On automata and language based grammar metrics

Matej Crepinsek, Tomaz Kosar, Marjan Mernik, Julien Cervelle, Rémi Forax, Gilles Roussel

UDC 004.4, DOI: 10.2298/CSIS1002309C

On Automata and Language Based Grammar Metrics

Matej Črepinšek[1], Tomaž Kosar[1], Marjan Mernik[1],
Julien Cervelle[2], Rémi Forax[2], Gilles Roussel[2]

[1] University of Maribor, Faculty of Electrical Engineering and Computer Science,
Smetanova 17, 2000 Maribor, Slovenia
Email: {matej.crepinsek, tomaz.kosar, marjan.mernik}@uni-mb.si
[2] Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge,
77454 Marne-la-Vallée, France
Email: {julien.cervelle, remi.forax, gilles.roussel}@univ-mlv.fr

Abstract. Grammar metrics have been introduced to measure the quality and the complexity of the formal grammars. The aim of this paper is to explore the meaning of these notions and to experiment, on several grammars of domain specific languages and of general-purpose languages, existing grammar metrics together with the new metrics that are based on grammar LR automaton and on the language recognized. We discuss the results of this experiment and focus on the comparison between grammars of domain specific languages as well as of general-purpose languages and on the evolution of the metrics between several versions of the same language.

Keywords: grammar metrics, software language engineering, grammar engineering, grammarware.

1. Introduction

Grammar metrics were introduced to measure the quality and complexity of a given grammar in order to orient grammar engineering (grammarware [17]). We consider that existing metrics [23], more or less deduced from classical program metrics or from the structure of the specification, could be upgraded with new metrics that are specific to grammar behavior and could provide additional insights about the complexity of the grammar and the language generated by this grammar. Of course, a single metrics alone cannot capture the quality of the grammar, however a set of well chosen metrics could give interesting hints to grammar developers.

In order to complete the existing set of metrics, we propose two different kinds of metrics[3]. A first set of metrics is computed from the LR automaton,

[3] This work is sponsored by bilateral project "Advanced Topics in Grammar Engineering" (code BI-FR/08-09-PROTEUS-008) between Slovenia and France.

generated from the grammar. A second set is more related to the language recognized, than to the grammar itself. These different kinds of metrics produce results that are complementary for the grammar developers.

In order to compute these metrics, we have developed a tool. As an input, it takes ANTLR [22] or Tatoo [4, 5] grammars and computes classical metrics [23] together with our new metrics. It uses Tatoo engine to construct the LR automaton for these grammars.

Using this tool we have computed the values of these metrics on several grammars that form a good benchmark of grammars. These grammars cover domain specific languages (DSL [20]) and general-purpose languages (GPL [28]). They also cover the evolution of a grammar between different versions of the language. From these experimentations, we discuss the different values of the metrics.

The structure of the paper is as follows. Presented in the Section 2 is related work and existing metrics. In Section 3, the new metrics are defined. Section 4 describes the tool and how it is linked to Tatoo. In Section 5, experimental results on the grammars are detailed and discussed. In Section 6, some usage guidelines are presented. Section 7 carries conclusions and remarks. In the appendix, the computation of the closure application of rules is detailed.

2. Overview of Related Work

In the field of grammar metrics, only a few tools and papers exist. The most pertinent of these tools is *SynC tool* by Power and Malloy [23]. In *SynC tool*, grammar metrics are divided into size and structural metrics. In the first metrics group, an adaptation of standard metrics for programs [13], the following grammar size metrics are defined [23]:

- `term` – number of terminals,
- `var` – number of non-terminals,
- `mcc` – McCabe cyclomatic complexity,
- `avs` – average size of right hand side, and
- `hal` – Halstead effort.

Size metrics feature useful information about the grammars. More maintenance is expected for grammars with large numbers of non-terminals (`var`). The `mcc` provides the number of alternatives for non-terminals. The `mcc` value indicates the effort required for grammar testing and a greater potential for parsing conflicts. A big `avs` value points to less readable grammar as well as it impacts on the parsers' performance, because symbols have to be placed on the parser stack. The `hal` value evaluates grammar designers' efforts to understand the grammar.

