Metric-Based Analysis of Context-Free Grammars

James F. Power
Department of Computer Science
National University of Ireland, Maynooth
Kildare, Ireland
James.Power@may.ie

Brian A. Malloy*
Department of Computer Science
Clemson University
Clemson, SC 29634
malloy@cs.clemson.edu

Abstract

Recent advances in software engineering have produced a variety of well established approaches, formalisms and techniques to facilitate the construction of large-scale applications. Developers interested in the construction of robust, extensible software that is easy to maintain should expect to deploy a range of these techniques, as appropriate to the task. In this paper, we provide a foundation for the application of established software metrics to the measurement of context-free grammars. The usual application of software metrics is to program code; we provide a mapping that allows these metrics to be applied to grammars. This allows us to interpret six software engineering metrics in a grammatical context, including McCabe's complexity metric and Fenton's impurity metric. We have designed and implemented a tool to automatically compute the six metrics; as a case study, we use these six metrics to measure some of the properties of grammars for the Oberon, ISO C, ISO C++ and Java programming languages. We believe that the techniques that we have developed can be applied to estimating the difficulty of designing, implementing, testing and maintaining parsers for large grammars.

1 Introduction

Recent advances in software engineering have produced a variety of well established approaches, formalisms and techniques to facilitate the construction of large-scale applications. Developers interested in the construction of robust, extensible software that is easy to maintain should expect to deploy a range of these techniques, as appropriate to the task. This range of techniques covers a spectrum of program representations ranging from abstract models to concrete models, and ultimately to the program itself. One of the more abstract models of a program is a set of software metrics that seek to describe quantitative aspects, rather than the requirements or operation of the application.

Conventional parser implementation, as represented by yacc[9], is a prime example of some aspects of software engineering in that the code for a parser can be generated automatically from a corresponding specification, expressed as a context-free grammar. Despite this, parser construction has not benefitted from the application of the full range of software engineering techniques. To date, much of the research on parser design has been theoretical in nature. However, the construction of parsers for large programming languages is both a software engineering problem and a theoretical problem.

In this paper, we provide a foundation for the application of established software metrics to the measurement of context-free grammars. The usual application of software metrics is to program code; we provide a mapping that allows these metrics to be applied to grammars. This allows us to interpret six software engineering metrics in a grammatical context, including McCabe's complexity metric and Fenton's impurity metric[10, 4]. We have designed and implemented a tool to automatically compute the six metrics; as a case study, we use these six metrics to measure some of the properties of grammars for the Oberon, ISO C,

*This work was funded, in part, by an Enterprise Ireland International Collaboration Grant.

Accepted for the 8th IEEE International Workshop on Program Comprehension
Limerick, Ireland, 10-11 June, 2000

ISO C++ and Java programming languages[11, 7, 8, 6]. We believe that the techniques that we have developed can be applied to estimating the difficulty of designing, implementing, testing and maintaining parsers for large grammars.

The remainder of this paper is organized as follows. In the next section we introduce grammars and the associated terminology, and we describe the mapping from programs to grammars. In Section 3 we define the six metrics used in this paper and show how they can be applied to grammars. Section 4 describes the design and implementation of the tool that we used to automatically compute these metrics. In Section 5 we present the results of applying the metrics construction tool to grammars for Oberon, C, C++ and Java, and we analyze the metrics. Finally, in Section 6, we draw conclusions.

2 Grammars as Programs

In this section we define some of the terminology associated with context-free grammars, and we describe the mapping from programs to grammars that forms the basis of our approach. A general description of languages, context-free grammars and parsing can be found in reference [1]; the definitions of successor relation and grammatical levels are found in reference [3].

2.1 Terminology

Given a set of words (known as a lexicon), a *language* is a set of valid sequences of these words. A *grammar* defines a language; any language can be defined by a number of different grammars. When describing formal languages such as programming languages, we typically use a grammar to describe the *syntax* of that language; other aspects, such as the semantics of the language typically cannot be described by context-free grammars. Extended Backus-Naur Form (EBNF) is a commonly-used notational enrichment of context-free grammars which does not enhance their descriptional power.

