the left-hand side is replaced by the contents of the called paragraph $Par\text{-}name_1$ resulting in an in-line construction. Note that the contents of an in-line `PERFORM` may only contain statements, so it is not a simple textual replacement, hence the omission of the periods after $Stat_2$ and $Stat_3$ in the in-line version. In fact, the `out2in` tool also flattens the code that the out-of-line `PERFORM` calls. The straightforward specification of the `out2in` tool consists of about 25 ASF equations.

```
Par-name_1.                          Par-name_1.
    Stat_1                               Stat_1
    Stat_2.                              Stat_2.
                  out2in
    Stat_3.         |-->                 Stat_3.
Par-name_2.                          Par-name_2.
    PERFORM                              PERFORM
        Par-name_1                           Stat_1
    UNTIL Cond.                              Stat_2
                                             Stat_3
                                         UNTIL Cond.
```

Figure 4: Out-of-line to in-line transformation.

```
IF Cond_1                     IF Cond_1
    IF Cond_2                     IF Cond_2
        IF Cond_3                     IF Cond_3
            Stat_1                        Stat_1
                        aei
        ELSE            |-->          ELSE
            Stat_2                        Stat_2
    ELSE                              END-IF
        Stat_3.                   ELSE
                                      Stat_3
                                  END-IF
                              END-IF.
```

Figure 5: Add `END-IF` transformation.

5.3 Add-`END-IF` Tool

In COBOL74 there is no explicit scope terminator for `IF` statements. All (open) `IF` statements are closed by a period. Furthermore, an `ELSE` can be closed by another `ELSE` that is matching a “higher” `IF`. In COBOL85 an explicit scope terminator `END-IF` has been added as an optional feature. The Add-`END-IF` tool (`aei` for short) transforms the implicit scope terminators into explicit ones. In Figure 5 we give an example fragment and its output.

It will be clear that the insertion of explicit scope terminators improves the legibility of the code. So, this tool aids in reengineering and maintenance. In practice, such transformations are part of migrations from COBOL74 to COBOL85 code. The specification of this tool comprises about 25 equations. We note that tools for the insertion of other scope terminators, like `END-READ` or `END-WRITE`, can be specified in a similar way.

5.4 Generation of an Unparser

One of the advantages of using the ASF+SDF Meta-Environment is that given an SDF definition of COBOL it is possible to generate tools for COBOL programs. We already saw the incremental generation of parsers during the development of a COBOL grammar. An elaborate discussion on the subject of generating such

```
PROCEDURE DIVISION.
MAIN.
   IF X = Y OR Z
      DISPLAY "s"
   ELSE
      DISPLAY "f".
```

Figure 6: Typesetted fragment of a COBOL program.

tools is out of scope. We only mention that the development of unparsers is supported by an unparser generator tool [BV96], since unparsers aid in reengineering and maintenance. The unparser generator generates a set of equations which can be fine-tuned in order to obtain the desired layout of COBOL programs. The equations translate a COBOL program to a language independent expression that can be translated to ASCII-text or TEX-code. The TEX output can be used to produce correct documentation generated directly from the code. In Figure 6 we have extracted the `PROCEDURE DIVISION` of the program in Figure 2 to give the reader an idea of the functionality of our COBOL formatter.

The purpose of an unparser is not only to improve the readability of code, but it can also be of use when migrating from one compiler to another. The Microfocus compiler does not mind when, for instance, `PROGRAM-ID` does not start in the A-area, but another compiler might (see the program in Figure 3). Then an unparser is useful to modify existing code to meet the standards of the new compiler. Another application of an unparser is that code can be made migration-ripe by uniformizing it so that certain assumptions on the form of the code can be made in order to facilitate massive automatic migrations.

The generation and fine-tuning of an unparser for COBOL given the grammar we discussed in Section 4.4 was less than one week work. This is due to the fact that although for each production rule equations are generated, half of them need small modifications, since it is impossible to predict, in general, how an arbitrary language has to be formatted.

6 Conclusions

In this paper we have motivated that in order to specify reengineering and maintenance tools for COBOL it is essential that such tools are built on top of a COBOL definition that is geared towards the specific COBOL code that has to be processed by such tools. We have shown that it is sensible to separate the development of the tools from the development of the grammar that the tools use. To aid in the development of the grammars, we have presented a general method to specify COBOL that uses on the one hand an ANSI COBOL85 grammar and on the other hand real code that has to be reengineered or maintained. Since there are many COBOL dialects such general methods are essential in the development of tools for COBOL. It turned out that specifying tools on top of such grammars is now a convenient task.

We used modular techniques to specify the grammars and the tools. This guarantees that (parts of) grammars and tools can be reused without problems since those techniques support definition of arbitrary context-free grammars.

So we do not have the traditional shift-reduce and reduce-reduce conflicts that come with Lex+Yacc-like formalisms.

Concluding, we hope have shown that the quality of reengineering and maintenance tools drastically improves when these tools are based on a unified context-free grammar of the language in which the code has been written.

References

[Ame85] American National Standards Insitute, Inc. *Programming Language – COBOL*, ANSI X3.23–1985 edition, 1985.

[BFK+94] P. Baumann, J. Fässler, M. Kiser, Z. Öztürk, and L. Richter. Semantics-based reverse engineering. Technical Report 94.08, Department of Computer Science, University of Zurich, Switzerland, 1994.

