JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2004; **Accepted for publication**:1–20 *Prepared using* smrauth.cls
[Version: 2003/05/07 v1.1]

Research

A metrics suite for grammar-based software

James F. Power[1], and Brian A. Malloy[2]*

[1] *Dept. of Computer Science, National University of Ireland, Maynooth, Co. Kildare, Ireland.*
[2] *Dept. of Computer Science, Clemson University, Clemson, SC 29634, USA.*

SUMMARY

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

No. of Figures: 3. No. of Tables: 3. No. of References: 48.

KEY WORDS: Program comprehension, software metrics, source code analysis, call graph, maintenance of grammar-based tools

1. Introduction

One of the most important and costly phases in the life cycle of software applications is maintenance. Estimates of the cost of maintenance range from 65% [1] to 80% [2, 3] of the software budget. Much of this cost derives from the burgeoning complexity of the software application as it evolves over time. During the life cycle of an application, many modifications and extenstions to the software are needed to support evolving business requirements, increasing the complexity of the application. A program that is overly complex may be difficult

*Correspondence to: Brian Malloy at malloy@cs.clemson.edu

Received October 3, 2003
Revised August 16, 2004

to comprehend and therefore more costly to maintain. Software complexity may induce a significant number of operational failures that require a greater amount of maintenance to correct errors introduced as a result of modifications and extensions to the code.

One approach to measuring and managing the complexity of software, as it evolves over time, is to exploit software metrics. The use of software metrics has become essential to good software engineering [4]. Many developers measure characteristics of a program to determine whether the requirements are consistent and complete, whether the design is of high quality and whether the code is too complex to permit easy test and maintenance. Metrics have been used to estimate the complexity of the maintenance effort [5, 6, 7, 8], to facilitate change impact analysis [9], and as an indicator for automatic detection of a transformation that can improve the quality of a system [10]. Recent research incorporates semantic information into metric considerations [11, 12].

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

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

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

Prepared using smrauth.cls

To demonstrate the feasability of our approach, we present the design and implementation of a tool, SYNQ[†], that automatically computes our metrics for a given grammar. We demonstrate the applicability of our work to software evolution by using SYNQ to chart the evolution of the GNU C++ compiler through several versions. We selected GNU C++ as our case study because it is a commonly used grammar-based application and the source code is available through the GNU public license. This case study of evolution demonstrates that our metrics quantify the expected properties associated with the evolution. In particular, we show that the grammar at the core of the GNU C++ compiler has increased in complexity, and that the metrics permit us to quantify the differences between minor and major complexity increases. Since the behaviour of a grammar-based tool is typically choreographed by the grammar rules, the measure of syntactic complexity that our metrics provide can guide maintainers of grammar-based applications in locating problematic areas in the application.

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

2. Background

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

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

[†]SYNQ is pronounced "sink", the SYNtactic Quantification or measurement of grammar complexity.

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.

Formally a grammar is a four-tuple (N, T, S, P) where N and T are disjoint sets of symbols known as non-terminals 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 that produce sequences of terminals, known as the *sentences* of the language.

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

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

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

2.2. Grammatical Levels

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

If non-terminal A derives some sequence of symbols β, and β contains some non-terminal B we say that B is an *immediate successor* of A, and write $A \rhd B$. If β derives some sequence of symbols γ, and 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*. For any two non-terminals A and C in different levels, if $A \rhd C$ then this naturally induces a corresponding ordering on

their levels. For different levels L_1 and L_2, if $A \in L_1$ and $C \in L_2$, then when $A \rhd C$ we write $L_1 \succ L_2$.

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

VAR	The number of non-terminals
PROD	The number of production rules
LEV	The number of grammatical levels
DEP	The number of non-terminals in the largest grammatical level
HEI	The maximum length of the chain of levels $L_0 \ldots L_n$ such that each $L_i \succ L_{i+1}$ for $0 \leq i \leq n$

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

3. A Metrics suite for programming language syntax

In this section, we define a set of ten metrics that are used in the remainder of this paper. The *size* metrics are adaptations of standard metrics for programs and procedures [21, 4]. The *structural* 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 [20, 22]. For each of the ten metrics we present its formal definition and discuss some of the pragmatics of its use.