Structural metrics for grammars are derived from grammatical levels [8], where a grammar is represented as a graph. In the graph, the nodes are non-terminals and the edges represent a successor relationship between a left hand

side non-terminal and a non-terminal on the right hand side. In order to compute structural metrics, we compute the strongly connected components of the graph, which leads to a partition of the set of non-terminals into *grammatical levels*. We use the following structural metrics, as defined in [23]:

- `timp` – tree impurity,
- `clev` – normalized counts of levels,
- `nslev` – number of non-singleton levels,
- `dep` – size of largest level, and
- `vhei` – Varju height metrics.

Tree impurity (`timp`) measures how much the graph resembles a tree (0% – graph is a tree, 100% – graph is fully connected). A high `timp` value for a grammar means that refactoring, the grammar will be complicated, since a change in one rule may impact many other rules. A normalized count of levels (`clev`) is the normalization of the number of grammatical levels by the total number of non-terminals expressed in percentage. A high `clev` indicates more opportunities for grammar modularization. Many of the equivalence classes are of size 1, while language concepts such as declarations, expressions, and commands tend to be represented by larger classes. The (`nslev`) metrics identifies the number of such classes. The size of the largest level (`dep`) metrics measures the number of non-terminals in the largest grammatical level. A high `dep` indicates an uneven distribution of the non-terminals among grammatical levels. The Varju height metrics (`vhei`) is the maximum distance of any non-terminal from the start symbol, and is expressed as a percentage of the number of equivalence classes.

Paper [3] presents a methodology for iterative grammar development. Well-known techniques from software engineering are applied to the development

- version control,
- grammar metrics,
- unit testing, and
- test coverage analysis.

It demonstrates how these techniques can make grammar development a controlled process. As mentioned above, one of the techniques used involves grammar metrics. Authors use size and structural metrics defined in [23] and extend them with disambiguation metrics, which are SDF [15] specific:

- `frst` – number of follow restrictions,
- `rejp` – number of reject productions,
- `assoc` – number of associativity attributes, and
- `upp` – number of unique productions in priorities.

These metrics merely count the various types of disambiguation in the SDF notation. Aside from Halstead's effort metrics, some of the ingredient metrics and related metrics are presented and used for grammar engineering. In a

similar manner, as done in this work, we propose new metrics, which brings additional insights into grammar development.

One of the applications for grammar metrics is also in the field of grammar testing. The concept of grammar testing is explained in [18]. This paper presents context-dependent branch coverage on parser testing and grammar recovery; it proposes new tests for checking the accuracy and the completeness of grammars. We believe that our grammars' metrics could be a valuable contribution to the field of grammar testing, used as effort estimation in grammar engineering (i.e., software engineering applied to grammars).

3. Proposed New Metrics

In this section, we describe in detail two new kinds of metrics, LR table-based metrics and generated-language based metrics.

3.1. LR Table Metrics

The first set of metrics is based on the LR automaton that is used to produce an efficient bottom-up parsers for the grammar, but could also simulate top-down parsing [25], comparable to LL parsers. It is surprising that information given by this automaton has never been used before to qualify grammars.

The LR states are built using the following algorithm. A more detailed description of this algorithm can be found in [2].

First, the grammar is increased by adding a new production

$$X \to S\,\mathsf{EOT}$$

where S is the start symbol of the grammar, X a fresh non-terminal which becomes the new axiom and EOT, a fresh terminal which symbolizes the end of input.

$$E \to (E)|E + E|E - E| - E|\mathrm{id}$$

Fig. 1. Grammar G_1

States of the LR automaton are defined by a set of items. An item is a production where an inter-letter space is marked (usually with a dot) on the right-hand side. The set of items defines all of the productions that can be found at this stage of the parsing. For instance, for the grammar G_1 described in Fig. 1, after reading «$(E + E$» (the E means that a word derived from E is recognized) the state contains:

$$E \to E \cdot + E$$
$$E \to E \cdot - E$$
$$E \to E + E \cdot$$

The item $E \to E + E \cdot$ indicates that «$E + E$» has been read and will be considered as a single E, while the item $E \to E \cdot - E$ indicates that the second E is part of an $E - E$ expression that will be considered as a single E.

