
01 **Research**

02
03 **A metrics suite for**
04 **grammar-based software**
05
06

07 James F. Power¹ and Brian A. Malloy^{2,*},[†]
08

09 ¹*Department of Computer Science, National University of Ireland, Maynooth,*
10 *Co. Kildare, Ireland*

11 ²*Department of Computer Science, Clemson University, Clemson, SC 29634, U.S.A.*
12

13
14
15 **SUMMARY**
16

17 **One approach to measuring and managing the complexity of software, as it evolves over time, is to exploit**
18 **software metrics. Metrics have been used to estimate the complexity of the maintenance effort, to facilitate**
19 **change impact analysis, and as an indicator for automatic detection of a transformation that can improve**
20 **the quality of a system. However, there has been little effort directed at applying software metrics to the**
21 **maintenance of grammar-based software applications, such as compilers, editors, program comprehension**
22 **tools and embedded systems. In this paper, we adapt the software metrics that are commonly used to**
23 **measure program complexity and apply them to the measurement of the complexity of grammar-based**
24 **software applications. Since the behaviour of a grammar-based application is typically choreographed by**
25 **the grammar rules, the measure of complexity that our metrics provide can guide maintainers in locating**
problematic areas in grammar-based applications. Copyright © 2004 John Wiley & Sons, Ltd.

26 **KEY WORDS:** program comprehension; software metrics; source code analysis; call graph; maintenance of
27 grammar-based tools
28
29

30
31 **1. INTRODUCTION**
32

33 One of the most important and costly phases in the life cycle of software applications is maintenance.
34 Estimates of the cost of maintenance range from 65% [1] to 80% [2,3] of the software budget. Much of
35 this cost derives from the burgeoning complexity of the software application as it evolves over time.
36 During the life cycle of an application, many modifications and extensions to the software are needed
37 to support evolving business requirements, increasing the complexity of the application. A program
38 that is overly complex may be difficult to comprehend and therefore more costly to maintain. Software
39 complexity may induce a significant number of operational failures that require a greater amount of
40 maintenance to correct errors introduced as a result of modifications and extensions to the code.
41
42

43 *Correspondence to: Brian Malloy, Department of Computer Science, Clemson University, Clemson, SC 29634, U.S.A.

44 [†]E-mail: malloy@cs.clemson.edu

01 One approach to measuring and managing the complexity of software, as it evolves over time,
02 is to exploit software metrics. The use of software metrics has become essential to good software
03 engineering [4]. Many developers measure characteristics of a program to determine whether the
04 requirements are consistent and complete, whether the design is of high quality and whether the code
05 is too complex to permit easy test and maintenance. Metrics have been used to estimate the complexity
06 of the maintenance effort [5–8], to facilitate change impact analysis [9], and as an indicator for
07 automatic detection of a transformation that can improve the quality of a system [10]. Recent research
08 incorporates semantic information into metric considerations [11,12].

09 In this paper we seek to extend the domain of application of software metrics to *grammar-*
10 *based* software applications, such as compilers, editors, program comprehension tools and embedded
11 systems. The traditional example of a grammar-based application is the parser module of a compiler
12 for a programming language. This module will likely include either a grammar or grammar-based
13 code together with semantic actions embedded in the grammar; usually these semantic actions are
14 themselves written in a high-level programming language [13]. Related applications include language-
15 based editors, which incorporate a grammar as part of the implementation of the integrated environment
16 [14]. Grammar-based software also forms the core of program comprehension, which involves reading
17 the source code of a software module or application, analysing the source code, and building a model of
18 that code appropriate for the task at hand. Direct applications of program comprehension include tools
19 for software visualization, reverse engineering, translation, metrication and instrumentation. Embedded
20 systems include grammar-based code to overcome hurdles in the performance of these embedded
21 computing solutions [15,16].

22 The complexity of constructing grammar-based applications is related, at least, to the syntactic
23 complexity of the language under consideration. Further, the evolution of the language, including
24 changes to its syntax, has a direct bearing on the maintainability of the corresponding grammar-based
25 tools. However, there is a paucity of research that applies metrics to the measurement and maintenance
26 of such grammar-based applications.

27 In this paper, we adapt the software metrics that are commonly used to measure program complexity
28 and apply them to the measurement of the syntactic complexity of grammars. We adapt concepts
29 about programs such as control flow and function invocation, which are used in many program
30 metrics, and apply them to grammars. This adaptation permits us to quantify some of the anecdotal
31 information about grammars found in the literature. For example, [17] describes the C++ grammar as
32 ‘complex’, and [18] describes the GNU C++ grammar as ‘complex, fragile and having a high degree
33 of modification’. By applying our grammar metrics to the ISO C++ grammar and to several versions
34 of the GNU C++ grammar, we provide results that support these anecdotal observations.

35 To demonstrate the feasibility of our approach, we present the design and implementation of a
36 tool, SYNQ[‡], that automatically computes our metrics for a given grammar. We demonstrate the
37 applicability of our work to software evolution by using SYNQ to chart the evolution of the GNU
38 C++ compiler through several versions. We selected GNU C++ as our case study because it is a
39 commonly used grammar-based application and the source code is available through the GNU public
40 license. This case study of evolution demonstrates that our metrics quantify the expected properties
41
42

43 _____
44 [‡]SYNQ is pronounced ‘sink’, the SYNTactic Quantification or measurement of grammar complexity.



01 associated with the evolution. In particular, we show that the grammar at the core of the GNU C++
02 compiler has increased in complexity, and that the metrics permit us to quantify the differences
03 between minor and major complexity increases. Since the behaviour of a grammar-based tool is
04 typically choreographed by the grammar rules, the measure of syntactic complexity that our metrics
05 provide can guide maintainers of grammar-based applications in locating problematic areas in the
06 application.

07 The remainder of this paper is organized as follows. In Section 2 we introduce grammars and the
08 associated terminology and we describe some of the previous research about grammar complexity.
09 In Section 3 we define the metrics used in this paper and show how they can be applied to grammars
10 that describe programming languages. Section 4 describes the design and implementation of SYNQ,
11 a tool that automatically computes these metrics. In Section 5 we present the results of applying the
12 metrics construction tool to four modern programming languages, and we analyse the metrics. Since
13 parser generators such as *yacc* are based on grammars, the evolution of software that contains a parser
14 often parallels the evolution of the grammar at its core. In Section 6 we use our metrics to chart the
15 evolution of the GNU C++ compiler through several versions and in Section 7 we review the work that
16 relates to syntactic grammar complexity. Finally, in Section 8, we draw conclusions.

17

18

19

20 2. BACKGROUND

21 In this section we define some of the terminology associated with context-free grammars, and we
22 describe the basic concepts associated with grammar measurement. A general description of languages,
23 context-free grammars and parsing can be found in [19]; the definitions of successor relation and
24 grammatical levels are found in [20]. The definitions in this section are somewhat formal, but we
25 feel that a degree of formality is necessary at this point in order to provide unambiguous definitions of
26 our metrics in later sections.

27

28

29 2.1. Terminology

30

31 Given a set of words (known as a lexicon), a *language* is a set of valid sequences of these words.
32 A *grammar* defines a language; any language can be defined by a number of different grammars.
33 When describing formal languages such as programming languages, we typically use a grammar to
34 describe the *syntax* of that language; other aspects, such as the semantics of the language, typically
35 cannot be described by context-free grammars.

36 Formally a grammar is a four-tuple (N, T, S, P) where N and T are disjoint sets of symbols known
37 as non-terminals and terminals, respectively, S is a distinguished element of N known as the start
38 symbol, and P is a relation between elements of N and the union and concatenation of symbols from
39 $(N \cup T)$, known as the production rules. A grammar defines a language by specifying valid sequences
40 of derivation steps that produce sequences of terminals, known as the *sentences* of the language.

41 One procedure for using a grammar to derive a sentence in its language is as follows. We begin
42 with the start symbol S and apply the production rules, interpreted as left-right rewriting rules, in some
43 sequence until only non-terminals remain. This process defines a tree whose root is the start symbol,
44 whose nodes are non-terminals and whose leaves are terminals. The children of any node in the tree

01 correspond precisely to those symbols on the right-hand side of a production rule. This tree is known
 02 as a *parse tree*; the process by which it is produced is known as *parsing*.