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

3.1. From Software Metrics to Grammar Metrics

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 [23].

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.

3.2. Size Metrics

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

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

3.2.1. Number of Terminals and Non-Terminals (TERM, VAR)

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, denoted VAR in [20]. This size metric is commonly reported by parser generators such as *yacc* and *bison*.

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

A related metric is the number of terminal symbols:

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

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

3.2.2. McCabe Cyclomatic Complexity (MCC)

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

Decisions in a context-free grammar are represented by the union and option operators for conditionals, and the closure operator for iteration. Thus, our mapping of McCabe complexity

to grammars is to count the total number of alternatives in that grammar, as represented by occurrences of these operators.

$$\text{McCabe Complexity (MCC)} = \sum_{(n \to \alpha) \in P} mccabe(\alpha) \tag{3}$$

where

$$\begin{array}{rcll} mccabe(v) & = & 0 & for\ v \in (N \cup T), \\ mccabe(f^k(\bar{x})) & = & 1 + mccabe(\bar{x}) & for\ k \in \{|, ?, *, +\}, \\ mccabe(f^k(\bar{x})) & = & mccabe(\bar{x}) & for\ k \in \{\cdot, \epsilon\} \end{array}$$

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. For parser generators that use only the union operators, such as *yacc* and *bison*, the McCabe complexity of a grammar is the number of distinct production rules it contains, and is also usually one of the measures reported by such tools. The MCC metric is thus an extension of the $PROD$ metric from context-free grammars to full regular rightpart grammars.

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

3.2.3. Average RHS Size (AVS)

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 on the RHS of a production rule. To compute the average RHS size, we calculate the total of the RHS sizes for each rule, and divide by the number of non-terminals.

$$\text{Average RHS Size (AVS)} = \frac{\sum_{(n \to \alpha) \in P} size(\alpha)}{\#N} \tag{4}$$

where

$$\begin{array}{rcll} size(v) & = & 1 & for\ v \in (N \cup T), \\ size(f^k(\bar{x})) & = & size(\bar{x}) & for\ k \in \{\cdot, \epsilon, |, ?, *, +\} \end{array}$$

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

3.2.4. Halstead Effort (HAL)

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

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

$$\text{Halstead Effort (HAL)} = \frac{\mu_1 \eta_2 (\eta_1 + \eta_2) \log_2(\mu_1 + \mu_2)}{2\mu_2} \tag{5}$$

where

$$\begin{aligned}
\mu_1 &= \text{no. of unique operators} \\
&= \#\{\cdot, \epsilon, |, ?, *, +\} = 6 \\
\mu_2 &= \text{no. of unique operands} \\
&= \#T + \#N \\
\eta_1 &= \text{total occurrences of operators} \\
&= \sum_{(n \to \alpha) \in P} opr(\alpha) \\
\eta_2 &= \text{total occurrences of operands} \\
&= \sum_{(n \to \alpha) \in P} 1 + opd(\alpha)
\end{aligned}$$

and

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

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 $A \rhd B$. 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 (TIMP)

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 normalise 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:

$$\text{Tree Impurity (TIMP)} = \frac{2(e-n+1)}{(n-1)(n-2)} \cdot 100 \tag{6}$$

where

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

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

3.3.2. Normalised Count of Levels (CLEV)

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

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

$$\text{Levels (CLEV)} = \frac{\#(N_{\equiv})}{\#N} \cdot 100 \tag{7}$$

where

$N_{\equiv}$ is the partition induced on the set of non-terminals N by the equivalence relation $\equiv$ defined in section 2.

A low value here suggests that the non-terminals are clustered into a few equivalence classes, and that these are logical choices for modularisation. A higher value means that the grammar

is more evenly spread out between the non-terminals, and there should be more opportunities for modularising the grammar.