Note that the information that a «(» has been read, is kept in the state stack of the parser, not in the LR state.

Only the initial state contains the item:

$$X \to \cdot S \, \mathsf{E} \mathsf{O}_\mathsf{T}$$

States are built by applying a creation rule to existing states, until a new state cannot be built.

To explain this rule, we first define the closure $\mathcal{C}(I)$ of an item I as the smallest set to verify the following set of equations, where $\mathcal{P}$ is the set of productions of the grammar:

- $I \in \mathcal{C}(I)$;
- $\forall E \to \alpha \cdot X\beta \in \mathcal{C}(I)$ and $\forall X \to \gamma \in \mathcal{P}$, then $X \to \cdot\gamma \in \mathcal{C}(I)$.

The closure of a state is defined as the union of the closure of its items.

Then, new states are built from a state St by applying the following rule: for each terminal or non-terminal $\mathfrak{v}$ such that an item $X \to \alpha \cdot \mathfrak{v}\beta$ is in $\mathcal{C}(\mathrm{St})$, the following state is created

$$\{Y \to \delta\mathfrak{v} \cdot \zeta \,|\, Y \to \delta \cdot \mathfrak{v}\zeta \in \mathcal{C}(\mathrm{St})\}$$

If $\mathfrak{v}$ is a terminal, we say that the state St can shift the terminal $\mathfrak{v}$.

The set of LR states for the grammar G_1 computed using previous algorithm is the following:

$$\{X \to \cdot E \, \mathsf{E} \mathsf{O}_\mathsf{T}\}$$
$$\{X \to E \, \mathsf{E} \mathsf{O}_\mathsf{T} \cdot\}$$
$$\{E \to (\cdot E)\}$$
$$\{E \to - \cdot E\}$$
$$\{E \to \mathrm{id} \cdot\}$$
$$\{E \to (E) \cdot\}$$
$$\{X \to E \cdot \, \mathsf{E} \mathsf{O}_\mathsf{T},\ E \to E \cdot + E,\ E \to E \cdot - E\}$$
$$\{E \to E + \cdot E\}$$
$$\{E \to E - \cdot E\}$$
$$\{E \to (E \cdot),\ E \to E \cdot + E,\ E \to E \cdot - E\}$$
$$\{E \to -E \cdot,\ E \to E \cdot + E,\ E \to E \cdot - E\}$$
$$\{E \to E + E \cdot,\ E \to E \cdot + E,\ E \to E \cdot - E\}$$
$$\{E \to E - E \cdot,\ E \to E \cdot + E,\ E \to E \cdot - E\}$$

In the last state, for instance, the terminals $+$ and $-$ can be shifted.

From the possible metrics that can be extracted from the LR automaton we have chosen the following:

- The metrics `lrs` represents the number of states in the LR automaton. This number is 13 for the grammar G_1. This metrics captures the complexity of the grammar.
- The `lat` metrics sums, for each terminal in the grammar, the number of states in the LR automaton that does not lead to an error when this terminal is in the lookahead, and it is normalized by the number of terminals. This metrics gives an idea for each terminal of the probability to accept it during the parsing.
- Metrics `lat` can be further normalized by the number of states `lrs`. Metrics `lat/lrs` computes complexity of relations between the terminals and the states in LR tables.
- The metrics `lrtla` sums, for each state in the automaton, the number of terminals that, when they are in the lookahead, do not lead to an error in this state, and it is normalized by the number of states. It indicates the complexity of each state in the automaton. If we normalize metrics `lrtla` by the number of terminals we get metrics `lat/lrs`.
- The metrics `lcc` counts the number of LR conflicts. These conflicts are shift-reduce or reduce-reduce conflicts found in some of the states. The `lcc` metrics gives an insight into the complexity of the grammar. Indeed, the conflicts solved by priority/associativity rules could also be solved by rewriting the grammar instead of introducing unnecessary productions that degrade the readability of the grammar [2]. Note that conflicts may be implicitly resolved by many compiler generators. A shift-reduce conflict is resolved implicitly by choosing to shift over reduce. On the other hand a reduce-reduce conflict is resolved implicitly by choosing to reduce the rule that first appears in the grammar. However, every such conflict should be carefully studied and checked if default behavior is indeed appropriate. Conflicts certainly raise the complexity of the grammar and `lcc` metric captures it.