03 If there is a production rule of the form $A \rightarrow \beta$ we say that non-terminal A derives phrase β . If we
 04 can subsequently apply production rules to β to produce some other phrase γ we write $A \rightarrow^* \gamma$; this
 05 phrase is the reflexive transitive closure of the derivation relation.

06 While the simplest formulation of grammar rules allows only union and concatenation in their
 07 definition, it is not unusual to allow the right-hand side of a rule to be formed from regular expressions
 08 over $(N \cup T)$. Such grammars are known as regular rightpart grammars, and are equivalent, modulo a
 09 change of notation, to grammars presented in Extended Backus-Naur Form (EBNF). In what follows
 10 we will assume that grammar rightparts may arbitrarily mix union, concatenation, option and closure
 11 operations.

12 Using either regular rightpart grammars or EBNF does not increase the power of expression over
 13 context-free grammar notation. However, it considerably eases the task of presenting a grammar, and
 14 many programming language standards choose EBNF or one of its variants. It is a feature of our
 15 approach that all the metrics we present work with either the standard context-free grammar notation
 16 or with EBNF-style presentations.

18 2.2. Grammatical levels

20 In this section we present some of the concepts and notation from [20]. We also review some of the
 21 metrics presented in [20] in order to provide a context for our own metrics presented in later sections.

22 If non-terminal A derives some sequence of symbols β , and β contains some non-terminal B we say
 23 that B is an *immediate successor* of A , and write $A \triangleright B$. If β derives some sequence of symbols γ , and
 24 γ contains some non-terminal C we say that C is a *successor* of A , and write $A \triangleright^* C$.

25 The successor relation induces an equivalence relation on the non-terminals, where we say that A
 26 is equivalent to C if $A \triangleright^* C$ and $C \triangleright^* A$, and we write $A \equiv C$. Any equivalence relation on a set
 27 partitions that set into a collection of equivalence classes, and in the case of grammar non-terminals,
 28 these classes are known as *grammatical levels*. For any two non-terminals A and C in different levels,
 29 if $A \triangleright C$ then this naturally induces a corresponding ordering on their levels. For different levels L_1
 30 and L_2 , if $A \in L_1$ and $C \in L_2$, then when $A \triangleright C$ we write $L_1 \succ L_2$.

31 Based on these definitions, [20] defines the following *complexity measures* for a context-free
 32 grammar:

34 VAR—the number of non-terminals,
 35 PROD—the number of production rules,
 36 LEV—the number of grammatical levels,
 37 DEP—the number of non-terminals in the largest grammatical level,
 38 HEI—the maximum length of the chain of levels $L_0 \dots L_n$ such that each $L_i \succ L_{i+1}$ for
 39 $0 \leq i \leq n$.

41 In the remainder of this paper, we make direct use of the VAR, DEP and HEI measures. We propose
 42 seven additional metrics, three of which are modified versions of PROD and LEV, and four of which
 43 are original to this paper. We discuss the implementation and use of all of these metrics, and apply
 44 them to a range of grammars and parsers.



01 3. A METRICS SUITE FOR PROGRAMMING LANGUAGE SYNTAX

02 In this section, we define a set of ten metrics that are used in the remainder of this paper. The *size*
 03 metrics are adaptations of standard metrics for programs and procedures [4,21]. The *structural* metrics
 04 are derived from the grammatical levels described in Section 2 and these metrics were originally used
 05 to measure descriptive complexity of context-free grammars [20,22]. For each of the ten metrics we
 06 present a formal definition and discuss some of the pragmatics of their use.

07 The purpose of this section is to provide a formal, unambiguous definition of the metrics we use.
 08 We also provide an informal justification of the definition in each case. In Section 5 we provide an
 09 empirical justification of the metrics by applying them to several programming languages.
 10

11 3.1. From software metrics to grammar metrics

12 Since any grammar defines a language and provides a basis for deriving elements of that language,
 13 a grammar may be considered as both a specification and a program; indeed, this duality is often
 14 exploited in the construction of recursive descent parsers [23].

15 Conceptually, we may think of any program as consisting of a set of procedures, where each
 16 procedure is defined by some procedure body, constructed using the control primitives of the language.
 17 Thus a procedure body may be represented as a graph whose nodes are statements and whose edges
 18 represent the flow of control between these statements. At a higher level of abstraction, we may
 19 represent the interaction between procedures by a *call graph*, whose nodes are procedures and whose
 20 edges represent a call from one procedure to another.
 21

22 In order to interpret the concepts of control-flow graph and call graph for context-free grammars we
 23 proceed as follows. The procedures correspond to non-terminals, and procedure bodies are the right-
 24 hand sides of the production rules. The control primitives are the union and concatenation operations
 25 of context free grammars, which correspond to alternation and sequencing, respectively. This mapping
 26 can be extended in a straightforward manner to the closure and option operators used in EBNF. In line
 27 with this mapping we interpret the call graph of a program as the graph of the successor relation
 28 between non-terminals.
 29

30 3.2. Size metrics

31 To ease the description of these metrics we adopt an algebraic notation. Given any grammar
 32 (N, T, S, P) , and a production rule $(n \rightarrow \alpha) \in P$, we refer to α as the *right-hand side* (RHS) of
 33 the rule. The RHS α is constructed from the application of the grammatical operators to the terminals
 34 and non-terminals from T and N .
 35

36 We denote this application in general as $f^k(\bar{x})$ where $k \in \{\cdot, |, ?, *, +, \epsilon\}$, representing the
 37 operations of concatenation, union, optionality, closure, positive closure and the empty string.
 38 The operands, represented by \bar{x} , must correspond in number to the arity of the operator. We specify that
 39 ϵ has no operands, optionality has one operand, and all the other operators have exactly two operands.
 40

41 3.2.1. Number of terminals and non-terminals

42 One of the simplest, course-grained metrics that can be applied to a program to measure its size
 43 is a count of the number of procedures that appear in that program. The equivalent size metric for
 44

01 context-free grammars is the number of non-terminals in that grammar, denoted VAR in [20]. This size
02 metric is commonly reported by parser generators such as *yacc* and *bison*.

$$03 \quad \text{Number of non-terminals (VAR)} = \#N \quad (1)$$

04
05 A related metric is the number of terminal symbols:

$$06 \quad \text{Number of terminals (TERM)} = \#T \quad (2)$$

07
08 Although these are the simplest possible estimates of grammar size, they can still provide useful
09 information about the grammar. A larger number of non-terminals implies a greater maintenance
10 overhead, since changes to the definition of one may effect many others. In implementation terms, the
11 size of the parse table is usually proportional to the number of terminals and non-terminals, particularly
12 for popular predictive parsing algorithms such as *LL(1)* and *LALR(1)*.
13

14 3.2.2. McCabe cyclomatic complexity

15
16 McCabe's metric measures the number of linearly independent paths through a flow graph [21].
17 This metric is typically interpreted as a measure of the number of decisions in the flow graph,
18 where decisions are typically represented by the use of Boolean-valued expressions in conditional and
19 iteration statements. This is a useful indicator of the level of difficulty involved in testing the procedure
20 under consideration, since a good test suite will seek to utilize as many paths as possible through the
21 flow graph.

22 Decisions in a context-free grammar are represented by the union and option operators for
23 conditionals, and the closure operator for iteration. Thus, our mapping of McCabe complexity to
24 grammars is to count the total number of alternatives in that grammar, as represented by occurrences
25 of these operators.

$$26 \quad \text{McCabe's cyclomatic complexity (MCC)} = \sum_{(n \rightarrow \alpha) \in P} mccabe(\alpha) \quad (3)$$

27
28 where

$$29 \quad mccabe(v) = 0 \quad \text{for } v \in (N \cup T),$$

$$30 \quad mccabe(f^k(\bar{x})) = 1 + mccabe(\bar{x}) \quad \text{for } k \in \{!, ?, *, +\},$$

$$31 \quad mccabe(f^k(\bar{x})) = mccabe(\bar{x}) \quad \text{for } k \in \{., \epsilon\}$$

32
33
34
35 Two grammars with the same number of non-terminals can still differ in essential complexity if one
36 grammar has significantly more alternatives for its non-terminals than the other grammar. The sum
37 of the McCabe complexity measure for these alternatives will highlight this difference. For parser
38 generators that use only the union operators, such as *yacc* and *bison*, the McCabe complexity of a
39 grammar is the number of distinct production rules it contains, and is also usually one of the measures
40 reported by such tools. The *MCC* metric is thus an extension of the *PROD* metric from context-free
41 grammars to full regular rightpart grammars.