3.3.3. Number of Non-Singleton Levels (NSLEV)

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

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

$$\text{Non-Singleton Levels (NSLEV)} = \#\{n \in N_{\equiv} \mid \#n > 1\} \tag{8}$$

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

3.3.4. Size of Largest Level (DEP)

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

$$\text{Size of Largest Level (DEP)} = \max\{\#n \mid n \in N_{\equiv}\} \tag{9}$$

3.3.5. Maximum Height (HEI)

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

$$\text{Maximum Height (HEI)} = \max\{i \mid N_1 \rhd N_2 \rhd \ldots \rhd N_i\} \tag{10}$$

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

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

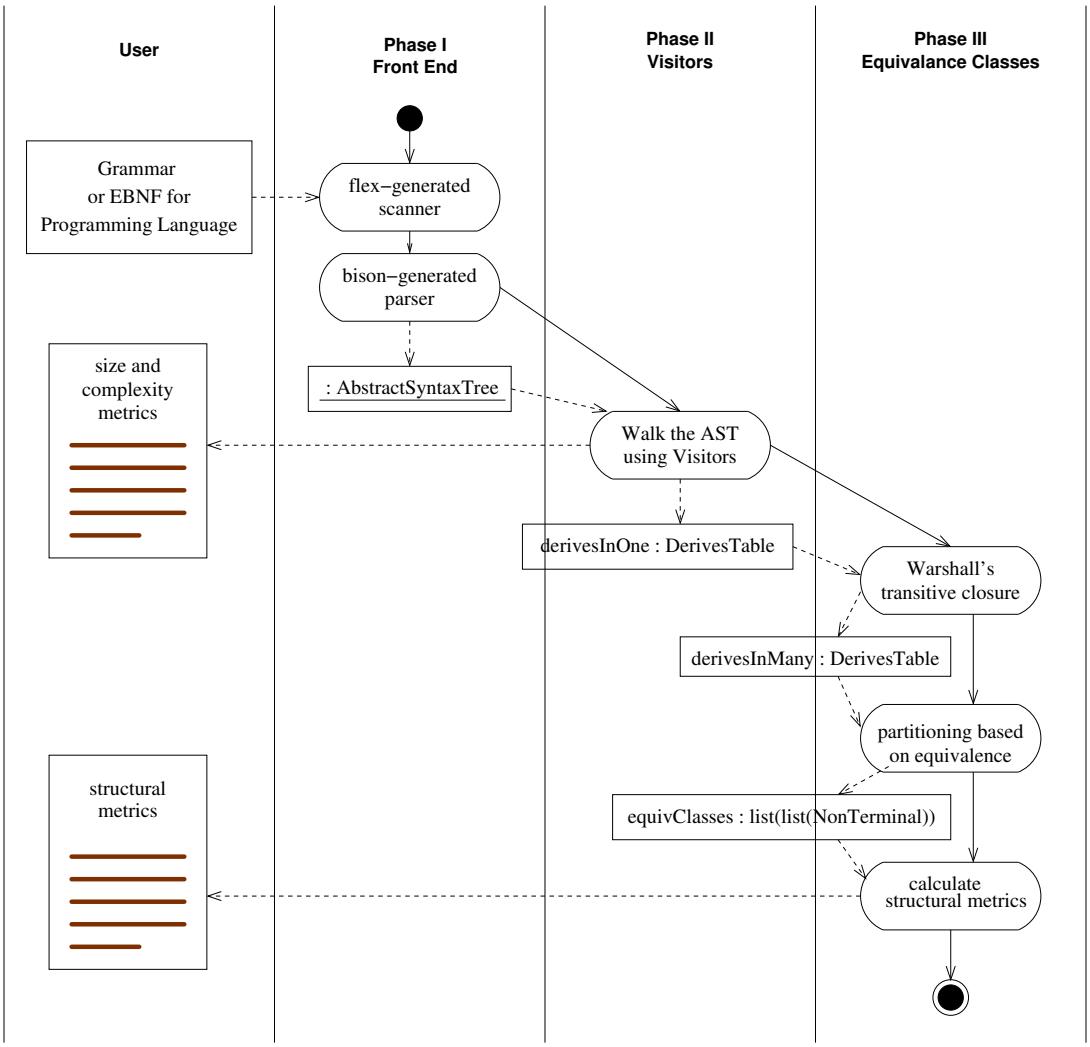

Figure 1. *A behavioural overview of the tool.* This UML activity diagram provides an overview of SynQ, the tool that we constructed to compute the ten metrics. Input to SynQ, indicated in the upper left corner, is the EBNF for the grammar under consideration. SynQ's output is the result of the computed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

