Development, Assessment, and Reengineering of Language Descriptions

Alex Sellink and Chris Verhoef

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

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

Abstract

We discuss tools that aid in the development, the assessment and the reengineering of language descriptions. The assessment tools give an indication as to what is wrong with an existing language description, and give hints towards correction. From a correct and complete language description, it is possible to generate a parser, a manual, and on-line documentation. The parser is geared towards reengineering purposes, but is also used to parse the examples that are contained in the documentation. The reengineered language description is a basic ingredient for a reengineering factory that can manipulate this language. We demonstrate our approach with a proprietary language for real time embedded software systems that is used in telecommunications industry. The described tool support can also be used to develop a language standard without syntax errors in the language description and its code examples.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring; D.3.4. [**Processors**]: Parsing.

Additional Key Words and Phrases: Reengineering, System renovation, Language description development, Grammar reengineering, Document generation, Computer aided language engineering (CALE), Message Sequence Charts.

1 Introduction

Since the emerge of computer languages, the need to describe languages in a precise way became an indispensable part of computer science. In his paper on the syntax and semantics of the proposed international algebraic language, Backus [2] writes: ‘we shall need some metalinguistic conventions for characterizing various strings of symbols. To begin, we shall need *metalinguistic formulae*.’ Then he introduced using an example what is now widely known as the Backus-Naur Formalism. In virtually all documents that give a precise language description the method of Backus is used: first the syntax description notation is explained using an example accompanied with some conventions, and then the language description itself follows. In this way myriads of dialects of the Backus-Naur Formalism emerged. They are referred to as BNF, or EBNF, for extended BNF, or metasyntax, metalanguage.

Language descriptions serve more than one purpose: they are used as a guide to implement tools such as compilers or they serve as a reference manual for users. We use language descriptions to implement tools that serve the reengineering of those languages. Such grammar descriptions form the basis of our approach towards reengineering. Let us give an idea to make this more concrete. It is possible to generate all kinds of prefab components that are useful in an environment for reengineering. We can generate a native pattern language from a context-free grammar that can be used to recognize code fragments [30]. It is possible to generate full documentation for such a language [9]. In [8] we generate components for software renovation factories. A sophisticated parser can be generated from this grammar [17]. A structured editor can be generated from the grammar [22]. It is also possible to generate complete programming environments from context-free grammars. In order to generate such environments, one needs an environment as well. The ASF+SDF Meta-Environment [19] is such an environment. We use it for the generation of tool factories [8]. SDF stands for Syntax Definition Formalism [16], it can be used to define the syntax of a language. ASF stands for Algebraic Specification Formalism [3], it can be used to describe the semantics of a language. The combination is thus adequate for defining syntax and semantics of languages and the ASF+SDF Meta-Environment is the supporting environment for both formalisms.

It is not a trivial task to construct a grammar for reengineering purposes. First of all, such a grammar should have certain properties that make reengineering easy. Secondly, since reengineering problems do not have the habit to reside in small languages, the development process is time consuming. For instance, many academics and companies have struggled with a language definition for COBOL in order to create a decent parser for reengineering targets. Due to the myriad of COBOL dialects, it can be the case that such a grammar itself needs reengineering. Such grammars can suffer from large maintenance problems. In [7] this was called the Year 1999 problem: before that date the grammars had to be ready so that the generated parsers can be used to analyze Year 2000 problems. We refer to [7] for an overview of current parser technology that is used in reengineering and problems that induce maintenance problems on grammars. Since in reengineering, the grammar seems to be the variable and the problem the constant, grammars should be modifiable and tool support should be insensitive to such modifications. Therefore, generating everything from a grammar is in our opinion a solution. According to [28] there are two problems with parser-based technology: first the stringent constraints on the input, and second it is problematic to extend existing parsers. We solve this by using modular grammars that are easily modifiable, and we use unification of grammar rules that are not important for reengineering tasks [5]. Despite the use of new technology, it is still not easy to develop a new grammar for reengineering purposes or reengineering old grammars to migrate them to the new technology. Therefore, we developed tools to support development, the assessment and the reengineering for language descriptions.

Let us give an example of current practice in serious language description documentation. We have encountered the following case in the literature. The language description document (an ITU standard) of so-called Message Sequence Charts [18], is an MS-Word document. In order to extract the BNF syntax, the PostScript version of the Word document was first converted to `ASCII`, then using a script called `extract.perl` [12], the (nonlexical) BNF rules were extracted. Then, fourteen manual corrections were needed (see the comments in the script [12]). Then the BNF rules were fed to another script that generates an HTML document so that the BNF rules can be browsed. This is obviously not an optimal situation.

Our approach to develop a standard would be to write a complete grammar using the ASF+SDF Meta-Environment and put the accompanying text in comments in the grammar specification. Then, by using the formatting technology discussed in [9] we generate a LaTeX document and produce hard copy. Using technology presented in [15], we can generate the HTML version, and using the ASF+SDF Meta-Environment we can generate a programming environment for the language. As can be seen, we take the opposite route to develop a standard. The advantage of our approach is that the grammar is complete, its syntax is checked and the examples in the document are parsed using the generated parser.

Results After finishing this paper, we used this technology to generate a complete software renovation factory from the source code of a compiler. See [31] for details. Moreover we used an extension of the technology described in this paper to generate a correct VS COBOL II grammar from its corresponding IBM Manual (see [25] for a URL). Correct means here that our generated parser could parse 800.000+ lines of VS COBOL II code without problems. Using this technology, it is possible to generate correct grammars for other languages as well, e.g., SQL, CICS, PL/I, and so on. This means that it is viable for both obscure and well-known languages to recover their grammars, and as such to speed up tool development for software renovation. So, the results in this paper are of significant economic importance.