42 The job of a parsing algorithm is to provide a means of choosing between the alternatives in a
43 grammar during a derivation. Thus, a high McCabe complexity indicates a greater potential for conflicts
44 in a lookahead-based parser, and a greater scope for backtracking in a search-based parser.



01 3.2.3. Average RHS size

02 In a procedure, the size metric is the number of nodes in the corresponding flow graph, and is used
 03 as a formal alternative to the common lines-of-code (loc) measure. Since production rules correspond
 04 to procedures, the nodes in a flow graph correspond to terminals or non-terminals on the RHS of a
 05 production rule. To compute the average RHS size, we calculate the total of the RHS sizes for each
 06 rule, and divide by the number of non-terminals.
 07

$$08 \text{ Average RHS size (AVS)} = \frac{\sum_{(n \rightarrow \alpha) \in P} \text{size}(\alpha)}{\#N} \quad (4)$$

09 where

$$10 \text{ size}(v) = 1 \quad \text{for } v \in (N \cup T),$$

$$11 \text{ size}(f^k(\bar{x})) = \text{size}(\bar{x}) \quad \text{for } k \in \{., \epsilon, |, ?, *, +\}$$

12 The average RHS size provides a measure of the number of symbols that we can expect to find,
 13 on average, on the right-hand side of a grammar rule. In some parsers, longer RHSs may mean that
 14 larger number of symbols or associated attributes must be placed on the parse stack, and so may have
 15 performance implications. Typically, it is usually possible to decrease the length of a RHS by replacing
 16 some of it by a new non-terminal, and thus the average RHS size metric should always be considered
 17 in association with the total count of the number of non-terminals.
 18
 19
 20
 21
 22

23 3.2.4. Halstead effort

24 Halstead's software science defines two main metrics to quantify programs: V , a measure of the size
 25 (or 'volume') of the program, and E , an attempt to estimate the effort required to understand that
 26 program. Both of these metrics are calculated as functions of the number of operators and operands
 27 a program contains. We can apply this to grammars by interpreting the operators as the standard
 28 grammatical operations, and the operands as the terminals and non-terminals in a given grammar.
 29 The value for program volume V yields little more information than $TERM$ and VAR given earlier.
 30 However, Halstead's effort metric E has the effect of relativising McCabe's metric, which counts
 31 the number of operators, by multiplying a weighting for the number of occurrences of the grammar
 32 symbols.
 33

34 Following the standard definition of Halstead effort E from [24], we define:

$$35 \text{ Halstead effort (HAL)} = \frac{\mu_1 \eta_2 (\eta_1 + \eta_2) \log_2(\mu_1 + \mu_2)}{2\mu_2} \quad (5)$$

36 where

$$37 \mu_1 = \text{no. of unique operators}$$

$$38 = \{., \epsilon, |, ?, *, +\} = 6$$

$$39 \mu_2 = \text{no. of unique operands}$$

$$40 = \#T + \#N$$

$$\begin{aligned}
& \eta_1 = \text{total occurrences of operators} \\
& = \sum_{(n \rightarrow \alpha) \in P} opr(\alpha) \\
& \eta_2 = \text{total occurrences of operands} \\
& = \sum_{(n \rightarrow \alpha) \in P} 1 + opd(\alpha)
\end{aligned}$$

and

$$\begin{aligned}
opr(v) &= 0 && \text{for } v \in (N \cup T) \\
opr(f^k(\bar{x})) &= 1 + opr(\bar{x}) && \text{for } k \in \{., \epsilon, |, ?, *, +\} \\
opd(v) &= 1 && \text{for } v \in (N \cup T) \\
opd(f^k(\bar{x})) &= opd(\bar{x}) && \text{for } k \in \{., \epsilon, |, ?, *, +\}
\end{aligned}$$

Since *HAL* is weighted by a measure of the grammar's size, unlike *MCC*, it provides a better basis for judging differences in complexity between grammars of different sizes.

3.3. Structural metrics

As described in Section 2 we can represent a grammar as a graph whose nodes are non-terminals, and where there is an edge between non-terminals A and B precisely when $A \triangleright B$. This concept parallels that of the call graph in programming languages, and provides the basis for the calculation of metrics based on the structure of a grammar.

3.3.1. Tree impurity

The call graph for a program is a directed graph indicating the dependencies between procedures in the program [4]. A high ratio of number of edges to procedures in a call graph indicates a high level of dependency between procedures; this complexity can complicate the testing process and can possibly indicate poor design. Since we regard a non-terminal as a procedure, and since the successor relation \triangleright^* 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, and express it as a percentage.

Following [4], the formula to compute the impurity metric for a call graph with n nodes and e edges is

$$\text{Tree impurity (TIMP)} = \frac{2(e - n + 1)}{(n - 1)(n - 2)} \times 100 \quad (6)$$

where

$$\begin{aligned}
n &= \#N \\
e &= \#\{(A \triangleright^* B) \mid A, B \in N\}
\end{aligned}$$



01 Very often a grammar must be refactored in order to make it amenable to a particular parsing
 02 algorithm. It is reasonable to suggest that a high impurity level for a grammar indicates that this
 03 refactoring process will be complicated, since a change in one rule may impact many other rules.

04 3.3.2. Normalized count of levels

05
 06 In this metric, we use the call graph, used in the calculation of the tree impurity metric, to
 07 partition the non-terminals into a set of equivalence classes called grammatical levels. Since each
 08 of these grammatical levels internally forms a complete graph, we may assume a high degree of
 09 interdependence between the non-terminals in a given level.
 10

11 The number of levels that can be derived from a grammar *LEV* gives some idea of the spread of non-
 12 terminals among the grammatical levels. It is, however, dependent on the number of non-terminals,
 13 since, for any grammar, the number of levels lies between 1 and $\#N$. Thus, to facilitate comparison
 14 between languages, we define the *normalized* number of levels as the total number of levels expressed
 15 as a percentage of the total possible number of levels.

$$16 \text{ Levels (CLEV)} = \frac{\#(N_{\equiv})}{\#N} \times 100 \quad (7)$$

17
 18 where N_{\equiv} is the partition induced on the set of non-terminals N by the equivalence relation \equiv defined
 19 in Section 2.

20 A low value here suggests that the non-terminals are clustered into a few equivalence classes, and
 21 that these are logical choices for modularization. A higher value means that the grammar is more
 22 evenly spread out between the non-terminals, and there should be more opportunities for modularizing
 23 the grammar.
 24

25 3.3.3. Number of non-singleton levels

26
 27 An equivalence class of size 1 indicates a high degree of specification in the grammar, since this
 28 non-terminal is not readily interchangeable with any others. On the other hand, larger equivalence
 29 classes represent high degrees of mutual recursion among the rules, suggesting a clustering of related
 30 functionality.
 31

32 Our experience indicates that many of the equivalence classes derived from the call graphs are in fact
 33 of size 1, and that central language concepts, such as *declarations*, *expressions* and *statements* tend to
 34 be represented by larger classes. Thus we define a metric to measure the number of these larger classes,
 35 since investigation of these non-singleton sets throws the most light on the logical groupings among
 36 the non-terminals:

$$37 \text{ Non-singleton levels (NSLEV)} = \#\{n \in N_{\equiv} \mid \#n > 1\} \quad (8)$$

38 Since the count of non-singleton levels is usually quite small, we choose not to normalize this for each
 39 grammar.
 40

41 3.3.4. Size of largest level (DEP)

42
 43 The depth metric for a grammar measures the number of non-terminals in the largest grammatical level.
 44 If the depth value constitutes a significant proportion of the total number of non-terminals, then this

01 value indicates (at least) an uneven distribution of the non-terminals among these levels.

$$02 \quad \text{Size of largest level (DEP)} = \max\{\#n \mid n \in N_{\equiv}\} \quad (9)$$

04 3.3.5. Maximum height

06 We can extend the notion of the successor relation to grammatical levels, allowing us to form a tree
07 with levels as nodes. For any two grammatical levels $N_1, N_2 \in N_{\equiv}$, set $N_1 \triangleright N_2$ precisely when there
08 exists $n_1 \in N_1$ and $n_2 \in N_2$ such that $n_1 \triangleright n_2$. The maximum height of this tree gives us another
09 measure of the dispersion of the non-terminals among the grammatical levels.

$$11 \quad \text{Maximum height (HEI)} = \max\{i \mid N_1 \triangleright N_2 \triangleright \dots \triangleright N_i\} \quad (10)$$

13 While this metric can be used effectively in the discussion of theoretical properties of context-free
14 grammars, our experience suggests that it is less useful in practice.