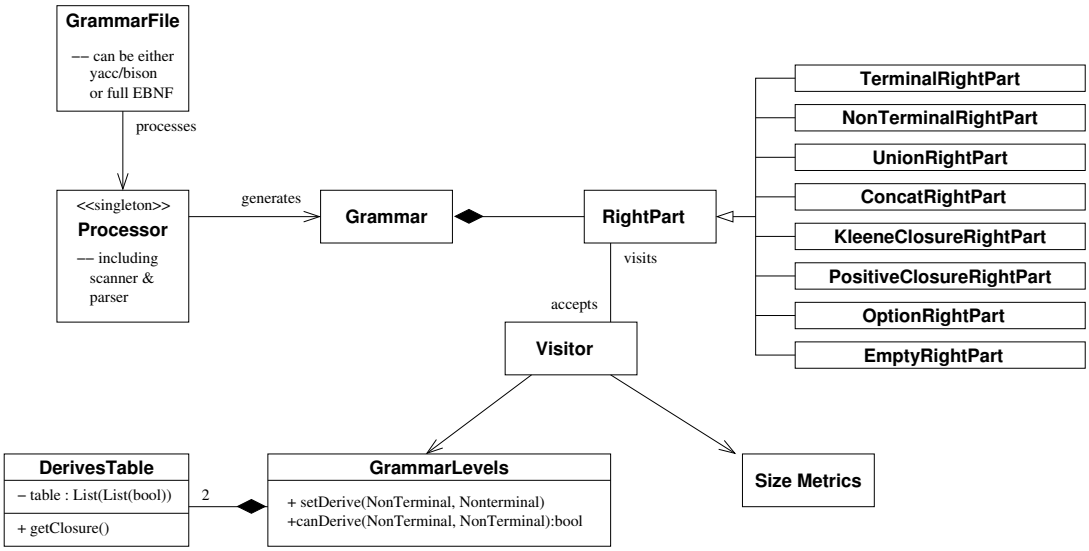

Figure 2. *An architectural overview of* SynQ. This UML class diagram depicts the main classes used in SynQ. The central entity is the Grammar class, which represents a grammar rules as a mapping from a non-terminal to an instance of the RightPart class

4. SynQ: A Tool to Calcuate Syntactic Metrics

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

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

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

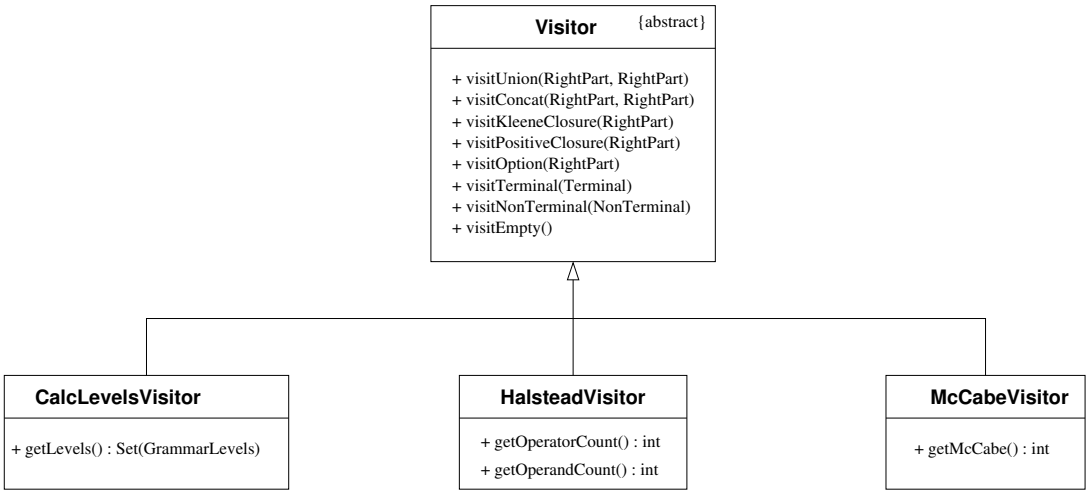

Figure 3. *The Visitor Classes.* This UML class diagram presents some of the classes in our Visitor hierarchy. The abstract base class Visitor is subclassed for each metric that is calculated.