Organization In Section 2 we discuss syntax definition languages and their syntax and semantics. We discuss parsing of language description documents, and pretty printing them. In Section 3 we describe a number of quality assessment tools that give an insight in the current state of language descriptions. Then in Section 4 we discuss tools that aid in the reengineering of a language description. We discuss modularization and conversion to an executable format. We illustrate that this step aids in the detection of semantic errors in a language description document. We illustrate the use of our tools by applying them to a real-world example: a language description document of a nontrivial language that is used in telecommunications industry. In Section 5 we discuss an assembly line that takes care of the automatic migration from the format used in manuals to an executable format. Finally, in Section 6, we discuss our conclusions and future work.

Related work In [10] the ASF+SDF Meta-Environment was used to check the formal parts of a book on action semantics [27]. In [10] it has been reported that these checks revealed about one error on every two pages. We use similar technology to reveal errors in language description documents. In a tutorial on functional programming [13] one of the exercises is constructing a BNF parser. We use a BNF parser to parse language descriptions. In the GMD Toolbox for Compiler Construction, there are conversion tools available to convert certain BNF dialects into others. We implemented a converter from BNF to our preferred syntax definition language SDF in order to reveal semantic errors in language descriptions. In [11] the Yacc part of a C++ grammar is automatically converted to SDF using the ASF+SDF Meta-Environment. In that paper the C++ grammar is used to parse C++ programs in order to perform optimizing source to source transformations using an algebraic specification. As far as we know, no other publications exist where language definitions are reengineered so that the resulting grammars can be used as input for renovation tools.

Acknowledgements We thank Arie van Deursen (CWI), Dinesh (CWI), and Paul Klint (CWI) for pointers to related work. We thank Joakim Ek (Frameworks, Ericsson Software Technology AB), Leif Ekman and Johanna Persson (both Reengineering Center, Ericsson Software Technology AB), Roger Holmberg (Ericsson Utvecklings AB), and Magnus Nilsson (Software Engineering, Ericsson Software Technology AB) for inviting us over, for visiting us in return, and for their comments on an earlier version of this paper. We thank them for their willingness to provide us with their proprietary language description that is the running example in this paper. We thank the reviewers for their valuable comments.

2 Syntax Definition Languages

We think of metasyntax as a domain specific language. It is a language geared towards the definition of the syntax of (programming) languages. We will call such languages *syntax definition languages*. Any dialect of the Backus-Naur Formalism (BNF) is an example of a syntax definition language. The Syntax Definition Formalism (SDF) [16] is another example of a syntax description language. In contrast with BNF, SDF is not available in a myriad of dialects: it is part of a support environment (the ASF+SDF Meta-Environment) as a means to define syntax.

A document containing a language description can be seen as a program written in a syntax definition language;

the text in natural language is the comment. This phenomenon is widely known as literate programming [21]. Seen from this perspective, maybe the most literate programs that one can think of are language description documents, in particular a standard. In our opinion, it would be natural to parse and compile such documents. For, it is a fact of life that programs that are neither parsed, nor compiled have a high risk of containing errors. Indeed, we have experienced that the language descriptions that we have parsed and compiled often contain errors.

Let us first explain what we consider parsing in the case we are dealing with a language description document. There are at least two possibilities. First of all we can think of a parser for the metalanguage that is explained in the preliminary part of the document. Secondly, we can think of a parser of the language that is described. Using the metalanguage parser we can parse the language description, and using the parser for the language itself we can parse the code that we wish to describe with the language (for instance, the code examples in the language description document).

There are at least two possibilities for the meaning of compilation of a language description document. First, the documentation in some typeset form can be generated, or a parser can be generated from the documentation. Note that this idea is not new: it is in accordance with the philosophy of literate programming [21]. Let us give an example. From various sources (standards, manuals, programs) we constructed a COBOL grammar for reengineering purposes [5]. On the one hand we use the grammar as input for a parser generator, so we can parse code that we wish to analyze or transform [8, 6, 30] (in this paper we use this approach as well, for details we refer to Section 5). On the other hand we can generate typeset documentation using technology described in [9]. This document can be used by operators of our COBOL factory. For COBOL this amounts to a 25 page LaTeX document with a table of contents, natural language, cross references, etc. This type of documentation is in compliance with the so-called book paradigm [29]. In [15] technology has been implemented that enables the generation of on-line documentation in HTML format. We can use this to create an HTML document with cross references.

2.1 Syntax Errors and Semantic Errors

The first step in language description development is, in our opinion, parsing the language description itself. For, a typographical error in a language description now becomes a syntax error during parsing. Although this phase is obvious to us, this is not a common approach (but see [10] where errors in a book on action semantics [27] are detected in a similar way). Many language description documents, including standards that we have parsed contain syntax errors.

Apart from syntax errors in language description documents they also contain semantic errors. Let us first explain what we consider the semantics of a language description document. The semantics of a language description program can be seen as the set of objects that can be recognized by this program. Suppose, for instance, that we have the following BNF program `x ::= 'a'`. This program can recognize a single `a`. The BNF program consisting of the rules `x ::= 'a'` and `x ::= x x` recognizes `a`, `aa`, `aaa`, etc. A semantic error is recognition of other objects than intended. So if it was our intention to recognize a `b` there is a semantic error in the BNF program.

A typical situation is that the parser (generated from the language description document) does not recognize the example programs that are contained in the document. Another error is that the generated parser does recognize examples but in an unexpected way, for instance a keyword is parsed as an identifier. Then either the language description contains an error, or the example. In either case, the problem should be resolved.

We have seen errors where it is obvious from the context what their cause was. But we have also encountered situations where we had no idea. Testing all the possibilities with the compiler is an option, but then we discover the interpretation of the compiler constructor. Another compiler constructor may have another interpretation. Our solution for this is to restrict ourselves to a certain hardware platform. For us this is not a problem: if we need to reengineer code, it is our intention that it will run on a already known architecture. We note that for a language description document that has the status of an official standard, our solution breaks down. Therefore, it seems useful to apply our methods to develop standards for languages, since this would reveal such errors before the standard will be published.