15 Each of the metrics described above can be calculated automatically from a context-free grammar
16 and a tool to accomplish this is described in Section 4.

18 4. SYNQ: A TOOL TO CALCULATE SYNTACTIC METRICS

21 In this section we present an overview of the implementation of SYNQ a tool that, given a grammar,
22 will automatically compute those metrics described in Section 3. SYNQ was written in C++ and
23 implemented using GNU flex version 2.5.4, GNU bison version 1.35, and the GNU C++ compiler
24 version 3.2.2. As well as demonstrating the feasibility of our approach, the construction of SYNQ also
25 acts as an operational definition of the syntactic metrics described formally in Section 3.

26 Figures 1–3 use the Unified Modelling Language, UML, to capture information about SYNQ [25].
27 Figure 1 provides an overview of the behaviour of SYNQ. The input to SYNQ is a grammar, described
28 using a superset of the *yacc* syntax extended to include the full set of EBNF operators. It is important
29 to allow the full set of EBNF operations here, since many programming language standards present
30 their grammar in this way, and transformation to another format might affect the metrics.

31 As can be seen in Figure 1, in phase I the input grammar is first scanned and parsed, and an abstract
32 syntax tree (AST) representing the production rules is then generated. The output is produced in two
33 further phases. Phase II uses a set of *Visitors*, described below, to generate the size and complexity
34 metrics, and to create a table representing the successor relation between non-terminals. Phase III
35 calculates the closure of this relation, derives the equivalence classes, and produces the structural
36 metrics.

38 4.1. The role of Visitors

40 Phase II of SYNQ is best discussed in the context of the architectural view of SYNQ, as given in the
41 class diagram of Figure 2. The AST representing the input grammar is implemented as a sequence of
42 production rules, where each rule consists of the non-terminal being defined, along with a `RightPart`.
43 As can be seen from Figure 2, the `RightPart` is recursively defined as a tree representing all the various
44 EBNF operations.



01
02
03
04
05
06
07
08
09
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44

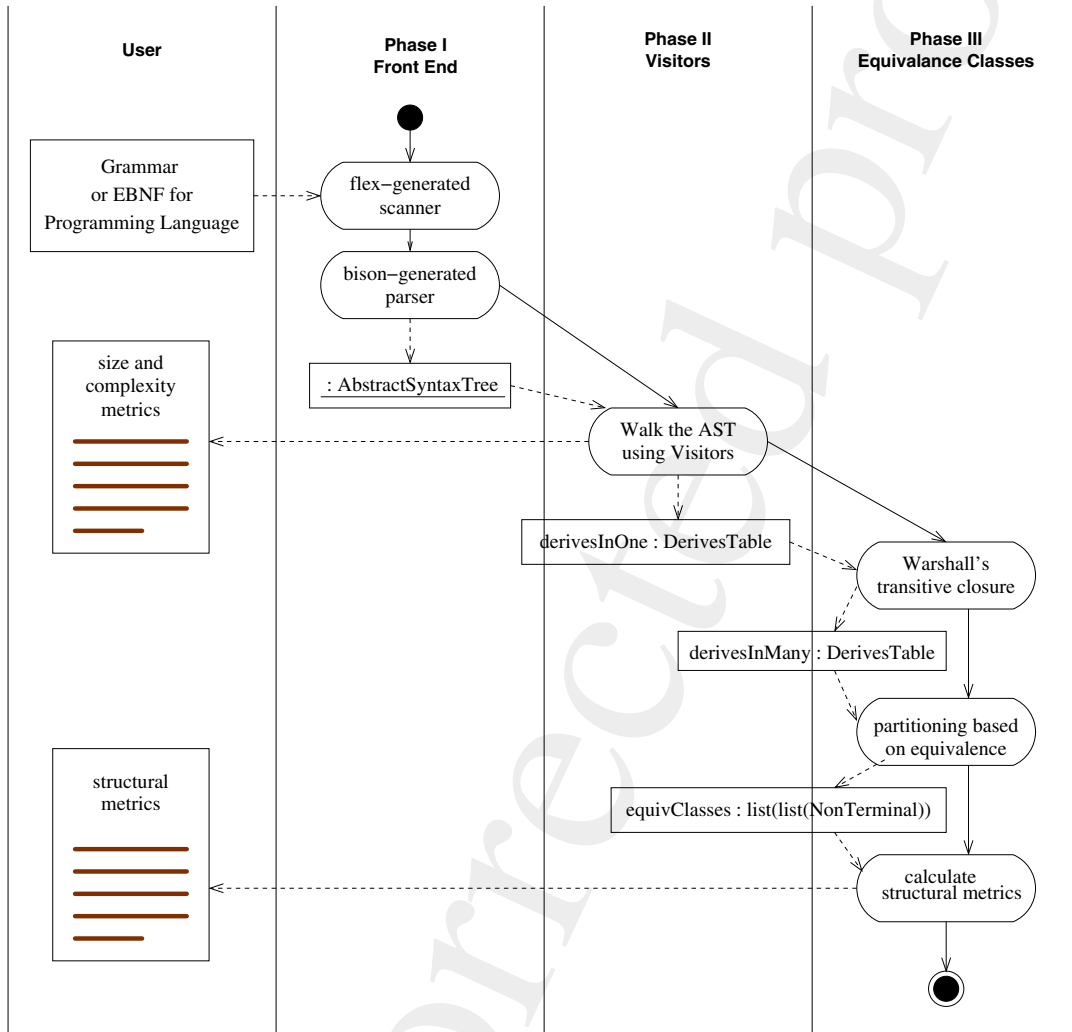


Figure 1. A behavioural overview of the tool. This UML activity diagram provides an overview of SYNQ, the tool that we constructed to compute the ten metrics. Input to SYNQ, indicated in the upper left corner, is the EBNF for the grammar under consideration. SYNQ's output is the result of the computed metrics that measure the *size and complexity* of the grammar and *structure* of the grammar.

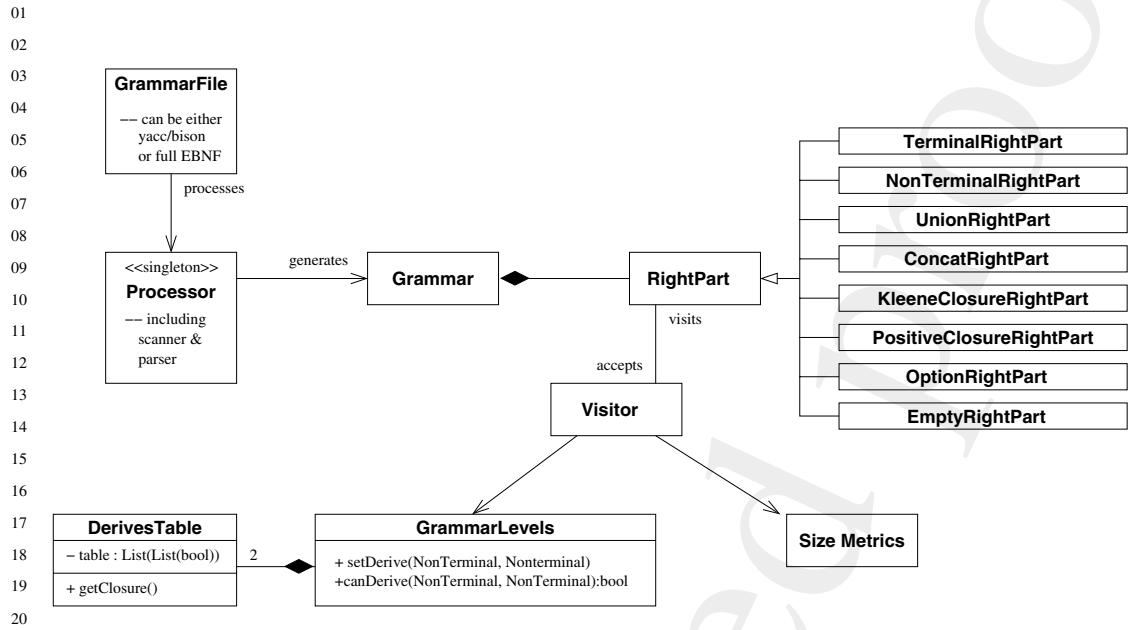


Figure 2. An architectural overview of SYNQ. This UML class diagram depicts the main classes used in SYNQ. The central entity is the Grammar class, which represents a grammar rules as a mapping from a non-terminal to an instance of the RightPart class.