4.1. The role of Visitors

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

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

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

4.2. Representing Grammatical Levels

From Figure 1, we can see that phase III of SynQ 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.

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

We traverse this matrix, applying Warshall's transitive closure algorithm to produce a second instance of DerivesTable, representing the closure of the first. This is basically a matrix S^*, where now $S^*[A, B]$ is true precisely when $A \rhd^* B$. The tree 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 SynQ represents the list of grammatical levels. Each grammatical level consists of a set of non-terminals where any 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. These levels can thus be readily calculated from the S^* matrix. The remaining structural metrics are generated directly from this list of grammatical levels.

5. Measurement of Programming Languages

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

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

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

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 first sub-section below, we present our results for those metrics that describe the size and complexity of the programming language syntax. In the second subsection below, 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.

Table I. *Size and complexity metrics.* This table shows the results of applying the metrics to measure the overall size and complexity of the four programming languages. 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

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 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 relativised 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#.

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

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

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

The three non-singleton levels for Java reflect categories for expressions, statements and types. The size of the largest level, reflected by the depth metric, DEP, corresponds to the number of non-terminals in the level for Java expressions. The other two non-singleton levels

for Java contain 25 non-terminals for statements and 4 non-terminals for types. A similar picture emerges with C#, where these three levels are augmented with two other small levels, one each for class and namespace member declarations.

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 is the cardinality of the only non-singleton grammatical level for this grammar. This is the least modular of all the grammars, where non-terminals for types, declarations, statements and expressions are combined into a single large grammatical level containing 86% of the non-terminals in the grammar.

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

6. Case Study: The evolution of the GNU C++ Parser

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

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

Table III. *The evolution of the GNU C++ parser.* This 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.

	Feb 1992	Mar 1992	Jun 1992	Dec 1992	Jun 1993	Jan 1994	Nov 1994	Nov 1995	Jan 1998	Jul 1999	Jun 2001
version	**2.0**	**2.1**	**2.2.2**	**2.3.3**	**2.4.5**	**2.5.8**	**2.6.3**	**2.7.2**	**2.8.0**	**2.95**	**3.0**
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

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 February 22, 1992, and version 3.0, released on June 18, 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.

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

Prepared using smrauth.cls

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

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

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

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

7. Related Work

In this section we review previous research that relates to our efforts at quantifying grammatical complexity and applying this quantification to the maintenance of grammar-based software applications. There has been considerable work directed at estimating the complexity of the maintenance effort [5, 6, 7, 8], and the incorporation of semantic considerations into this estimate [11, 12]; however, none of the previous research on metric complexity has targeted grammar-based applications. The work on grammar complexity was initially reported in references [22, 40] and extended in reference [20]; we begin by reviewing the work in references [20] and [40]. There has been some important work on parsing complexity that relates to our work and we review the research described in reference [41]. Finally, a preliminary version of this work was presented in reference [42]; we compare our current work reported in this paper to the preliminary report.

The definitions of VAR, PROD, LEV, DEP and HEI were introduced in references [22, 43, 44, 45] and we reviewed them in Section 2. These complexity measures classify context-free languages according to the size or structural properties of their grammars. The size of grammars are expressed as the number of terminals, VAR, and the number of productions, PROD. The number of grammatical levels LEV, the maximal number of elements of grammatical levels DEP, and the length of the digraph of grammatical levels HEI, are the complexity measures reflecting the structure of grammars. An important aspect of complexity theory is the study of the functional behaviour of the complexity measures on language classes. Complexity measures are functions defined on context-free languages, where the function values are natural numbers. The main result of reference [20] establishes the unboundedness of complexity measures VAR, PROD, LEV, DEP and HEI on the classes of languages defined by grammar forms.

Reference [41] presents algorithms for corpus-based parsing and several metrics are described for evaluating the algorithms. Input to corpus-based parsing is a *treebank*, a collection of text annotated with the "*correct*" parse tree. The goal is to find algorithms that, given unlabelled text from the treebank, produce a parse that is similar to the one in the treebank. Evaluation metrics are used to determine whether a candidate parse matches the correct parse; the metrics include, among other criteria, N_G, the number of nonterminals in the guessed parse tree, and N_C, the number of nonterminals in the correct parse. Some of the metrics used in reference [41] are similar to those described in references [22, 43, 44, 45] and [20], but most are based on parse trees rather than grammars.