2.2 Parsing of Documents

In order to explain our approach we treat a real-world example. We were contacted by Ericsson Reengineering Center to investigate whether it is possible to cooperate in tool development that can be used for their proprietary language. We asked for a language description, and they mailed us an electronic version of the language manual. The running example in this paper is their language description. We analyzed their language description so that we had a clear view of how much work it would be to turn it into a working grammar. We abbreviate the language that Ericsson uses by SSL which stands for Switching System Language. SSL is a real-time language developed by Ericsson in the early seventies and modernized in the early eighties.

The intended hardware platforms are either the central or regional processor of the AXE 10 program controlled telephone switch. SSL is the language that is used to run on the central processor. SSL is compiled to an assembly language that runs on the central processor. We note that, although SSL was designed for telephone switches only, the language is substantial (over 200 keywords and the hardcopy language reference manual is over 150 pages).

In order to parse a language description document, we need a parser. The information we need is contained in the section describing the metasyntax. We specified the metasyntax in SDF. The ASF+SDF Meta-Environment incrementally generates a GLR parser [17] and a structured text editor [22]. So we can parse existing language descriptions and we can build them using a structured text editor. If we have the language description document available in some electronic form, we can extract the formal language description using standard Unix tools. In the case of SSL we obtained an HTML file from Ericsson Reengineering Center. So, it was not hard to extract the syntax rules. Below we depict the definition of the BNF dialect that is used in the HTML file. We call it SBNF, short for SSL Backus Naur formalism. The description consists of an example with explanations and some conventions. We mention the example and the conventions below:

```
if-statement ::=
  'IF' ['NOT'] cond-exp 'THEN' sequence-of-statements
    [{'ELSIF' cond-exp 'THEN' sequence-of-statements}]
    ['ELSE' sequence-of-statements]
  'FI' ';'
cond-exp ::=
  variable <'=' | '/='> value
```

The following conventions are used:

- Terminals (representing actual SSL keywords and symbols) are surrounded by single quotes.
- Nonterminals are lower case words.
- `[ ]` represents optional parts of a construction. From the example above we see that an `IF` statement may or may not contain a `NOT` keyword.
- `{ }` represents a non-empty sequence of elements. For example, we see that an `IF` statement may contain a sequence of `ELSIF` branches. (Since the `ELSIF` construction is also surrounded by `[ ]`, the sequence of `ELSIF` branches may be left out altogether.)
- `< >` represents a choice of elements. The alternatives are separated by vertical bars. We see that a conditional expression may contain either an equality operator (=) or an inequality operator (/=), but not both.

Note that, for instance, `cond-exp` is a nonterminal and `IF` is a terminal. A nonterminal is also called a sort. The language description is supposed to be a syntactically correct SBNF program. We opened a structured text editor that understands SBNF and loaded the SBNF program consisting of the complete language description that we extracted from the HTML file. It did not parse. We will discuss a few typical errors, to give the reader an idea. The first error that we found were missing quotes around an equality sign. We found many more such errors and repaired them without effort. Another problem that we encountered was that text brackets indicating optional constructs in SBNF did not match. An example of such a rule is:

```
single-signal-reception-statement ::= 'ENTER'
  signal ['WITH' signal-datum [{',' signal-datum}] ';'
```

There is a `]` missing. A short inquiry of example code revealed that the closing text bracket should be inserted just before the quoted semicolon. This error was propagated in the document, apparently due to a copy-paste editing session.

The convention that nonterminals are lower case words was violated. We detected nonterminals containing digits, upper case letters, and in one case even an ellipse (`flconnfid...`). We solved this by relaxing the conventions that were given in the manual. With the aid of real-world code or a knowledge expert we can reconstruct the actual situation and repair the language description accordingly.

2.3 The Language Description Parser

Creation of a parser amounts to defining the grammar of the metasyntax in SDF. The parser is generated on-the-fly by the ASF+SDF Meta-Environment [17]. Let us depict the SDF module containing the grammar of SBNF:

```
imports Layout
exports
  sorts
    SBNF-Terminal SBNF-NonTerminal
    SBNF-Element SBNF-Rule SBNF-Program
    SBNF-Elements
  lexical syntax
    "'" ~[']* "'"               -> SBNF-Terminal
    [A-Za-z0-9\-]+              -> SBNF-NonTerminal
  context-free syntax
    "flconnfid..."   -> SBNF-NonTerminal
    SBNF-Terminal    -> SBNF-Element
    SBNF-NonTerminal -> SBNF-Element
    SBNF-Element* -> SBNF-Elements
    "[" SBNF-Elements "]" -> SBNF-Element
    "{" SBNF-Elements "}" -> SBNF-Element
    "<" { SBNF-Elements "|" }+ ">" -> SBNF-Element
    SBNF-NonTerminal "::=" SBNF-Elements -> SBNF-Rule
    SBNF-Rule+ -> SBNF-Program
```