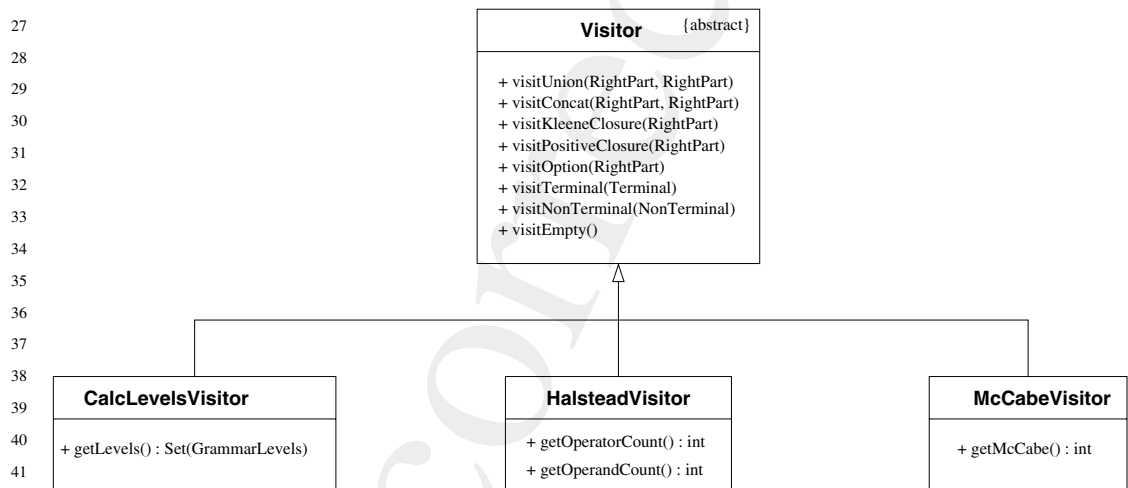


Figure 3. The Visitor classes. This UML class diagram presents some of the classes in our Visitor hierarchy. The abstract base class Visitor is subclassed for each metric that is calculated.



01 The structure of the `RightPart` class hierarchy is constructed to correspond to the ‘node hierarchy’
 02 of the *Visitor Pattern* as described in reference [26]. The use of the *Visitor Pattern* here allows us to
 03 add new functionality, such as a new metric, in a modular way, without changing the SYNQ front-end.
 04 It also facilitates the decoupling of the front- and back-ends of SYNQ. The visitors are used to directly
 05 generate the size and complexity metrics, as well as the grammar levels from which the structural
 06 metrics can be calculated.

07 An overview of some of the visitors used is given in Figure 3. Each metric is calculated by iterating
 08 through the production rules, applying the visitor to each `RightPart`. The generic code for handling
 09 visitors in a `RightPart` simply hands back control to the `Visitor` object, which has specialized methods
 10 to deal with each type of `RightPart`, as can be seen in the `Visitor` class of Figure 3. Each of the size
 11 and complexity metrics has its own subclass of `Visitor` whose methods contain all the code required by
 12 that metric, e.g. there is a `McCabeVisitor` and a `HalsteadVisitor` to calculate the respective metrics.

14 4.2. Representing grammatical levels

15 From Figure 1, we can see that phase III of SYNQ uses three main data structures: a graph of the
 16 immediate successor relation, a graph of the successor relation, and a graph of equivalence classes
 17 representing the grammatical levels. From these graphs we compute the structural metrics.

18 The immediate successor relation is calculated using a subclass of `Visitor`, which generates as an
 19 instance of class `DerivesTable`, as shown in Figure 2. This contains a two-dimensional Boolean matrix,
 20 S , indexed by non-terminals. For any non-terminals A and B , the entry $S[A, B]$ is true precisely when
 21 $A \triangleright B$.

22 We traverse this matrix, applying Warshall’s transitive closure algorithm to produce a second
 23 instance of `DerivesTable`, representing the closure of the first. This is basically a matrix S^* , where
 24 now $S^*[A, B]$ is true precisely when $A \triangleright^* B$. The tree impurity metric can be calculated directly by
 25 counting the number of nodes and edges in S , since this graph represents the successor relation.

26 The final data structure required by SYNQ represents the list of grammatical levels. Each
 27 grammatical level consists of a set of non-terminals where any two non-terminals A and B are in the
 28 same grammatical level when $A \equiv B$, i.e. when both $S^*[A, B]$ and $S^*[B, A]$ are true. These levels can
 29 thus be readily calculated from the S^* matrix. The remaining structural metrics are generated directly
 30 from this list of grammatical levels.

33 5. MEASUREMENT OF PROGRAMMING LANGUAGES

34 In this section, we describe the results of a study using SYNQ, a tool that implements the approach
 35 described in Section 4. There were two purposes to this study. First, we wished to calibrate the metrics,
 36 to ensure that they could discriminate between different grammars, and that the results could be
 37 explained. Second, we wanted to provide a basis for quantifying some of the anecdotal information
 38 regarding the size and complexity of programming languages.

39 Our experiments were conducted on the syntax of four well-known modern programming languages:

- 41 • ISO C [27,28],
- 42 • ISO C++ [29,30],
- 43 • Java v1.1 [31],
- 44 • ECMA-Standard C# [32].

Table I. Size and complexity metrics. The results of applying the metrics to measure the overall size and complexity of the four programming languages are shown. The last row of the table lists results for the Halstead effort metric, HAL; the columns are sorted according to the results for HAL in increasing order.

	C	Java	C++	C [#]
TERM	86	100	116	138
VAR	65	149	141	245
MCC	149	213	368	466
AVS	5.9	4.1	6.1	4.7
HAL	51	95	173	228

In each case the programming language syntax is described by a grammar taken directly from the first reference cited. The grammars were in various formats, all corresponding roughly to EBNF. None of the grammars were transformed structurally, and the only modifications made were minor changes of notation and elimination of duplicate production rules. In particular, no transformations were applied to make the grammars more amenable to any parsing algorithm. In each case, the grammars describe the syntactic structure of the language and no semantic attributes or actions are included in any of the grammars.

In the Section 5.1 we present our results for those metrics that describe the size and complexity of the programming language syntax. In Section 5.2 we present our results for those metrics that describe the structure of the language, in particular, those metrics derived from the grammatical levels of the syntax.

5.1. Size and complexity metrics

Table I presents the results of using five metrics that measure the overall syntactic size and complexity of the four languages. The header row of the table lists the languages and the leftmost column of the table lists the applied metrics. The columns are sorted in increasing order of Halstead's effort metric, expressed in thousands.

The first and second rows of data in Table I list the number of terminals, **TERM**, and non-terminals, **VAR**, in each grammar. Since these counts are the ones most obvious to someone reading a grammar, we might partition the grammars into three classes. The smallest grammar is ISO C, in the middle we have Java and C++, while C[#] has the largest number of terminals and non-terminals. The relative positions of C and C++ will probably not surprise anyone familiar with these languages, but it is interesting that the C[#] language, still at a relatively early stage of evolution, should have such a high count of grammar symbols.

The third row of data in Table I gives the McCabe complexity, **MCC**, for each grammar, and shows a pattern somewhat similar to the previous two rows. As expected, C is the smallest, with Java closely



Table II. Structure metrics. The results depicted in this table show the results of applying the metrics to measure the structure of the four languages. The columns are sorted in increasing order of TIMP, the tree impurity metric.

	C#	Java	C	C++
TIMP (%)	29.7	32.7	64.1	85.8
CLEV (%)	64.9	59.7	33.8	14.9
NSLEV	5	4	3	1
DEP	44	33	38	121
HEI	28	23	13	4

following. Also, as expected, C++ is considerably larger than both, but again C# comes out with the highest complexity.

The similarity of the numbers across the fourth row of data in Table I, ranging from 4.1 up to 6.1, reflect the notion that grammar writers typically do not allow the RHS 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.