Formally a grammar is a four-tuple (N, T, S, P) where N and T are disjoint sets of symbols known as non-terminal and terminals respectively, S is a distinguished element of N known as the start symbol, and P is a relation between elements of N and the union and concatenation of symbols from $(N \cup T)$, known as the production rules. A grammar defines a language by specifying valid sequences of derivation steps which produce sequences of terminals form the sentences of the language.

The procedure of using a grammar to derive a sentence in its language is as follows. We begin with the start symbol S and apply the production rules, interpreted as left-right rewriting rules, in some sequence until only non-terminals remain. This process defines a tree whose root is the start symbol, whose nodes are non-terminals and whose leaves are terminals. The children of any node in the tree correspond precisely to those symbols on the right-hand-side of a production rule. This tree is known as a *parse tree*; the process by which it is produced is known as *parsing*.

If there is a production rule of the form $A \to \beta$ we say that non-terminal A derives β. If the sequence of symbols β contains some non-terminal B we say that B is an immediate *successor* of A, and write $A \rhd B$.

If we can subsequently apply production rules to β to produce some other set of symbols γ we write $A \to^* \gamma$ - this is the reflexive transitive closure of the derivation relation. If the sequence of symbols γ contains some non-terminal C we say that C is a *successor* of A, and write $A \rhd^* C$.

The successor relation induces an equivalence relation on the non-terminals, where we say that A is equivalent to C if $A \rhd^* C$ and $C \rhd^* A$, and we write $A \equiv C$. Any equivalence relation on a set partitions that set into a collection of equivalence classes, and in the case of grammar non-terminals, these classes are known as *grammatical levels*.

Accepted for the 8th IEEE International Workshop on Program Comprehension
Limerick, Ireland, 10-11 June, 2000

2.2 From Programs to Grammars

Since any grammar defines a language and provides a basis for deriving elements of that language, a grammar may be considered as both a specification and a program; indeed, this duality is often exploited in the construction of recursive descent parsers[1].

Conceptually, we may think of any program as consisting of a set of procedures, where each procedure is defined by some procedure body, constructed using the control primitives of the language. Thus a procedure body may be represented as a graph whose nodes are statements and whose edges represent the flow of control between these statements. At a higher level of abstraction, we may represent the interaction between procedures by a *call graph*, whose nodes are procedures and whose edges represent a call from one procedure to another.

In order to interpret the concepts of control-flow graph and call graph for context-free grammars we proceed as follows. The procedures correspond to non-terminals, and procedure bodies are the right-hand-sides of the production rules. The control primitives are the union and concatenation operations of context free grammars, which correspond to alternation and sequencing respectively. This mapping can be extended in a straightforward manner to the closure and option operators used in EBNF. In line with this mapping we interpret the call graph of a program as the graph of the successor relation between non-terminals.

In the following section we use the mapping described above as a basis for the application of some standard software metrics to context-free grammars.

3 Metrics

In this section, we first define the six metrics that we apply in Section 5 and we then describe their application to context-free grammars. The first four metrics are adaptations of standard metrics for programs and procedures[4, 10]. The final two metrics are derived from the grammatical levels described in Section 2 and these metrics were originally used to measure descriptional complexity of context-free grammars[2, 3].

- **Number of Non-Terminals**
  One of the simplest, course-grained metrics that can be applied to a program to measure its size is a count of the number of procedures that appear in that program. The equivalent size metric for context-free grammars is the number of non-terminals in that grammar[2, 3]. This size metric is commonly reported by parser generators such as yacc and bison.

- **McCabe Complexity**
  McCabe's metric measures the number of linearly independent paths through a flow graph[10]. This metric is typically interpreted as a measure of the number of decisions in the flow graph and is a useful indicator of the level of difficulty to test the measured procedure. Decisions in a context-free grammar are represented by the union operator. Two grammars with the same number of non-terminals can still differ in essential complexity if one grammar has significantly more alternatives for its non-terminals than the other grammar. The sum of the McCabe complexity measure for these alternatives will highlight this difference. Thus, our mapping of McCabe's complexity to grammars is to count the total number of alternatives in that grammar as represented by occurrences of the union, closure and option operators, since each of these represents exactly one decision.