Let us discuss this module. First of all, SDF is a modular syntax definition language. We import an SDF module that defines layout for SBNF. Since this was not in the conventions, we use a standard SDF layout module. In the so-called `exports` section, we describe the complete syntax of SBNF. We declare the used nonterminals in the `sorts` paragraph. In the `lexical syntax` paragraph we define terminals and nonterminals of SSL: the first lexical rule says that the terminal quote (`'`) followed by zero or more characters that are not a quote followed by a quote (`'`) is an `SBNF-Terminal`. This is the first convention of the metasyntax. The second convention is defined in the other rule: a string that consists of one or more upper case letters, lower case letters, digits, or minus signs is an `SBNF-NonTerminal`. Note that we relaxed the convention, in order to parse the SBNF program. Then we enter the `context-free syntax` paragraph. First we deal with the `flconnfid...` by declaring this to be an `SBNF-NonTerminal`. Then we construct from the smallest parts of the grammar the elements that can be present in an SBNF rule. Since an `SBNF-NonTerminal` and an `SBNF-Terminal` can occur in an SBNF rule, we inject them into the sort `SBNF-Element`. Zero or more such elements are called `SBNF-Elements`. This is the next rule. Then we implement the next three conventions. If we have some `SBNF-Elements` and put text brackets around them, they become a new `SBNF-Element`. The intended interpretation is, of course, the optional part of a construction. In a similar way we implement *one or more occurrences* using the curly braces. The next rule expresses that a list of one or more `SBNF-Elements` separated by the symbol `|` surrounded by angle brackets is again an `SBNF-Element`. Now we can construct an `SBNF-Rule`: it is a `SBNF-NonTerminal` followed by the terminal `::=` followed

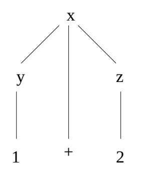

Figure 1: Parse tree of `1 + 2`.

by some `SBNF-Elements`. Finally, one or more `SBNF-Rules` forms an `SBNF-Program`.

Constructing and testing this grammar was not a hard job. Furthermore, removing the syntax errors was also easy. It was maybe one hour work to extract the SBNF rules from the HTML file, to make the grammar, and to remove the errors. Using the approach that we describe here makes it a trivial task to remove all the syntax errors in a language description document.

2.4 Language Description Formatter

Using technology described in [9] we are able to generate a formatter for language description programs. We use this formatter to pretty print the SBNF rules, or to generate LaTeX documentation. In [15] the formatting technology of [9] has been used so that HTML code can be generated. The generated formatter that we use for pretty printing the SBNF code has been fine tuned so that it satisfies the coding style that is used in the SSL manual.

3 Quality Assessment Tools

In order to judge the quality of an existing language description, it is useful to have tools that can give an indication of the quality of a grammar. Let us first explain what we experienced to be sensible measures to assess the quality of a language description. Suppose we have a language consisting of the rules `x ::= y '+' z`, `y ::= '1'`, and `z ::= '2'`. This language is capable of recognizing `1 + 2`. We depict the parse tree of this term in Figure 1.

As we can see, there is one sort at the top of this tree, and at the bottom we have only terminals. Now suppose that in the above language description the rule `z ::= '2'` was missing. Then the sort `z` would be at the bottom of the parse tree. We call this a bottom sort. If such a sort occurs there is an indication that part of the grammar is missing. Now suppose that the rule `x ::= y '+' z` would be missing. Then we have two sorts at the top of the parse tree: `y` and `z`. We can parse the symbol `1` and the symbol `2` but we are not able to parse their sum. The situation that we have more than one top sort is an indication that a production rule is missing that 'glues' grammar rules together. Therefore, we made tools that list top and bottom sorts. In the remainder of this section we discuss a number of such tools, and we indicate what their use was by applying them to our running example language description for SSL. We like to stress that our tools are in fact very simple to implement. The contribution of our work is to advocate the *use* of such simple tools to solve problems that are perceived to be hard by many colleagues in our field. Since the tools are so simple, they are easily implemented by others as well.

top and bottom sorts We implemented two tools that report the number of top sorts and the number of bottom sorts. A top sort is a sort that is defined but not used, a bottom sort is a sort that is used but not defined. In case of the SSL language description we reported 107 top sorts and 118 bottom sorts. We note that this situation is not uncommon, after all, such documents were never parsed before. The high number of bottom sorts is explained is as follows: since it is not possible in BNF to express lexical syntax, there is not a single lexical syntax rule in the SBNF program. This means that all the rules that are comparable with `y ::= '1'` are missing. Let us give an example fragment of the SBNF program.

```
ioconnection ::= 'ID' 'IS' vn-1 ','
   'ABRANCH' 'IS' label ',' 'CODE' 'IS' vn-2
   [',' 'BUFFER' 'IS' buf] [',' 'POINTER' 'IS' p]
```

The sorts `vn-1`, `label`, `vn-2`, `p`, and `buf` are all bottom sorts. In fact, variable names of a certain types are meant. This implies that addition of lexical rules will solve this. We estimate that those sorts can be turned into one sort representing identifiers. So it might be the case that in the above rule we can substitute a sort `Identifier` for all of them, and give a lexical definition for this sort. Another solution for this problem is to define all those sorts lexically. In order to solve it satisfactorily, we either need to see more SSL code or compiler experts of Ericsson inform us what the best solution is.

The high number of top sorts is caused by the fact that the rules connecting the language constructs are often missing. A reason for this could be that the authors of the manual focussed on describing the individual language constructs, and not on the overall structure of the language. Let us give an example. For instance `statement` is a bottom sort and `case-statement` is a top sort. Obviously, a production rule like `statement ::= < ... | case-statement | ... >` is missing. This rule expresses part of the overall structure of the language: it answers the question "What kind of statements do exist in SSL?" Note that it is also possible to turn `case-statement` into `statement`. We do not opt for the latter choice, since the grammar will become shallow. This implies that it becomes more work to implement tools to reengineer code that has been written in the language, which is our purpose for defining a correct language description of SSL. Of course, for language description purposes another option may be more appropriate.

list-sorts We also have a tool that lists all the sorts. This may seem a silly tool, but it is a useful tool as well. First, it puts the number of top and bottom sorts in perspective, and second it reveals other problems as well. We counted initially 294 sort names. One sort name was `FLEXTERNAL`. Since we relaxed the conditions on the sort names (because they contained forbidden upper case letters) the keyword `FLEXTERNAL` was parsed as a sort name. This could happen since in one of the SBNF rules the quotes for the terminal FLEXTERNAL were missing. We repaired this. We also found the sorts names flconfid and flconnfid. After inspection we learned that flconfid contained a typographical error. It should have been flconnfid. So the language description contains 292 sort names. Thus, we found two nasty errors. Moreover, we can conclude that the number of top and bottom sorts is high.