The fifth row of data in Table I shows the values of Halstead's effort metric, HAL, for each grammar, and could be seen as a summary of the preceding rows, since the complexity and number of operators is relativized by the number of symbols. We see a similar ranking to that provided by the McCabe metric, with a progression from C, to Java, to C++ and finally to C#.

5.2. Structural metrics

Table II presents the results of using the structural metrics to measure the overall structure of the four languages. The header row of the table lists the four languages and the leftmost column lists the structural metrics applied. The columns have been sorted in order of increasing tree impurity, expressed here as a percentage. The Tree Impurity metric, TIMP, is derived from the closure of the call graph generated by the grammar, whereas the remaining metrics are derived from the calculated grammatical levels.

The metric presented in the first data row of Table II, TIMP, presents the results for the tree impurity of the grammars. The ordering now is changed from the size metrics, in that C# and Java have quite similar impurity levels, whereas C and its successor, C++, have quite high levels of impurity. The impurity numbers for C and C++, at 64.1% 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 for these languages.

The second data row of Table II shows the total number of levels for each grammar, CLEV, as a percentage of the total number of non-terminals. Since the impurity metric measures the degree to which the equivalence classes form a fully connected graph, it is not surprising that the number of equivalence classes in a grammar vary roughly inversely with the impurity. Both C# and Java have

01 a high number of equivalence classes, indicating that the non-terminals are well spread out between
02 these classes. At the other end, the relatively low figure for C++, just under 15%, indicates that the
03 non-terminals are clustered among relatively few equivalence classes.

04 A closer picture of the distribution of non-terminals among grammatical levels is given by the last
05 three data rows of Table II. The number of non-singleton levels, NSLEV, is particularly revealing,
06 since these describe the main syntactic components within the grammar, such as declarations, types,
07 statements or expressions. We can see that for all the grammars other than C++ this value is either 3, 4
08 or 5.

09 The three non-singleton levels for Java reflect categories for expressions, statements and types.
10 The size of the largest level, reflected by the depth metric, DEP, corresponds to the number of non-
11 terminals in the level for Java expressions. The other two non-singleton levels for Java contain 25
12 non-terminals for statements and four non-terminals for types. A similar picture emerges with C#,
13 where these three levels are augmented with two other small levels, one each for class and namespace
14 member declarations.

15 There were just two non-singleton grammatical levels generated for the C grammar. The depth
16 metric of 38 reflects the cardinality of the largest grammatical level, which contained non-terminals for
17 expressions and declarations. The other non-singleton grammatical level, that contained non-terminals
18 relating to statements, had a cardinality of 6. Finally, the depth metric for C++ at 121 non-terminals
19 is the cardinality of the only non-singleton grammatical level for this grammar. This is the least
20 modular of all the grammars, where non-terminals for types, declarations, statements and expressions
21 are combined into a single large grammatical level containing 86% of the non-terminals in the grammar.

22 The HEI metric, shown in the last row of Table II, is more difficult to interpret. The grammatical
23 levels form a tree, where the root node is the level containing the start symbol, and level L_2 is a child
24 of L_1 precisely when $L_1 \succ L_2$. The HEI metric measures the height of this tree but, due to the large
25 number of singleton levels, does not appear to shed any new light on the grammatical complexity;
26 nevertheless, we include the HEI metric for completion so that all of the metrics of [20] are included
27 in our study. It is notable at least that the comparatively low HEI value for C++ reflects the clustering
28 of a large number of non-terminals in a single level.

29

30

31 6. CASE STUDY: THE EVOLUTION OF THE GNU C++ PARSER

32

33 In Section 5 we applied our metrics to the syntax of four different programming languages. While most
34 programming languages are defined by standards documents that specify a grammar for the language,
35 the transformation of this grammar into a parser is often a non-trivial task. Some standards, such as [28]
36 or [31], present both a reference grammar and a parser-friendly equivalent. However, the construction
37 of a parser for ISO C++ programming is notoriously difficult [33–38]. While some of this difficulty
38 is related to semantic issues, a portion of it is related to the complexity and scale of the language, as
39 indicated by the metrics in the previous section.

40 Programming languages can evolve over time as features are added and standardized and this
41 can also result in a need for the evolution of the corresponding grammar. In this section we use
42 our metrics to chart the evolution of the GNU C++ grammar, contained in *gcc*, the GNU compiler
43 collection. This evolution was the result of two separate, but related threads: the evolution of the C++
44 programming language toward ISO standardization, and the convergence of the GNU C++ parser



Table III. The evolution of the GNU C++ parser. The metrics in this table chart the development of the grammar used in the C++ parser from each major version of the GNU compiler, from version 2.0 to 3.0. The table is ordered chronologically, by version number.

Version	Feb 1992	Mar 1992	Jun 1992	Dec 1992	Jun 1993	Jan 1994	Nov 1994	Nov 1995	Jan 1998	Jul 1999	Jun 2001
TERM	105	107	107	107	109	108	104	107	112	110	111
VAR	146	150	149	148	152	158	193	202	214	232	236
MCC	483	491	490	497	500	512	511	524	568	592	624
HAL	576	584	591	632	624	664	545	528	585	620	699
AVS	11	11	11	12	12	12	8.8	8.4	8.6	8.3	8.7
TIMP (%)	56.9	55.7	56.0	57.4	57.8	58.7	64.1	63.5	71.5	72.6	73.4
CLEV (%)	42.5	44.0	43.6	41.9	41.4	40.5	33.7	34.2	27.6	25.9	25.0
NSLEV	2	2	2	2	2	2	2	2	2	3	3
DEP	84	84	84	86	89	94	127	131	152	169	174
HEI	11	12	12	12	12	12	12	14	11	12	12

toward that standard. By using our metrics to quantify this evolution, we establish two important properties of the measures. First, the increasing complexity of the GNU C++ parser is reflected by the metrics as we move from version to version. Second, the metrics are capable of quantifying the differences between major and minor changes as we move from one version to the next.

6.1. Stages in the evolution of the GNU C++ compiler

Table III presents the results of applying our metrics to the grammar used in each major version of the C++ parser from *gcc* between version 2.0, released on 22 February 1992, and version 3.0, released on 18 June 2001. This series of releases covers the evolution of the C++ parser from one of the earliest versions of the compiler to a version that is close to compliance with the ISO standard [39]. Each version of *gcc* uses a *bison*-compatible C++ grammar at the core of the C++ compiler, and it is this grammar that was measured in each case.

Although metrics cannot provide detailed information about the impact of grammar transformations across different versions of the *gcc* C++ compiler, they can enable us to track trends in the evolution of the parser. The number of terminal symbols **TERM** remains broadly constant through the versions, as might be expected, increasing gradually from 105 to 111 terminals. The number of non-terminals **VAR** does not show much variance at first, but makes a sharp jump between versions 2.5 and 2.6, when 35 new non-terminal symbols were added. As can be seen from the figures reflecting the average RHS size, this corresponds to a drop in the size of a rule, from 12 to 8.8 symbols, clearly the result of a major refactoring.

MCC increases slowly up to a value of 512 between versions 2.0 and 2.5, reflecting a gradual addition of functionality to the parser. It is noticeable that **MCC** varies little between versions 2.5 and 2.6, further underlining our observation that this change is a refactoring, rather than a significant

01 addition to the grammar. Again, after this version the complexity increases more rapidly, reflecting the
02 increased level of development of the parser as it approaches ISO compliance. At 624, the complexity
03 of the *gcc* 3.0 grammar is almost twice that of the ISO standard C++ grammar, at 368. Clearly, this
04 reflects the degree of difficulty in constructing an *LALR*-compliant parser based on the C++ standard.

05 The rise in *HAL* mirrors the McCabe complexity, except for a drop between versions 2.5 and 2.6,
06 from 664 to 545, reflecting the increased number of non-terminal symbols. As before, there is a
07 notable acceleration in the rate of increase of effort after this version, climbing from 545 in version
08 2.6 to a value of 699 in version 3.0. This *HAL* for version 3.0 at 699 is considerably larger than the
09 corresponding value for the ISO standard C++ grammar at 173.

10 In the previous section we noted that most of the non-terminals for the ISO standard C++ grammar
11 are clustered into a single grammatical level. Thus, the structural metrics for the versions of the *gcc*
12 grammars shown in Table III reflect this to some extent. For version 2.0 through to version 2.8, the two
13 non-singleton levels shown in *NSLEV* reflect one large level containing most of the non-terminals,
14 and one small level containing definitions for external declarations. For the last two versions of *gcc*,
15 versions 2.95 and 3.0, another small level is added for template definitions.