3.2. Generated Language Metrics

The second set of proposed metrics is based on some of the characteristics of the language recognized. The following metrics, discussed in more detail below, are proposed: `ss`, `ssm`, `ltps`, `ltpsm`, `ltpsa`, and `ltpsn`.

The metrics `ss` builds, for each production, the shortest sample that uses it and stores the average size of these samples. This metrics provides hints of the verbosity of the language produced by the grammar. The metrics `ssm` is the maximum size of the samples.

The shortest sample for a production is produced using a recursive algorithm. More precisely, the algorithm for computing the shortest sample using a production in a grammar consists in three steps. The first is the computation, from each non-terminal, of the shortest word made of terminals derived from

this non-terminal. The second step is the computation, for any non-terminal N, of the shortest word generated by an axiom, that is only made of terminals and one occurrence of N, which we call the sequel of the *shortest word leading to* X. Details of these first two steps can be found in appendix. Once these first two steps are accomplished, to get the shortest sample using production $X \to \alpha$, one starts with the word w obtained in step two for non-terminal X, replace X by α in w and finally replace all remaining non-terminals with the shortest words computed in step one.

The word produced is still the shortest since, if a shorter one exists, either its derivation tree would lead to a shortest way to produce a word that contains X or the shortest words for non-terminals of α. Since the first two steps are done using a closure operation on rules, if only the shortest sample for one single production needs to be computed, one can save computation time using a lazy and dynamic programming style, as it is done in Tatoo.

For instance for grammar G_1, the set of the shortest samples produced by this algorithm is:

$$\{(\mathrm{id}), \mathrm{id} + \mathrm{id}, \mathrm{id} - \mathrm{id}, -\mathrm{id}, \mathrm{id}\}$$

The average size of these samples is $\mathtt{ss} = 2.4$, while the maximum size of the shortest sample is $\mathtt{ssm} = 3$.

The other metrics are only concerned with the sequences of two terminals (terminal pairs) that may be found in the language recognized by the grammar.

$$\begin{aligned} S &\to L|L.L \\ L &\to B|L\,B \\ B &\to 0|1 \end{aligned}$$

Fig. 2. Grammar G_2

For instance, the grammar for Knuth's binary numbers, described in Fig. 2, allows 8 different terminal pairs. Combinations are presented in Table 1, where the first column and the first row represent all grammar terminals (ter) and `true` or `false` on position (ter_i, ter_j) indicate that, there exists a sentence recognized by this grammar which contains the pair of terminals (ter_i, ter_j).

Table 1. Allowed terminal pairs

i/j	0	1	.
0	true	true	true
1	true	true	true
.	true	true	false

From the table of allowed terminal pairs, we have defined four different metrics:

- The metrics `ltps` computes the number of different terminal pairs acceptable in the language. In case of G_2 value of `ltps` metrics is 8.
- The metrics `ltpsm` computes the maximum number of different pairs for one terminal. In case of G_2 value of `ltpsm` metrics is 3, because after terminal 1 one can find three different terminals (the same as in the case of terminal 0).
- The metrics `ltpsa` computes, given a terminal, the average number of terminals that can directly follow this terminal. In case of G_2 the value of `ltpsa` metrics is $(3+3+2)/3 \approx 2.666$.
- The metrics `ltpsn` normalizes the metrics `ltps`, by the number of possible combinations of terminals and is presented as a percentage. In the case of G_2 the value of `ltpsn`, the metrics is $(3+3+2)/9 \approx 88.888\%$.

Table 1 is calculated directly from the grammar by computing, for all non-terminal X, the sets of first $F(X) = \{a | X \Rightarrow^* a\beta\}$ and last $L(X) = \{a | X \Rightarrow^* \beta a\}$ possibly derived terminals, where a is a terminal. From these sets, it is easy to calculate pairs from the right hand sides of productions. For all occurrences of two consecutive terminals or non-terminals $\mathfrak{v}_1$ and $\mathfrak{v}_2$, one adds all of the pairs of $L(\mathfrak{v}_1)F(\mathfrak{v}_2)$ where $L(a) = F(a) = a$ in case a is a terminal.