list-keywords A tool that gives a list of all the keywords is also useful. It gives an idea of the size of the language. More importantly, it reveals errors in the language description. In our running example we found some errors. We will treat an example. In the following rule temp is a keyword due to the quotes. Inspection of other rules learned that it is in fact a sort name.

```
convert-call ::=
  'CONVERT' conversion [',' 'CODE' 'IS' 'temp']
    [',' 'POINTER' 'IS' pointer] ';'
```

list-redundant-rules We also implemented a tool that reports multiple occurrences of production rules. The language description of SSL counts 203 production rules. We detected 12 redundant production rules. So 6 % of the productions is redundant. The reported rules need to be removed, since double rules make the grammar ambiguous. We note that our tools are insensitive to formatting differences. So, rules that are the same (modulo spaces, newlines, tabs, indentation, formatting, and so on) are still identified as being the same. To give an idea of the ease of implementing the redundant rules tool we depict the ASF specification of this tool. We recall that ASF stands for Algebraic Specification Formalism.

```
[01] double-rules(SBNF-Rule1* SBNF-Rule1
     SBNF-Rule2* SBNF-Rule1 SBNF-Rule3*) =
     double-rules(SBNF-Rule1* SBNF-Rule1
     SBNF-Rule2* SBNF-Rule3*) SBNF-Rule1
[default-02] double-rules(SBNF-Rule1*) = NO DOUBLE RULES FOUND.
[03] NO DOUBLE RULES FOUND. SBNF-Rule1+ = SBNF-Rule1+
```

We generated a native pattern language from the SBNF grammar, see [30] for details. So we have variables available in order to define tools on SBNF. In the ASF specification we see five of them. The variables SBNF-Rule1* – SBNF-Rule3* represent zero or more occurrences of arbitrary SBNF rules. The variable SBNF-Rule1 matches exactly one production rule. The pattern that we match checks for double occurrences of SBNF-Rule1, hence it occurs twice. If it occurs, we throw one SBNF-Rule1 over the edge and recursively search the remaining rules. The default rule that we have returns that there is no redundancy. The third rule removes this message as soon as a double is found. This is represented by the variable SBNF-Rule1+ with the plus representing one or more occurrences of a production rule.

The explanation for the redundancy in SSL is as follows. Again since the constructions are treated as separate entities, sorts that are necessary in more rules are reiterated. In fact, we can see this as a form of code duplication with all the maintenance problems that come with it. At first sight the use of words like maintenance problem and code duplication may seem exaggerated, however, we actually found such maintenance problems during modularization of the grammar. See Section 4 for more details.

rule complexity The above tools give an indication of the overall quality of a language description. It is also useful to assess the quality of individual production rules. For, if they are correct but complex, the language description will miss its purpose of explaining the language. In our opinion, a language description should be as self-documenting as possible. Well-named additional sort names are preferable to comments in the accompanying natural language and less sort names. We can reduce the complexity of a production rule by introducing sort names that produce subconstructions. In fact, this is comparable to a subroutine call that makes a program less complex and more self-documenting. According to [14], modularity of code is a very important factor for comprehension. A recognized measure for complexity is Tom McCabe's cyclomatic complexity [26]. We implemented this measure for the metalanguage of Ericsson. We note that a high McCabe is not always a sign of bad design. A high complexity is a warning flag indicating that a BNF rule might need redesign. In the SSL language description we found about 12 complex BNF rules that need redesign in our opinion. We give a typical example.

```
single-signal-transmission-statement ::=
  'SEND' signal ['REFERENCE' field-variable]
                ['WITH' signal-datum [{',' signal-datum}]]
                [[','] < 'BUFFER'
                       | 'HURRY'
                       | 'DELAY' < numeral
                                 | field-variable
                                 | field-expression >
                                 < 'MS' | 'S' | 'M' >
                       | 'DELAY' 'UNTIL'
                         month-day-hour-minute > ] ';'
```

The cyclomatic complexity of this BNF rule is 17. Below we depicted part of a solution. We depicted rules that handle the DELAY.

```
delay          ::= 'DELAY' delay-time
delay-time     ::= amount-of-time time-units
delay-time     ::= 'UNTIL' absolute-time
amount-of-time ::= < numeral
                   | field-variable
                   | field-expression >
time-units     ::= < 'MS' | 'S' | 'M' >
absolute-time  ::= ...
...
```

We turned both the DELAY alternatives into one sort delay. This means that the original SBNF rule will become more simple at that point. We can do the same for other parts of the rule. We have tools that can restructure BNF rules automatically, and that reduce the complexity per grammar rule. Of course they cannot define descriptive names automatically, so we have to take care of that using a table that converts the generated names into more descriptive ones. We note that if the rules do not have a high complexity, the sort names that we generate from the original sort names are acceptable. As soon as the rules become very complex, the generated sort names will become complex as well. We discuss those tools in section 5 where we explain the assembly line that takes care of an automated transformation from SBNF to SDF.

Other Measures If we inspect the sort names with a high McCabe, we sometimes find rules that are not complex at all. Case statements are an example: `a ::= <b|...|z>`. Therefore, McCabe is not in all cases an optimal choice. The just mentioned rule, for instance, has a very low nesting level. We implemented a tool that reports the nesting level per nonterminal so that we can correct on the false-positive McCabe's. We also implemented a measure that counts the number of decision point in a BNF rule. This is in fact a combination of McCabe and the Nesting degree. This measure gives us the highest correlation: a rule with many decision points is complex and vice versa. Using a simple `awk` [1] one-liner we transform the data that we list with our tools into formatted reports.