16 Looking at the figures for *TIMP* in Table III, we can see that the parser's evolution can be broken
17 into three phases. The first phase, between version 2.0 and version 2.5, reflects the slow evolution of
18 the parser, as indicated previously by the size and complexity metrics. The sharp jump between version
19 2.5 and version 2.6, from 58.7% to 64.1% reflects the fact that of the 35 new non-terminals introduced
20 in this change, 33 of these were introduced in order to re-factor rules for non-terminals belonging to
21 the largest level, changing the *DEP* metric from 94 to 127 non-terminals.

22 The other significant change in the impurity of the grammar is between versions 2.7 and 2.8, where
23 the impurity rises from 62.5% to 71.5%. Here 12 new non-terminals are added to the grammar, but
24 a total of 21 non-terminals are added to the largest level, changing the *DEP* metric from 131 to 152.
25 This results from a two-pass approach to processing inline method definitions, where their definitions,
26 and thus the corresponding productions, are effectively added in at the end of the class definition.
27 We can see that this change also slightly increases the normalized count of levels *CLEV*, from 33.7%
28 to 34.2%. After this change, the count of levels decreases, indicating that any new non-terminals are
29 added to existing levels, rather than creating new ones.

30

31

32 7. RELATED WORK

33

34 In this section we review previous research that relates to our efforts at quantifying grammatical
35 complexity and applying this quantification to the maintenance of grammar-based software
36 applications. There has been considerable work directed at estimating the complexity of the
37 maintenance effort [5–8], and the incorporation of semantic considerations into this estimate [11,12];
38 however, none of the previous research on metric complexity has targeted grammar-based applications.
39 The work on grammar complexity was initially reported in [22,40] and extended in [20]; we begin by
40 reviewing the work in [20,40]. There has been some important work on parsing complexity that relates
41 to our work and we review the research described in [41]. Finally, a preliminary version of this work
42 was presented in [42]; we compare our current work reported in this paper to the preliminary report.

43 The definitions of *VAR*, *PROD*, *LEV*, *DEP* and *HEI* were introduced in [22,43–45] and we
44 reviewed them in Section 2. These complexity measures classify context-free languages according



01 to the size or structural properties of their grammars. The size of grammars is expressed as the
02 number of terminals, **VAR**, and the number of productions, **PROD**. The number of grammatical levels
03 **LEV**, the maximal number of elements of grammatical levels **DEP**, and the length of the digraph of
04 grammatical levels **HEI** are the complexity measures reflecting the structure of grammars. An important
05 aspect of complexity theory is the study of the functional behaviour of the complexity measures on
06 language classes. Complexity measures are functions defined on context-free languages, where the
07 function values are natural numbers. The main result of reference [20] establishes the unboundedness
08 of complexity measures **VAR**, **PROD**, **LEV**, **DEP** and **HEI** on the classes of languages defined by
09 grammar forms.

10 Algorithms for corpus-based parsing and several metrics are described in [41] for evaluating the
11 algorithms. Input to corpus-based parsing is a *treebank*, a collection of text annotated with the 'correct'
12 parse tree. The goal is to find algorithms that, given unlabelled text from the treebank, produce a parse
13 that is similar to the one in the treebank. Evaluation metrics are used to determine whether a candidate
14 parse matches the correct parse; the metrics include, among other criteria, N_G , the number of non-
15 terminals in the guessed parse tree, and N_C , the number of non-terminals in the correct parse. Some of
16 the metrics used in [41] are similar to those described in [22,43–45] and [20], but most are based on
17 parse trees rather than grammars.

18 Software metrics is a well-established field, and [4] presents a good overview. The formulation of
19 our metrics, particularly Section 3, was inspired by the work on object-oriented metrics [46], which
20 stressed unambiguous formal definition as a basis for defining metrics. However, deciding on a choice
21 of metrics is often a difficult task, and can vary depending on their intended use. One approach to
22 validating metrics proposes a set of axioms that metrics should adhere to [47], but these are not without
23 controversy [4, Section 8.6]. Other approaches use standard statistical techniques to investigate desired
24 properties; for example, principal component analysis could be used to investigate the independence of
25 the metrics in our suite. However, further empirical work is required to provide sufficient data for such
26 a statistical analysis, as well as providing a basis for comparison with external product attributes.

27 The metrics presented in this paper were first described in preliminary form in [42]. The present
28 paper extends this work in a number of important ways. First, we have extended here and fully
29 formalized the metrics presented in [42]. Second, [42] only applied the metrics to C, C++ and Java,
30 whereas we have extended this to cover C[#], providing a better insight into the significance of the
31 metrics. Finally, the use of the metrics to chart the evolution of the C++ parser from *gcc* is unique to
32 this paper.

33

34

35

8. CONCLUDING REMARKS

36 In this paper, we have adapted the software metrics that are commonly used to measure program
37 complexity and to apply them to the measurement of the complexity of programming language syntax.
38 We have formally defined a suite of ten metrics measuring grammar size and structure. Measuring
39 grammars in this way extends the field of software metrics to cover grammar-based applications such
40 as compilers and program comprehension tools.

41 We have described SYNQ, our tool that takes, as input, an EBNF for a context-free grammar and
42 produces, as output, results for the computed metrics. We have presented the results of applying our
43 metrics to four commonly used programming languages as well as several versions of the evolution of
44 the GNU C++ parser.

01 Using our metrics, together with the metrics described by other researchers [20,22,40,43–45], we
02 have shown a correlation between the computed results of the metrics and anecdotal reports about
03 language and the difficulty of parser construction. By using our metrics to track the evolution of the
04 GNU C++ parser we have demonstrated that they can usefully distinguish major and minor version
05 changes, as well as measure the impact of those changes.

06 The results in this paper can contribute to software maintenance and evolution in the following areas.

- 07 • Our metrics permit a comparative analysis of programming languages, thus providing a basis
08 for the relative estimation of a facet of the maintenance effort for software written using these
09 languages. In particular, these metrics have direct implications for tool construction and program
10 comprehension activities.
- 11 • The metrics can be used by language designers to estimate the effect of changing a language's
12 syntax, or adding a new language feature. While the creation of new programming languages
13 is a relatively esoteric occupation, there is a burgeoning research field in the construction and
14 processing of domain-specific languages [48], and our metrics can contribute to this area.
- 15 • Our metrics allow for the integration of grammar-based software artifacts into the metrication
16 process. For example, the C source code of the GNU compiler could have been analysed using
17 standard software metrics; our syntactic metrics allow for a complete picture of the evolution of
18 this software.

19
20 We believe that the metrics suite outlined in this paper can help with the construction and evaluation
21 of software analysis tools used in software maintenance, as well as contributing to the study of the
22 evolution of grammar-based software.

23 24 25 REFERENCES

- 26
27 1. Schach SR. *Object-Oriented and Classical Software Engineering*. McGraw-Hill: New York, 2001; 648 pp.
- 28 2. Martin J, McClure CL. *Software Maintenance: The Problem and its Solutions*. Prentice-Hall: Englewood Cliffs NJ, 1983;
29 472 pp.
- 30 3. Pigoski TM. *Practical Software Maintenance: Best Practices for Managing your Software Investment*. Wiley: New York,
31 1997; 400 pp.
- 32 4. Fenton NE, Pfleeger SL. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Publishing:
33 London, 1998; 656 pp.
- 34 5. Abran A, Silva I, Primera L. Field studies using functional size measurement in building estimation models for software
35 maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 2002; **14**(1):31–64.
- 36 6. Ahn Y, Suh J, Kim S, Kim H. The software maintenance project effort estimation model based on function points. *Journal*
37 *of Software Maintenance and Evolution: Research and Practice* 2003; **15**(2):71–85.
- 38 7. Harrison MS, Walton GH. Identifying high maintenance legacy software. *Journal of Software Maintenance and Evolution:*
39 *Research and Practice* 2002; **14**(6):429–446.
- 40 8. Polo M, Piattini M, Ruiz F. Using code metrics to predict maintenance of legacy programs: A case study. *Proceedings*
41 *International Conference on Software Maintenance*. IEEE Computer Society: Los Alamitos CA, 2001; 202–211.
- 42 9. Briand LC, Wust J, Lounis H. Using coupling measurement for impact analysis in object-oriented systems. *Proceedings*
43 *International Conference on Software Maintenance*. IEEE Computer Society: Los Alamitos CA, 1999; 475–482.
- 44 10. Sahraoui HA, Godin R, Miceli T. Can metrics help to bridge the gap between the improvement of OO design and its
automation. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society: Los Alamitos CA,
2000; 154–162.
11. Chapin N. An entropic metric for software maintainability. *Proceedings 22nd Annual Hawaii International Conference on*
System Sciences. IEEE Computer Society: Los Alamitos CA, 1989; 522–523.
12. Etzkorn LH, Gholston S, Hughes WE. A semantic entropy metric. *Journal of Software Maintenance and Evolution:*
Research and Practice 2002; **14**(4):293–310.