- **Average Size**
  In a procedure, the size metric is the number of nodes in the corresponding flow graph, and is used as a formal alternative to the common lines-of-code (loc) measure. Since production rules correspond to procedures, the nodes in a flow graph correspond to terminals or non-terminals

Accepted for the 8th IEEE International Workshop on Program Comprehension
Limerick, Ireland, 10-11 June, 2000

on the right-hand-side of a production rule. To compute the average size, we calculate the total of the sizes, and divide by the number of non-terminals. This then gives us a measure of the average "length" of the production rules.

- **Fenton's Impurity**
  The call graph for a program is a directed graph indicating the dependencies between procedures in the program[4]. Many edges in a call graph indicate a large number of dependencies between procedures and this complexity complicates the testing process and can possibly indicate poor design. Since we regard a non-terminal as a procedure, and since the successor relation between non-terminals defines edges in the call graph, this metric can be applied directly to grammars. At a minimum, the call graph will be a tree, at a maximum it will be a fully connected graph; hence, to calculate the impurity metric we normalize the count of the number of edges between these bounds. The formula to compute the impurity metric for a call graph with n nodes and e edges is:
  $$\frac{2(e-n+1)}{(n-1)(n-2)}$$

- **Levels**
  In this metric, we use the call graph, used in the calculation of Fenton's impurity metric, to partition the non-terminals into a set of equivalence classes called grammatical levels. Since each of these grammatical levels forms a complete graph, we may assume a high degree of interdependence between the non-terminals in a given level. The level metric counts the number of such levels and this is a starting point for a study of the distribution of non-terminals among these levels.

- **Depth**
  The depth metric for a grammar measures the number of non-terminals in the largest grammatical level. If the depth value constitutes a significant proportion of the total number of non-terminals, then this value indicates an uneven distribution of the non-terminals among these levels.

Each of the above metrics can be calculated automatically from a context-free grammar and a tool to accomplish this is described in the next section.

4 The Implementation

In this section we present an overview of the implementation of a tool that, given a grammar, will automatically compute those metrics described in the previous section. The structure of the tool is illustrated in Figure 1. The rectangle illustrated in the upper left corner of the figure represents the input to our tool, an EBNF description of the syntax of a programming language. The output from the program is illustrated on the right side of Figure 1 and consists of text files listing the results for the computed metrics. The output is generated at two stages during execution of the tool, first producing size and complexity metrics and then producing structure metrics. The tool was written in C++ and implemented using GNU flex version 2.5.4, GNU bison version 1.28, and the GNU C++ compiler version 2.91.

The input grammar is described using a superset of the yacc syntax extended to include the full set of EBNF operators. The input grammar is scanned and parsed and an abstract syntax tree (AST) representing the production rules is then generated. The AST is constructed to correspond to the *node hierarchy* of the *Visitor pattern* as described in reference [5]. Each of the size and complexity metrics has its own visitor that walks the AST collecting the information required by that metric. This is represented in the top row of Figure 1 by the oval containing "Visitor Instance".

The vertical path depicted in Figure 1 represents

Accepted for the 8th IEEE International Workshop on Program Comprehension
Limerick, Ireland, 10-11 June, 2000