4 Language Description Reengineering

In the previous section we discussed many useful and simple tools that give an impression of the current state of a language description and we take care of the fact that it parses. This is not a guarantee that it does not contain errors anymore. In fact, some errors cannot easily be found using the technology that we explained so far. They can be revealed when we execute the language description, in other words with a semantical analysis. On the way to make the grammar executable we will also find errors, for instance during modularization. In theory, a language description is complete and error-free so it should be clear how to obtain a parser for the language. If you use tools like Lex and Yacc there is an effort, namely in resolving ambiguities in the language description. Since we use a more sophisticated parsing technology, we do not have to resolve such conflicts: they are resolved for us by the parser.

In practice, we have to reengineer a language description. In the case of our running example, this is clear from our quality assessment: many production rules are missing. There are at least three aspects that play a role in reengineering a language description. We can reengineer the original language description to make it complete, we can modularize it, and we can convert it to other syntax definition languages. The first aspect is obvious: a complete description is the target of a language description. The second aspect is useful for several reasons. It is useful to have related production rules grouped together. It is for our application (reengineering software written in the language) useful to have a modular grammar [7]. It is also useful to have a modular grammar since then we can generate a modular language description document from it. After all, a document is also modular: chapters, sections, and paragraphs. This modularity is inherited from the modular grammar definition. Another reason why modular grammars are useful, is that we can execute small parts of it (we will use this approach in an example where we show how to detect semantic errors). The third aspect (conversion) is useful for the semantical issues. If we convert the completed grammar to a syntax description language that is supported by a parser generator tool, we can generate a parser for the language description and then do semantical checks. We can, for instance, test the example code that is present in the manual, or real world code written in the language. In this way semantical errors can be traced. This is also interesting for standardization, since it prevents semantical errors *before* the standard is published.

remove-redundant-rules We have implemented a tool that removes the redundant production rules. Let us recall that the original number of rules is 203. To be honest, its 203 and a half. We found the (incorrect) `SBNF-Element [set-name:]parameter-name[.attribute]` in the on-line manual. We excluded this one since we had no idea what to do with it. Remains 203 rules. We removed the redundant ones so 191 are left.

Modularization We implemented a tool that groups production rules by the sort that is produced. This gives us on the one hand insight whether sorts can be defined more easily. On the other hand it gives an idea of how we can split up the language description into modules if we would convert the grammar to SDF, which supports modular syntax definition. Let us have a look at the problems that were revealed during modularization of SSL. We mention two examples. We found an error in one of the production rules for the sort `ioconnection`. The production rule for the `ioconnection` occurs 6 times in the manual (we removed those). It also occurs once in the wrong way. This was revealed during modularization when we saw all the rules that define `ioconnection` together. We depicted the erroneous production rule in the paragraph on the top and bottom sorts tools (we are curious to know whether an SSL expert reading this has detected this error when we depicted the rule earlier). It is supposed to recognize the first statement in the following code fragment that we took from the manual:

```
   RELEASE FILE,
     ID IS IOP,
     ABRANCH IS ROK,
     CODE IS IOCODE,
     POINTER IS P,
     BUFFER IS DBUF;
ROK) FREE DBUF, POINTER IS P;
```

We do not need to generate a parser for the `ioconnection` module to know that this code cannot be parsed using the earlier mentioned production rule. For, the rule expects a `BUFFER` first and then a `POINTER` and not vice versa, as is presented in the example. The rule happened to be erroneous.

Another example is that some rules are slightly different. This can be due to maintenance problems, or due to the fact that some combinations are in some cases slightly different. We give an example, of the `signal-datum` and a possible simplification. We found several rules for this sort. We give a few of them.

```
signal-datum ::=
  < field-variable | symbol-variable | pointer | numeral
  | string-object | buffer-variable | field-expression
  | '+' >
signal-datum ::= < field-variable | symbol-variable
  | pointer | numeral | string-object | buffer-variable
  | '+' >
signal-datum ::= < field-variable | string-variable
  | pointer | buffer-variable | '+' >
```

This can be simplified to one rule covering all the combinations:

```
signal-datum ::=
  < field-variable | symbol-variable | pointer
   | numeral | string-object | string-variable
   | buffer-variable | field-expression | '+' >
```

We can unify [5] the possibilities into one rule containing them all. If the various forms are only applicable in certain situations, we cannot do this. In that case, the name `signal-datum` should be changed. For instance, `reception-signal-datum` for the `signal-datum` that belongs to the `single-signal-reception-statement`, and so on. Both solutions are simple, the problem is to pick the right one. This was simply decided by feeding discriminating examples to the compiler.

BNF2SDF With the aid of the modularization tool, we identify possible modules. With another tool we convert these modules into SDF. The tool is called BNF2SDF. This tool translates a given BNF program into an SDF program. So it gives us a language description in SDF that is executable in the sense that we can generate a parser (and online documentation and a hard copy manual). The language description will then be in SDF notation, although it is possible to represent the SDF in BNF notation as well. If we convert from BNF to SDF then the SDF version of the BNF language description can be used to generate a tool factory. So for SSL this means that we can generate a tool factory in which we can construct tools for SSL. This can be quality assessment tools, development tools, and reengineering tools. The reengineering includes conversion to other languages, restructuring, and code analysis.

The first version of an executable language description will presumably not parse in the correct way. In the case of SSL this was certainly not the case. We used the first version to detect semantic errors in the language description.

