

# Development, Assessment, and Reengineering of Language Descriptions

## —Extended Abstract—

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. Language description is understood as the syntactical description of a language here. We develop assessment tools that 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. 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 [1] 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 [22]. It is also possible to generate full documentation for the language [6]. A sophisticated parser can be generated from the grammar [14]. A structured editor can be generated from the grammar [18]. In [3] we generate components for software renovation factories. 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 [16] is such an environment. We use it for the generation of tool factories [3]. SDF stands for Syntax Definition Formalism [13], it can be used to define the syntax of a language. ASF stands for Algebraic Specification Formalism [2], 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. 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 [21] 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 [4]. 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. The language description document (an ITU standard) of so-called Message Sequence Charts [15], is an MS-Word document. In order to extract the BNF syntax, the PostScript version of the Word document is first converted to ASCII, then using a script called `extract.perl` [9], the (nonlexical) BNF rules are extracted. Then, fourteen manual corrections are needed (see the comments in the script [9]). Then the BNF rules are fed to another script that generates an HTML document so that the BNF rules can be browsed.

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 [6] we generate a  $\LaTeX$  document and produce hard copy. Using technology presented in [12], 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. So maybe its an idea to use our strategy for producing a language standard.

**Organization** In Section 2 we discuss syntax definition languages and their syntax and semantics as well as parsing of language description documents. In Section 3 we describe some quality assessment tools that give an insight in the current state of language descriptions. Then, in Section 4 we briefly discuss reengineering of grammars. In Section 5, we give conclusions and future work.

**Related Work** In [7] the ASF+SDF Meta-Environment was used to check the formal parts of a book on action semantics [20]. In [7] 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 [10] 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 referred syntax definition language SDF in order to reveal semantic errors in language descriptions. In [8] 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.

**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.

## 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) [13] 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* [17]. 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. These errors can be divided into two different classes discussed in the next subsection.

### 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 [7] where errors in a book on action semantics [20] 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 or faulty recognize the example programs that are contained in the document. 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 have no idea. If the methodology described in this paper is used to develop standards for languages, such errors are revealed before the standard is published.

## 2.2 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 [14]. Let us depict the SDF module containing the grammar of SBNF:

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

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`. Then we enter the `context-free syntax` paragraph. 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`. Then we implement the next three conventions. If we have zero or more `SBNF-Element`'s 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 lists of zero or more `SBNF-Element`'s separated by the symbol | and 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 by zero or more `SBNF-Element`'s. Finally, one or more `SBNF-Rule`'s 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.

## 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. To give the reader a flavor of the nature of such tools we discuss three of the tools that we have implemented.

**Top and Bottom Sorts** We implemented two tools that report the number of top sorts (sorts that are defined but not used) and the number of bottom sorts (sorts that are used but not defined). In case of the SSL<sup>1</sup> 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 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.

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.

**Rule Complexity** Tools like the ones discussed above 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 [11], modularity of code is a very important factor for comprehension. A recognized measure for complexity is Tom McCabe's cyclomatic complexity [19]. 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.

## 4 Language Description Reengineering

In the previous section we discussed some useful tools to detect syntax errors in a language description. As we ex-

<sup>1</sup>Switching System Language, a language description we accessed for Ericsson upon request.

plained earlier, there can also be semantic errors in the language description. These semantic errors can be revealed when we execute the language description. In practice however, we need to reengineer the language description before we can execute it. 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 (modularization) not only improves the quality of the documentation in case it is generated from the grammar, it also helps when the grammar is used for reengineering as explained in [5]. 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.

## 5 Conclusions and Future Work

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. Future work with respect to this proprietary language depends on input by Ericsson. This includes sending us SSL code, a compiler, an emulator, and more documentation.

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.

## References

- [1] 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.
- [2] 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.
- [3] 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>.
- [4] 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>.
- [5] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the 6th International Workshop on Program Comprehension, IWPC'98, Ischia, Italy*, pages 108–117, 1998.
- [6] 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.
- [7] A. van Deursen. *Executable Language Definitions - Case Studies and Origin Tracking Techniques*. PhD thesis, University of Amsterdam, 1994.
- [8] T.B. Dinesh, M. Haverlaan, and J. Heering. A domain specific programming style for numerical software, 1998. Work in progress.
- [9] N. Faltin. `extract.perl`, 1996. Available at: <http://www7.informatik.uni-erlangen.de/~nsfaltin/mscbnf/extract.perl>.
- [10] J. Fokker. Functional parsers. In J. Jeurling and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 1–23. Springer-Verlag, 1995.
- [11] R.L. Glass and R.A. Noiseux. *Software Maintenance Guidebook*. Prentice-Hall, 1981.
- [12] M. van der Graaf. A specification of Box to HTML in ASF+SDF. Technical Report P9720, University of Amsterdam, Programming Research Group, 1997.
- [13] 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.
- [14] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990.
- [15] International Telecommunication Union. *Recommendation Z.120 (10/96) - Message sequence chart (MSC)*, 1996.
- [16] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.
- [17] D.E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
- [18] 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.
- [19] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-12(3):308–320, 1976.
- [20] P.D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
- [21] G.C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, 1996.
- [22] M.P.A. Sellink and C. Verhoef. Native patterns. Technical Report P9804, University of Amsterdam, 1998. This paper will be presented at WCRE'98. Available at <http://adam.wins.uva.nl/~x/pat/pat.html>.