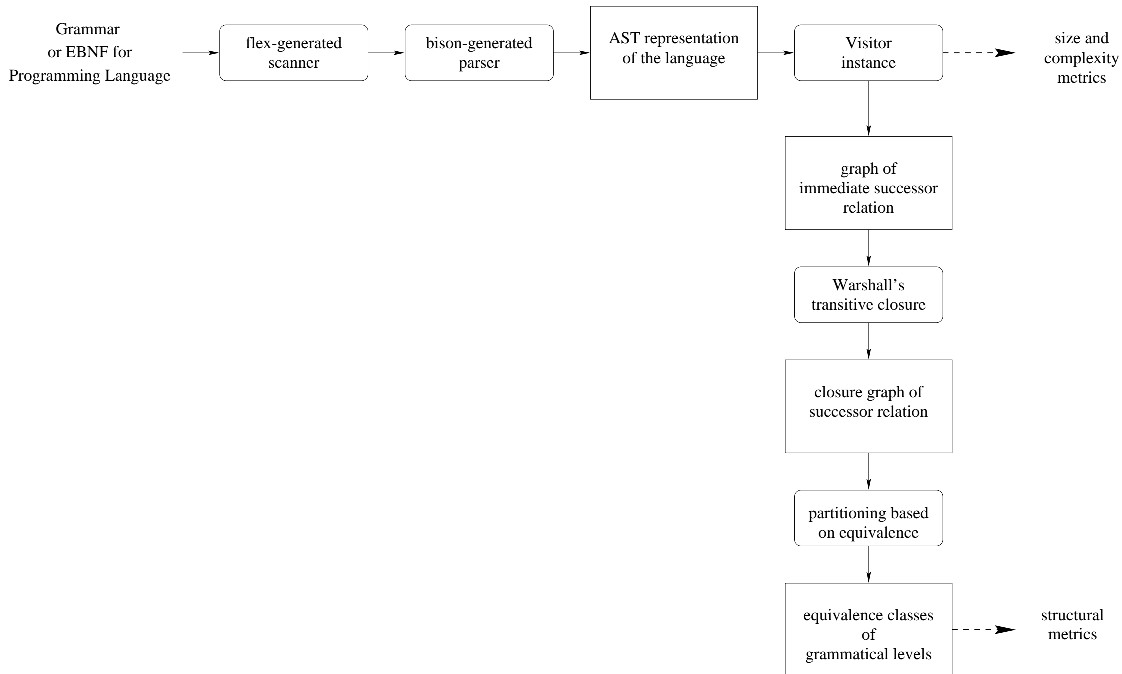

Figure 1: *Implementation overview.* This figure provides an overview of the tool that we constructed to compute the metrics for the four grammars. Input to the tool, indicated in the upper left corner, is the EBNF for the grammar under consideration. Output of the tool is the results of the computed metrics that measure the *size and complexity* of the grammar and *structure* of the grammar; the output is shown on the right side of the figure.

the construction of those data structures required to compute the grammatical levels, as described in Section 3. This process uses three main data structures: a graph of the immediate successor relation, a graph of the successor relation, and a graph of equivalence classes representing the grammatical levels. From these graphs we compute the structural metrics as shown at the bottom right of Figure 1.

The immediate successor relation is represented as a two-dimensional matrix, S, indexed by non-terminals, where for any non-terminals A and B, the entry $S[A, B]$ is true precisely when $A \triangleright B$. We next apply Warshall's transitive closure algorithm to this graph to produce a matrix S^*, where now $S^*[A, B]$ is true precisely when $A \triangleright^* B$. Fenton's impurity metric can be calculated directly by counting the number of nodes and edges in S, since this graph represents the successor relation.

The final data structure required by our tool represents the graph of grammatical levels. Each grammatical level consists of a set of non-terminals where two non-terminals A and B are in the same grammatical level when $A \equiv B$, i.e., when both $S^*[A, B]$ and $S^*[B, A]$ are true. The remaining structure metrics, Levels and Depth, are generated directly from this graph of grammatical levels.

5 Application to Four Programming Languages

In this section, we describe the results of our experiments using the implementation described in the previous section. Our experiments were conducted on grammars for four well-known programming languages: Oberon, ISO C, ISO C++ and Java[11, 7, 8, 6]. In the cases of Oberon and C, the grammar used was a bison-compatible adaptation of the grammar avail-

Accepted for the 8th IEEE International Workshop on Program Comprehension
Limerick, Ireland, 10-11 June, 2000

Size/Complexity Metric	Oberon	ISO C	ISO C++	Java
Number of Non-Terminals	48	64	141	149
McCabe Complexity	87	149	368	213
Average Size	6.8	7.6	6.1	4.1

Table 1: *Grammar size and complexity.* The results depicted in this table show the results of applying the metrics to measure the overall size and complexity of the four grammars. The three computed metrics include the Number of Non-Terminals metric, the McCabe Complexity metric and the Average Size metric.

able in the standards documents. In the cases of C++ and Java, the grammar used was taken directly from the standards document. In each case, the grammars describe the syntactic structure of the language and no semantic attributes or actions are included in any of the grammar. In the first section, we overview our results for those metrics that describe the size and complexity of the grammar. In the second section, we overview our results for those metrics that describe the structure of the grammar, in particular, those metrics derived from the grammatical levels of the grammar.