[BFKO93] P. Baumann, J. Fässler, M. Kiser, and Z. Öztürk. Beauty and the Beast or A Formal Description of the Control Constructs of Cobol and its Implementation. Technical Report 93.39, Department of Computer Science, University of Zurich, Switzerland, 1993.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[BKV96] M.G.J. van den Brand, P. Klint, and C. Verhoef. Re-engineering needs generic programming language technology. Technical Report P9618, University of Amsterdam, Programming Research Group, 1996. To appear in ACM SIGPLAN Notices.

[BKV97] M.G.J. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation – an annotated bibliography. *ACM Software Engineering Notes*, 22(1):57–68, 1997.

[BV96] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[CC90] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[CCL96] G. Canfora, A. Cimitile, and A. De Lucia. Specifying code analysis tools. In S.A. Bohner and Cimitile A, editors, *International Conference on Software Maintenance*, pages 95–103. IEEE, 1996.

[Com92] Computer Associates International, Inc. *CA-TELON — Programming Concepts Guide*, 2.1 edition, 1992.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[Hil96] J.A. Hillebrand. *Experiments in specification re-engineering.* PhD thesis, University of Amsterdam, 1996.

[Joh75] S.C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Kos91] C.H.A. Koster. Affix grammars for programming languages. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 358–373. Springer-Verlag, 1991.

[LS86] M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator.* Bell Laboratories, unix programmer's supplementary documents, volume 1 (PS1) edition, 1986.

[Man96] P.I. Manuel. ANSI COBOL III in SDF + an ASF definition of a Y2K tool. Master's thesis, University of Amsterdam, Programming Research Group, 1996.

[NM93] Ph. Newcomb and L. Markosian. Automating the modularization of large COBOL programs: application of an enabling technology for reengineering. In R.C. Waters and E.J. Chikofsky, editors, *Proceedings of Working Conference on Reverse Engineering*, pages 222–230. IEEE Computer Society Press, 1993.

[Rea92] Reasoning Systems, Palo Alto, California. *Refine User's Guide*, 1992.

[SK95] K. Slonneger and B.L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach.* Addison-Wesley, 1995.

[Wat91] D.A. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall, 1991.

[WMP+76] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisker. *Revised Report on the Algorithmic Language* ALGOL 68. Springer Verlag, 1976.

[WS91] L. Wall and R.L. Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., 1991.

Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site (`http://www.fwi.uva.nl/research/prog/reports/`) or by anonymous ftp to `ftp.fwi.uva.nl`, directory `pub/programming-research/reports/`.

[P9702] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Reengineering COBOL Software Implies Specification of the Underlying Dialects.*

[P9701] E. Visser. *Polymorphic Syntax Definition.*

[P9618] M.G.J. van den Brand, P. Klint, and C. verhoef. *Re-engineering needs Generic Programming Language Technology.*

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[P9610] T.B. Dinesh and S.M. Üsküdarlı. *Specifying input and output of visual languages.*

[P9609] T.B. Dinesh and S.M. Üsküdarlı. *The VAS formalism in VASE.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*

[P9604] E. Visser. *Multi-level specifications.*

[P9603] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Reverse engineering and system renovation: an annotated bibliography.*

[P9602] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives.*

[P9601] P.A. Olivier. *Embedded system simulation: testdriving the ToolBus.*

[P9512] J.J. Brunekreef. *TransLog, an interactive tool for transformation of logic programs.*

[P9511] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. *Grid protocols based on synchronous communication: specification and correctness.*

[P9510] P.H. Rodenburg. *Termination and confluence in infinitary term rewriting.*

[P9509] J.A. Bergstra and Gh. Stefanescu. *Network algebra with demonic relation operators.*

[P9508] J.A. Bergstra, C.A. Middelburg, and Gh. Stefanescu. *Network algebra for synchronous and asynchronous dataflow.*

[P9507] E. Visser. *A case study in optimizing parsing schemata by disambiguation filters.*

[P9506] M.G.J. van den Brand and E. Visser. *Generation of formatters for context-free languages.*

[P9505] J.M.T. Romijn. *Automatic analysis of term rewriting systems: proving properties of term rewriting systems derived from* ASF+SDF *specifications.*

[P9504] M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman, and E. Visser (editors). ASF+SDF*'95: a workshop on Generating Tools from Algebraic Specifications, May 11&12, 1995, CWI Amsterdam.*

[P9503] J.A. Bergstra and A. Ponse. *Frame-based process logics.*

[P9208c] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra (revised version of P9208b).*

[P9502] J.A. Bergstra and P. Klint. *The discrete time ToolBus.*

[P9501] J.A. Hillebrand and H.P. Korver. *A well-formedness checker for μCRL.*

[P9426] P. Klint and E. Visser. *Using filters for the disambiguation of context-free grammars.*

[P9425] B. Diertens and A. Ponse. *New features in PSF II: iteration and nesting.*

[P9424] M.A. Bezem and A. Ponse. *Two finite specifications of a queue.*

[P9423] J.J. van Wamel. *Process algebra with language matching.*

[P9422] R.N. Bol, L.H. Oei J.W.C. Koorn, and S.F.M. van Vlijmen. *Syntax and static semantics of the interlocking design and application language.*

[P9421] J.A. Bergstra and A. Ponse. *Frame algebra with synchronous communication.*