Let us give an example how we traced semantic errors that were not detected during static analysis tools. We converted a module of the SSL grammar describing the `DIRECTIVE` statement. The rule for the `directive-statement` contained a semantic error that was not so easy to reveal. Let us display the relevant production rules from the manual.

```
directive-statement ::=
  'DIRECTIVE' '[' message [{',' message}] ']' ';'
message ::= identifier [{directive-parameter}]
```

We converted this SBNF fragment to SDF. The output of the tool is depicted below:

```
imports Layout
exports
  sorts
    Directive-statement Message
    Identifier Directive-parameter
  context-free syntax
    "DIRECTIVE" "\[" {Message ","}+ "\]" ";"
                                 -> Directive-statement
    Identifier Directive-parameter* -> Message
```

Then we loaded this file into the ASF+SDF Meta-Environment as a syntax module. Since we have bottom sorts `Identifier` and `Directive-parameter`, we added some lexical syntax by hand:

```
imports Layout
exports
  sorts
    Directive-statement Message
    Identifier Directive-parameter
  lexical syntax
    "\"" ~[\"]* "\"" -> Identifier
    "\"" ~[\"]* "\"" -> Directive-parameter
  context-free syntax
    "DIRECTIVE" "\[" {Message ","}+ "\]" ";"
                                 -> Directive-statement
    Identifier Directive-parameter* -> Message
```

We tried to parse some code. In this example case we parsed a code fragment containing a single `DIRECTIVE` statement. In the manual we found the following statement:

```
DIRECTIVE "CBOPT" "ON";
```

It did not parse. The error message indicated that it expected a `[` right after the keyword `DIRECTIVE`. True: there is a keyword `[` in the original BNF rule. The quotes around the `[` and `]` should have been left out. Then the construction becomes a list of zero or more messages separated by a comma, which makes sense.

We repaired the SBNF rule and generated the SDF module again. Thus we obtained (after adding the missing lexical definitions):

```
imports Layout
exports
  sorts
    Directive-statement Message
    Identifier Directive-parameter
  lexical syntax
    "\"" ~[\"]* "\"" -> Identifier
    "\"" ~[\"]* "\"" -> Directive-parameter
  context-free syntax
    "DIRECTIVE" {Message ","}* ";"  -> Directive-statement
    Identifier Directive-parameter* -> Message
```

Using this module we could parse the above `DIRECTIVE` statement without errors. In the way we illustrated above, it is possible to obtain a modular grammar for SSL that can parse all the SSL code

Remark Some of the errors that we detected in earlier analyses, were in fact semantic errors. The reason that they were found earlier is that there was something special about them. We mention the erroneous `ioconnection`: since there was much redundancy we were alarmed that something could be wrong. Indeed during modularization, we found the two versions, and inspection of the examples revealed the error. If there had only been one version, we would perhaps not have been triggered that something could be wrong. But then a semantic analysis like the one above would have revealed the error.

5 An Assembly line for BNF2SDF

In this section we discuss the assembly line that we developed for the conversion of BNF programs to SDF programs. In fact, this is a language conversion. The conversion consists of several parts. In general, when converting from one language to another there are a few situations that we can encounter. There are constructs that are native in both languages. Those conversions are the most simple ones. Then there are constructs that are native in the original language but not available in the other. This means that those have to be simulated in the converted language. We also have the other possibility: constructions that are simulated in the original language that are available in the converted one. The latter is often an incentive to do a conversion project. Then we have the cases that a natural construct is simulated in the original language and also needs to be simulated in the target language. And we have the situation that a construct is not at all expressible in the target language or vice versa. For more information on language conversions we refer the reader to [32].

We discuss the assembly line. It contains some preprocessing operations, a main operation, and some postprocessing operations. This approach is also applied in [6] and [30]. All the operations are components. We glue these components together with a coordination language called SEAL [23]. SEAL stands for Semantics-directed Environment Adaptation Language; it not only takes care of the coordination but also of a graphical user interface [24].

Lists A list can be represented in SBNF as `s [{s}]` standing for one or more occurrences of sort `s`. In SBNF, this can also be represented as `{s}`. A simpleminded conversion would not see that the `s [{s}]` is in fact `{s}` and turn it into an unnatural SDF construct. We implemented a transformation that preprocesses SBNF and turns the simulated version into the native version. It is not possible to represent zero or more occurrences natively in SBNF. It is simulated as `[{s}]`. We translate such occurrences to an intermediate BNF formalism that we called IBNF. It contains the new notation `(s)` standing for zero or more occurrences of sort `s`. These transformations make converting to SDF a lot easier. Moreover, they contribute to as native as possible SDF, since SDF has native notation for both types of lists. We note that the transformations use pattern recognition. We recognize these patterns using our generated native pattern language for SBNF [30].

Separated lists In SBNF, lists with a separator can be simulated in various ways. We mention four possible ways `s [{',' s}]` or its equivalent `[{s ','}] s`, for one or more occurrences of sort `s` with a comma as separator; `[s [{',' s}]` or its equivalent `[[s ','}] s]` for the zero or more variant. We converted the first two to `{ s }_','` and the second two to `(s)_','`. The latter constructs are available in our intermediate IBNF language. We note that SDF contains an explicit notation for separated lists, as well. So conversion from SBNF via IBNF to SDF is a logical route.

Elimination of optionals, choices and complex list-items In this phase we simulate constructions that are not available in SDF. There are no optionals like `[s]` in SDF, no choices `<s|t>`, and no complex list items `{s t}`. The reason for that is that such constructions are not context-free functions in the strict sense. For more details on the design of SDF we refer to [16]. We note that this elimination process reduces the complexity of SBNF rules. Our task is to eliminate those constructs and change them into native SDF. We convert a rule `a ::= [b]` to the rules `a ::= opt-b`, `opt-b ::= b`, and `opt-b ::= `. We transform `a ::= <b|c>` into the rules `a ::= b` and `a ::= c`. We transform complex list items into one sort: `a ::= {b c}` into `a ::= {b-c}` and `b-c ::= b c`. Some of the generated sorts needed a more descriptive name. That required a little hand work.

Main operation: syntax swap Now that we have pretreated the SBNF program the actual transformation to SDF like syntax is easy. Since SDF contains context-free functions, the rules are denoted the other way around. Furthermore, the main operation takes care of the fact that SDF nonterminals start with an upper case. Since the SBNF program is pretreated the main operation is simple and smooth. Still, the code that we obtain is not complete SDF. We needed to postprocess the code a little.