5.1 Grammar Size and Complexity

Table 1 presents the results of using three metrics that measure the overall size and complexity of the four grammars. The first row of the table lists the grammars and the first column of the table lists the applied metrics. The second row of the table, our first chosen metric, lists the number of non-terminals in each grammar, Number of Non-Terminals. Oberon and C contain approximately the same number of non-terminals, 48 and 64, whereas C++ and Java contain significantly more non-terminals, 141 and 149 respectively. These numbers provide our first indication of the relative sizes of the four grammars and correspond to the VAR metric described in reference [3].

The third row of Table 1 lists the McCabe Complexity of each of the grammars. Since McCabe's complexity measures the total number of decisions in the grammar, this third row taken in conjunction with the second row provides a more accurate picture of the overall grammar size. The McCabe Complexity numbers for Oberon, C, C++ and Java are 87, 149, 368 and 213 respectively. These numbers reinforce and accentuate the ordering implied by the number of non-terminals metric. In particular, there is now a clear distinction in the complexity of our grammars for C over Oberon, and C++ over Java. For example, the number of non-terminals in the C++ and Java grammars are quite similar, but the increased complexity of C++ over Java is reflected by the McCabe metrics for the two grammars: 368 versus 213.

The final row of Table 1 lists the Average Size metrics for the four grammars: 6.8, 7.6, 6.1 and 4.1 respectively. The similarity of these numbers reflect the notion that grammar writers typically do not allow the right hand side of rules, on average, to grow to extreme lengths. This breaking-up of overly-long rules parallels the decision by programmers not to allow procedures to grow to extreme lengths.

5.2 Grammar Structure

Table 2 presents the results of using three metrics that measure the overall structure of the four grammars. The first row of the table lists the four grammars and the first column lists the structural metrics applied. The first metric, Fenton's Impurity metric, is derived from the closure of the call graph generated by the grammar, whereas the remaining metrics, Levels and Depth are derived from the calculated grammatical levels.

The metric presented in the second row of Table 2

Accepted for the 8th IEEE International Workshop on Program Comprehension
Limerick, Ireland, 10-11 June, 2000

Structural Metric	Oberon	ISO C	ISO C++	Java
Fenton's Impurity	31.6	65.3	85.8	32.7
Levels	26	21	21	89
Depth	10/48	38/64	121/141	33/149

Table 2: *Grammar Structure.* The results depicted in this table show the results of applying the metrics to measure the structure of the four grammars. The three computed metrics include the Fenton Impurity metric, the Levels metric and the Depth metric. Fenton's Impurity metric is derived from the closure of the call graph generated by the grammar. To compute the Levels and Depth metric, we partition the non-terminals in a grammar into equivalence classes called *grammatical levels.* The Levels and Depth metric are computed using the grammatical levels for each grammar.

---

presents the results for the impurity or lack of "treeness" of the grammars. There is a definite similarity between Oberon and Java, as reflected by their impurity values of 31.6% and 32.7% respectively. On the other hand, as we move from Oberon and Java to C and then to C++, there is a definite progression in impurity. In particular, the impurity numbers for C and C++, at 65.3% and 85.8% respectively, reflect a considerable density of edges in the closure of the call graph of these grammars. One possible consequence of high values for impurity is a decreased potential for modular construction of parsers from the grammar.

The metric presented in the third row of Table 2 presents the results for the Levels metric, which measures the number of grammatical levels in the grammars. Since each grammatical level is formed from equivalent non-terminals, it is reasonable to expect a link between these values and the number of non-terminals in the grammar. This link is reflected in the Level values for Oberon, C and Java at 26, 21 and 89 grammatical levels respectively. However, the value of 21 grammatical levels for C++ is at variance with this pattern and its low value is a consequence of the fact that one of the grammatical levels contained a high proportion of the non-terminals, as described by the depth metric presented in the fourth row of the table.