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

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

8. Concluding Remarks

In this paper, we have adapted the software metrics that are commonly used to measure program complexity and to apply them to the measurement of the complexity of programming language syntax. We have formally defined a suite of ten metrics measuring grammar size

and structure. Measuring grammars in this way extends the field of software metrics to cover grammar-based applications such as compilers and program comprehension tools.

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

Using our metrics, together with the metrics described by other researchers [22, 43, 44, 45, 40, 20], we have shown a correlation between the computed results of the metrics and anecdotal reports about language and the difficulty of parser construction. By using our metrics to track the evolution of the GNU C++ parser we have demonstrated that they can usefully distinguish major and minor version changes, as well as measure the impact of those changes.

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

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

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

REFERENCES

1. Schach SR. *Object-Oriented and Classical Software Engineering*, McGraw-Hill: New York NY, 2001; 648 pp.
2. Martin J, McClure CL. *Software Maintenance: The Problem and its Solutions*, Prentice-Hall: Englewood Cliffs, NJ: Englewood Cliffs NJ, 1983; 472 pp.
3. Pigoski TM. *Practical Software Maintenance: Best Practices for Managing your Software Investment*, Wiley: New York NY, 1997; 400 pp.
4. Fenton NE, Pfleeger SL. *Software Metrics: A Rigorous and Practical Approach*, International Thomson Publishing: London, England, 1998; 656 pp.
5. Abran A, Silva I, Primera L. Field studies using functional size measurement in building estimation models for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 2002; **14**(1): 31–64.
6. Ahn Y, Suh J, Kim S, Kim H. The software maintenance project effort estimation model based on function points. *Journal of Software Maintenance and Evolution: Research and Practice* 2003; **15**(2): 71–85.

7. Harrison MS, Walton GH. Identifying high maintenance legacy software. *Journal of Software Maintenance and Evolution: Research and Practice* 2002; **14**(6): 429–446.
8. Polo M, Piattini M, Ruiz F. Using code metrics to predict maintenance of legacy programs: a case study. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society: Los Alamitos, CA, 2001; 202–211.
9. Briand LC, Wust J, Lounis H. Using coupling measurement for impact analysis in object-oriented systems. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society: Los Alamitos, CA, 1999; 475–482.
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.
13. Malloy BA, Gibbs TH, Power JF. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software, Practice & Experience* 2002; **33**(1): 19–39.
14. Reps T, Teitelbaum T, Demers A. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems* 1983; **5**(3): 449–477.
15. Panda PR, Catthoor F, Dutt ND, Danckaert K, Brockmeyer E, Kulkarni C, Vandercappelle A, Kjeldsberg PG. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems* 2001; **6**(2): 149–206.
16. Rabbah RM, Palem KV. Data remapping for design space optimization of embedded memory systems. *ACM Transactions on Embedded Computing Systems* 2003; **2**(2): 186–218.
17. Maletic J, Collard M, Marcus A. Source code files as structured documents. *Proceedings 10th International Workshop on Program Comprehension*. IEEE Computer Society: Los Alamitos, CA, 2002; 289–292.
18. Irwin W, Churcher N. *A generated parser of C++*. Christchurch, New Zealand, 2001. http://citeseer.nj.nec.com/irwin01generated.html [13 January 2003].
19. Aho A, Sethi R, Ullman J. *Compilers: Principles, Techniques and Tools*, Addison-Wesley: Boston, MA, 1986; 500 pp.
20. Csuhaj-Varjú E, Kelemenová A. Descriptional complexity of context-free grammar forms. *Theoretical Computer Science* 1993; **112**(2): 277–289.
21. McCabe TJ. A complexity measure. *IEEE Transactions on Software Engineering* 1976; **2**(4): 308–320.
22. Brauer W. On grammatical complexity of context-free languages. *Proceedings Symposium and Summer School on the Mathematical Foundations of Computer Science*. Mathematical Institute of the Slovak Academy of Sciences: Czechoslovakia, 1973; 193–196.
23. Elder J. *Compiler Construction: A Recursive Descent Model*, Prentice-Hall: Englewood Cliffs, NJ, 1994; 437 pp.
24. Halstead M. *Elements of Software Science*, Elsevier Science Inc.: New York, NY, 1977; 127 pp.
25. Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language User Guide*, Object Technology Series, Addison-Wesley: Boston, MA, 1998; 482 pp.
26. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley: Boston, MA, 1995; 395 pp.
27. ISO/IEC JTC1/SC22/WG14. *International Standard: Programming Languages - C*, No. 9899:1999. International Organization for Standardization (ISO): Geneva, Switzerland, 1999; 538 pp.
28. Kernighan B, Ritchie D. *The C Programming Language*, Prentice-Hall: Englewood Cliffs, NJ, 1988; 274 pp.
29. ISO/IEC JTC1/SC22/WG21. *International Standard: Programming Languages - C++*, No. 14882:2003. International Organization for Standardization (ISO): Geneva, Switzerland, 2003; 786 pp.
30. Stroustrup B. *The C++ Programming Language*, Addison-Wesley: Boston, MA, 1997; 911 pp.
31. Gosling J, Joy B, Steele G. *The Java Language Specification*, Addison-Wesley: Boston, MA, 2000; 544 pp.
32. ECMA. *C# language specification*, No. ECMA-334. European Computer Manufacturers Association: Geneva, Switzerland, 2002; 471 pp.
33. Bodin F, Beckman P, Gannon D, Gotwals J, Narayana S, Srinivas S, Winnicka B. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. *Proceedings 2nd annual object-oriented numerics conference*. Rogue Wave Software: Corvallis, OR, 1994; 122–136.
34. Knapen G, Lague B, Dagenais M, Merlo E. Parsing C++ despite missing declarations. *Proceedings 7th International Workshop on Program Comprehension*. IEEE Computer Society: Los Alamitos, CA, 1999;