- 01 13. Malloy BA, Gibbs TH, Power JF. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software—*
 02 *Practice and Experience* 2002; **33**(1):19–39.
- 03 14. Reps T, Teitelbaum T, Demers A. Incremental context-dependent analysis for language-based editors. *ACM Transactions*
 04 *on Programming Languages and Systems* 1983; **5**(3):449–477.
- 05 15. Panda PR, Catthoor F, Dutt ND, Danckaert K, Brockmeyer E, Kulkarni C, Vandercappelle A, Kjeldsberg PG. Data and
 06 memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems*
 07 2001; **6**(2):149–206.
- 08 16. Rabbah RM, Palem KV. Data remapping for design space optimization of embedded memory systems. *ACM Transactions*
 09 *on Embedded Computing Systems* 2003; **2**(2):186–218.
- 10 17. Maletic J, Collard M, Marcus A. Source code files as structured documents. *Proceedings 10th International Workshop on*
 11 *Program Comprehension*. IEEE Computer Society: Los Alamitos CA, 2002; 289–292.
- 12 18. Irwin W, Churcher N. A generated parser of C++. Christchurch, New Zealand, 2001.
 13 <http://citeseer.nj.nec.com/irwin01generated.html> [13 January 2003].
- 14 19. Aho A, Sethi R, Ullman J. *Compilers: Principles, Techniques and Tools*. Addison-Wesley: Boston MA, 1986; 500 pp.
- 15 20. Csehaj-Varjú E, Kelemenová A. Descriptive complexity of context-free grammar forms. *Theoretical Computer Science*
 16 1993; **112**(2):277–289.
- 17 21. McCabe TJ. A complexity measure. *IEEE Transactions on Software Engineering* 1976; **2**(4):308–320.
- 18 22. Brauer W. On grammatical complexity of context-free languages. *Proceedings Symposium and Summer School on*
 19 *the Mathematical Foundations of Computer Science*. Mathematical Institute of the Slovak Academy of Sciences:
 20 Czechoslovakia, 1973; 193–196.
- 21 23. Elder J. *Compiler Construction: A Recursive Descent Model*. Prentice-Hall: Englewood Cliffs NJ, 1994; 437 pp.
- 22 24. Halstead M. *Elements of Software Science*. Elsevier Science: New York, 1977; 127 pp.
- 23 25. Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language User Guide (Object Technology Series)*. Addison-
 24 Wesley: Boston MA, 1998; 482 pp.
- 25 26. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-
 26 Wesley: Boston MA, 1995; 395 pp.
- 27 27. ISO/IEC JTC1/SC22/WG14. *International Standard: Programming Languages—C*, No. 9899:1999. International
 28 Organization for Standardization (ISO): Geneva, Switzerland, 1999; 538 pp.
- 29 28. Kernighan B, Ritchie D. *The C Programming Language*. Prentice-Hall: Englewood Cliffs NJ, 1988; 274 pp.
- 30 29. ISO/IEC JTC1/SC22/WG21. *International Standard: Programming Languages—C++*, No. 14882:2003. International
 31 Organization for Standardization (ISO): Geneva, Switzerland, 2003; 786 pp.
- 32 30. Stroustrup B. *The C++ Programming Language*. Addison-Wesley: Boston MA, 1997; 911 pp.
- 33 31. Gosling J, Joy B, Steele G. *The Java Language Specification*. Addison-Wesley: Boston MA, 2000; 544 pp.
- 34 32. ECMA. *C# Language Specification*, No. ECMA-334. European Computer Manufacturers Association: Geneva,
 35 Switzerland, 2002; 471 pp.
- 36 33. Bodin F, Beckman P, Gannon D, Gotwals J, Narayana S, Srinivas S, Winnicka B. Sage++: An object-oriented toolkit
 37 and class library for building Fortran and C++ restructuring tools. *Proceedings 2nd Annual Object-oriented Numerics*
 38 *Conference*. Rogue Wave Software: Corvallis OR, 1994; 122–136.
- 39 34. Knapen G, Lague B, Dagenais M, Merlo E. Parsing C++ despite missing declarations. *Proceedings 7th International*
 40 *Workshop on Program Comprehension*. IEEE Computer Society: Los Alamitos CA, 1999; 114–125.
- 41 35. Lillie J. PCCTS-based LL(1) C++ parser: Design and theory of operation, version 1.5. Westminster CO, 1997.
 42 <http://www.empathy.com/pccts> [13 January 2003].
- 43 36. Power JF, Malloy BA. Symbol table construction and name lookup in ISO C++. *Proceedings 37th International Conference*
 44 *on Technology of Object-Oriented Languages and Systems*. IEEE Computer Society: Los Alamitos CA, 2000; 57–68.
- 37 37. Reiss S, Davis T. *Experiences Writing Object-oriented Compiler Front Ends*. Brown University: Providence RI, 1995;
 38 18 pp.
- 39 38. Roskind J. A YACC-able C++ 2.1 grammar, and the resulting ambiguities, release 2.0. Indialantic FL, 1991.
 40 <ftp://ftp.iecc.com/pub/file/c++grammar> [13 January 2003].
- 41 39. Malloy BA, Linde SA, Duffy EB, Power JF. Testing C++ compilers for ISO language conformance. *Dr. Dobbs Journal*
 42 2002; **27**(6):71–80.
- 43 40. Ginsburg S, Lynch N. Size complexity in context-free grammar forms. *Journal of the Association for Computing Machinery*
 44 1976; **23**(4):582–598.
- 41 41. Goodman J. Parsing algorithms and metrics. *Proceedings 34th Annual Meeting of the Association for Computational*
 42 *Linguistics*. Association for Computational Linguistics: East Stroudsburg PA, 1996; 177–183.
- 43 42. Power JF, Malloy BA. Metric-based analysis of context-free grammars. *Proceedings 8th International Workshop on*
 44 *Program Comprehension*. IEEE Computer Society: Los Alamitos CA, 2000; 171–178.
- 43 43. Gruska J. Some classifications of context-free languages. *Information and Control* 1969; **14**:152–179.
- 44 44. Kelemenová A. Complexity of normal form grammars. *Theoretical Computer Science* 1984; **28**(3):299–314.

- 01 45. Kelemenová A. Structural complexity measures on grammar forms. *Proceedings 2nd Conference on Automata, Languages*
02 *and Programming Systems*. Springer: Heidelberg, 1988; 73–76.
- 03 46. Briand L, Daly J, Wust J. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions*
04 *on Software Engineering* 1999; **25**(1):91–121.
- 05 47. Weyuker E. Evaluating software complexity measures. *IEEE Transactions on Software Engineering* 1988; **14**(9):
06 1357–1365.
- 07 48. van Deursen A, Klint P, Visser J. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices* 2000;
08 **35**(6):26–36.

09 **AUTHORS' BIOGRAPHIES**



James F. Power is a lecturer at the Department of Computer Scienc, National University of Ireland, Maynooth. His research interests include compiler design, program comprehension and formal methods. He received a PhD and MSc in Computer Science from Dublin City University, and a BSc in Computer Science from University College Dublin.



Brian A. Malloy is an Associate Professor at the Department of Computer Science, Clemson University. His research interests include software engineering, compiler technology, software design, software specification and software testing. He received a PhD and MS in Computer Science from the University of Pittsburgh, and a BA in Mathematics from La Salle College, Philadelphia.

Annotations from smr293.pdf

Page 21

Annotation 1

Au:

Please provide town of publisher for reference [22].