The metric presented in the fourth and final row of Table 2 presents the results for the Depth metric, which measures the cardinality of the largest grammatical level in each grammar. Most of the computed grammatical levels for the four grammars were singleton sets containing just one non-terminal; only 9 grammatical levels in total for the four grammars contained more than one non-terminal.

There were three non-singleton grammatical levels generated for the Oberon grammar. The largest grammatical level for Oberon had cardinality 10 and related to the non-terminals for Oberon expressions and this is the value reported as the depth metric for this grammar. There were two other non-singleton grammatical levels for Oberon: the non-terminals relating to statements and types of cardinality 7 and 6 respectively.

We now consider the depth values for Java, since these were quite similar to those for Oberon. In particular, the largest grammatical level was that for Java expressions with a cardinality of 33 and this is the value reported as the depth metric for this grammar. There were two other non-singleton grammatical levels for Java: one for statements of cardinality 25 and one for types of cardinality 3.

There were just two non-singleton grammatical levels generated for the C grammar. The depth metric of 38 reflects the cardinality of the largest grammatical level, which contained non-terminals for expressions and declarations. The other non-singleton grammatical level, that contained non-terminals relating to statements, had a cardinality of 6.

Finally, the depth metric for C++ at 121 non-terminals

Accepted for the 8th IEEE International Workshop on Program Comprehension
Limerick, Ireland, 10-11 June, 2000

is the cardinality of the only non-singleton grammatical level for this grammar. The progression exhibited by these values is as follows. There were three categories of grammatical levels for Oberon and Java reflecting the non-terminals for expressions, statements and types. In C, two of these categories are effectively combined since the category for types is now subsumed into a more general category of expressions and declarations. This progression continues into C++ where all three categories distinguished in Oberon and Java are combined into a single large grammatical level containing 86% of the non-terminals in the grammar.

6 Concluding Remarks

In this paper, we have described an approach for applying metrics to the measurement of context-free grammars. We have described a technique to map six established metrics onto four grammars for the Oberon, C, C++ and Java programming languages[11, 7, 8, 6]. We have described our tool for automatically computing the metrics that take, as input, an EBNF for a context-free grammar and produces, as output, results for the computed metrics. Our tool uses flex and bison to parse the input grammar and uses the Visitor Pattern to gather the information for the metrics: a visitor for each of the metrics is used to walk the AST representation of the production rules, collecting the information required by the metric.

Our results using the Number of Non-Terminals metric indicate that the overall size of the grammars for C++ and Java are much larger than Oberon and C. Our results using McCabe's complexity metric indicate that the grammar for C++ is the most complex, followed by Java, then C, with Oberon the least complex of the four grammars. Fenton's Impurity metric is computed from the closure of the call graph derived from the grammar; impurity indicates a lack of "tree-ness" in the grammar and reflects a large number of dependencies between non-terminals. Our results using Fenton's impurity metric indicate that the grammar for C++ was the most "impure", followed by C, and then Java; Oberon was the most pure of the four grammars. The complete results for our metrics are described in Section 5.

We believe that the technique that we have developed can be applied to estimating the difficulty of designing, implementing, testing and maintaining parsers for large grammars.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] W. Brauer. On grammatical complexity of context-free languages. *Proceedings of Mathematical Foundations of Computer Science*, pages 193–196, 1973.

[3] Erzsébet Csuhaj-Varjú and Alica Kelemenová. Descriptional complexity of context-free grammar forms. *Theoretical Computer Science*, 112(2):277–289, 1993.

[4] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* Thomson Computer Press, first edition, 1996.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[6] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[7] ISO/IEC. *International Standard: Programming Languages - C.* Number 9899:1990(E) in ASC X3. American National Standards Institute, 1990.

[8] ISO/IEC. *International Standard: Programming Languages - C++.* Number 14882:1998(E) in ASC X3. American National Standards Institute, 1998.

[9] S. C. Johnson. Yacc – yet another compiler compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, USA, 1975.

[10] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[11] H. Mössenböck and N. Wirth. The Programming Language Oberon-2. *Structured Programming*, 12(4):179–195, 1991.

Accepted for the 8th IEEE International Workshop on Program Comprehension
Limerick, Ireland, 10-11 June, 2000