114–125.
35. Lilley J. *PCCTS-based LL(1) C++ parser: Design and theory of operation*, version 1.5. Westminster, CO, 1997. http://www.empathy.com/pccts [13 January 2003].
36. Power JF, Malloy BA. Symbol table construction and name lookup in ISO C++. *Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems*. IEEE Computer Society: Los Alamitos, CA, 2000; 57–68.
37. Reiss S, Davis T. *Experiences writing object-oriented compiler front ends*, Brown University: Providence, RI, 1995; 18 pp.
38. Roskind J. *A YACC-able C++ 2.1 grammar, and the resulting ambiguities*, release 2.0. Indialantic, FL, 1991. ftp://ftp.iecc.com/pub/file/c++grammar [13 January 2003].
39. Malloy BA, Linde SA, Duffy EB, Power JF. Testing C++ compilers for ISO language conformance. *Dr. Dobbs Journal* 2002; **27**(6): 71–80.
40. Ginsburg S, Lynch N. Size complexity in context-free grammar forms. *Journal of the Association for Computing Machinery* 1976; **23**(4): 582–598.
41. Goodman J. Parsing algorithms and metrics. *Proceedings 34th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics: East Stroudsburg, PA, 1996; 177–183.
42. Power JF, Malloy BA. Metric-based analysis of context-free grammars. *Proceedings 8th International Workshop on Program Comprehension*. IEEE Computer Society: Los Alamitos, CA, 2000; 171–178.
43. Gruska J. Some classifications of context-free languages. *Information and Control* 1969; **14**: 152–179.
44. Kelemenová A. Complexity of normal form grammars. *Theoretical Computer Science* 1984; **28**(3): 299–314.
45. Kelemenová A. Structural complexity measures on grammar forms. *Proceedings 2nd Conference on Automata, Languages and Programming Systems*. Springer-Verlag: Heidelberg, Germany, 1988; 73–76.
46. Briand L, Daly J, Wust J. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering* 1999; **25**(1): 91–121.
47. Weyuker E. Evaluating software complexity measures. *IEEE Transactions on Software Engineering* 1988; **14**(9): 1357–1365.
48. van Deursen A, Klint P, Visser J. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices* 2000; **35**(6): 26–36.

AUTHORS' BIOGRAPHIES

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

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