Addition of Sorts In SDF, we have to declare the sorts that we wish to use. In SBNF this is nonexistent. Therefore, we postprocess the code and add the sort names. This operation is also useful when writing an SDF grammar by hand. If a grammar is large, then it is hard to remember whether the produced sorts are defined already or not. So in our development environment for tool factories, we use this tool already for the support of language description development. We could reuse this component without a single change. For more information on reuse and our general approach towards (evolutionary) software engineering we refer to [20].

6 Conclusions

In this paper we proposed an approach to develop, assess and reengineer language description documents. The described methods and tools can be used to develop language description documents that are correct in the sense that the grammar of the language does not contain errors in the description, and that the examples in the document can be parsed by a parser that can be generated from the language description. From the language description it is possible to generate typeset documents, or on-line documents (using already existing technology). This is particularly useful for the development of standards. We applied our tools to a real-world example: a proprietary language from Ericsson. Our tools generated useful listings containing the information that is necessary to correct the errors. We also developed a method to test the definition in order to obtain a correct language description.

After finishing this paper, we used this technology to generate a complete SSL grammar from the source code of the compiler. We used that grammar to generate a software renovation factory See [31] for details. Moreover we used an extension of the technology described in this paper to generate a correct VS COBOL II grammar from its corresponding IBM Manual (see [25] for a URL). Correct means here that our generated parser could parse 800.000+ lines of VS COBOL II code without problems. Since many reengineering activities start with a grammar in good shape, the methods and tools that we discussed in this paper are extremely helpful for us to accomplish a cost-effective approach towards reengineering. It is, for instance, possible to generate correct grammars for SQL, CICS, PL/I, and so on from IBM manuals. This means that it is viable for both obscure and well-known languages to recover their grammars, and as such to speed up tool development for software renovation. So, the results in this paper are of significant economic importance.

References

[1] A. Aho, B.W. Kernighan, and P.J. Weinberger. *The AWK Programming Language.* Addison-Wesley, 1988.

[2] J.W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In S. de Picciotto, editor, *Proceedings of the International Conference on Information Processing*, pages 125—131. Unesco, Paris, 1960.

[3] 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.

[4] 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. Available at http://adam.wins.uva.nl/~x/trans/trans.html.

[5] 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*, electronic Workshops in Computing. Springer verlag, 1997. Available at http://adam.wins.uva.nl/~x/coboldef/coboldef.html.

[6] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Control flow normalization for COBOL/CICS legacy systems. In *proceedings of the second euromicro conference on maintenance and reengineering*, 1998. To appear. Available at http://adam.wins.uva.nl/~x/cfn/cfn.html.

[7] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117, 1998. Available at http://adam.wins.uva.nl/~x/ref/ref.html.

[8] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 1999. To appear. Available at http://adam.wins.uva.nl/~x/scp/scp.html. An extended abstract with the same title appeared earlier: [4].

[9] 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.

[10] A. van Deursen. *Executable Language Definitions - Case Studies and Origin Tracking Techniques.* PhD thesis, University of Amsterdam, 1994.

[11] T.B. Dinesh, M. Haveraaen, and J. Heering. A domain specific programming style for numerical software, 1998. Work in progress.

[12] N. Faltin. `extract.perl`, 1996. Available at: http://www7.informatik.uni-erlangen.de/~nsfaltin/mscbnf/extract.perl.

[13] J. Fokker. Functional parsers. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 1–23. Springer-Verlag, 1995.

[14] R.L. Glass and R.A. Noiseux. *Software Maintenance Guidebook.* Prentice-Hall, 1981.

[15] M. van der Graaf. A specification of Box to HTML in ASF+SDF. Technical Report P9720, University of Amsterdam, Programming Research Group, 1997.

[16] 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.

[17] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990.

[18] International Telecommunication Union. *Recommendation Z.120 (10/96) - Message sequence chart (MSC)*, 1996.

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

[20] P. Klint and C. Verhoef. Evolutionary software engineering: A component-based approach. In R.N. Horspool, editor, *IFIP WG 2.4 Working Conference: Systems Implementation 2000: Languages, Methods and Tools*, pages 1–18. Chapman & Hall, 1998. Available at: http://adam.wins.uva.nl/~x/evol-se/evol-se.html.

[21] D.E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[22] J.W.C. Koorn. GSE: A generic text and structure editor. In J.L.G. Diets, editor, *Computing Science in the Netherlands (CSN92)*, SION, pages 168–177, 1992.

[23] J.W.C. Koorn. Connecting semantic tools to a syntax-directed user-interface. In H.A. Wijshoff, editor, *Computing Science in the Netherlands (CSN93)*, SION, pages 217–228, 1993.

[24] J.W.C. Koorn. *Generating uniform user-interfaces for interactive programming environments.* PhD thesis, University of Amsterdam, 1994.

[25] R. Lämmel and C. Verhoef. *VS COBOL II grammar V1*, v1 edition, 1999. Available at http://adam.wins.uva.nl/~x/grammars/vs-cobol-ii/.

[26] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-12(3):308–320, 1976.

[27] P.D. Mosses. *Action Semantics.* Cambridge University Press, 1992.

[28] G.C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, 1996.

[29] P.W. Oman and C.R. Cook. The book paradigm for improved maintenance. *IEEE Software*, 7(1):39–45, 1990.

[30] M.P.A. Sellink and C. Verhoef. Native patterns. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Society, 1998. Available at http://adam.wins.uva.nl/~x/npl/npl.html.

[31] M.P.A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 245–255, 1999. Available via http://adam.wins.uva.nl/~x/com/com.html.

[32] A.A. Terekhov and C. Verhoef. The realities of language conversions, 1999. Available at http://adam.wins.uva.nl/~x/con/con.html.