4. Tool Description

In this section, we present the *gMetrics* tool. This tool extracts information from grammars and calculates the metrics proposed by Power and Malloy [23] as well as the new ones, LR-based metrics and generated language-based metrics, proposed in the previous section. The global activity diagram of gMetrics is presented in Fig. 3.

The main objective of this tool is to extract from input grammars as much information as possible and perform as few modifications as possible in relation to the original grammar. Indeed, we would like to avoid potential metric disturbance and to process all metrics from the same specification.

We currently support the formats of compiler construction tools ANTLR version 3 [22] (an LL parser generator) and Tatoo [5] (a LR parser generator). The metrics are divided, as explained before, into four categories: size metrics, structural metrics, LR automaton-based metrics and generated language-based metrics. In practice, to reuse an existing grammar specification, one must take into account the grammar form (BNF, EBNF, CNF, etc.), the grammar type (LL, LR, LALR, IELR [9], etc.), the file format (a tool mainly dependent and potentially customized with semantics and other annotations) and the version of the tool. In this work, we limited ourselves to grammar specifications used by ANTLR and Tatoo. Moreover, automata-based metrics are calculated from LR automaton, despite that a particular grammar can be of different type (e.g.,

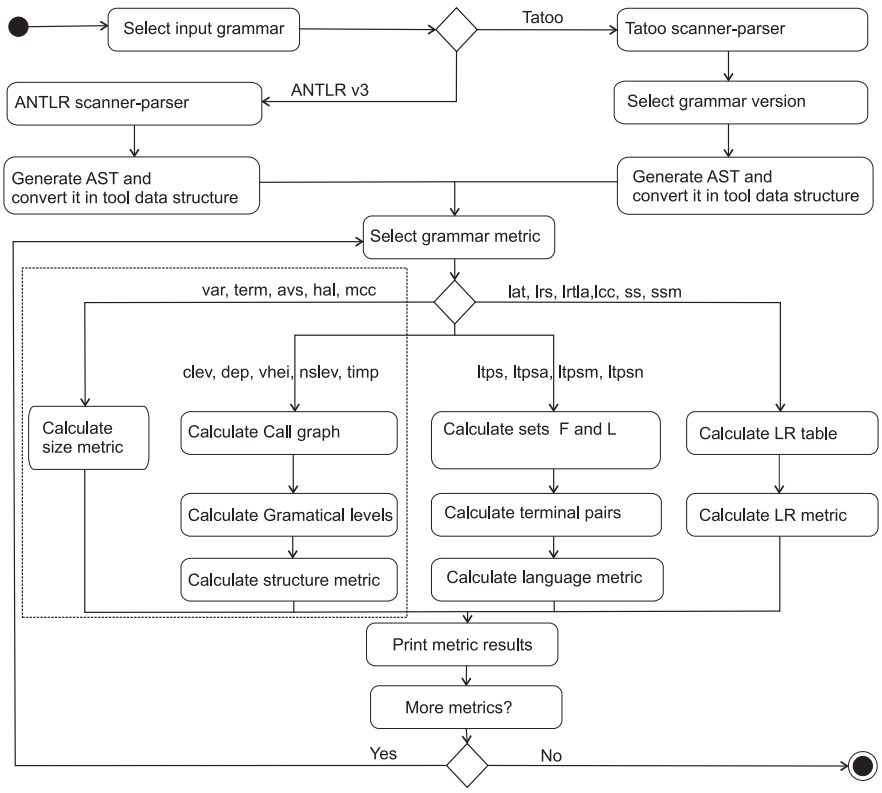

Fig. 3. Activity diagram

LL, LALR, IELR). Although, we report the number of shift-reduce and reduce-reduce conflicts that indicate that a grammar might not be the LR grammar (some conflicts are resolved by additional rules in parser generators). Our future work lies in identifying the correct type of grammar while calculating the automata-based metrics, as well to analyze the conflicts, as presented in [9].

Nevertheless, some transformations are unavoidable because the original input format of the grammars is different and some metrics make use of specific algorithms that require constrained input. However, `gMetrics` minimizes such transformations.

To solve the problem of a unique grammar representation, we have chosen to use an in-memory intermediate form close to an EBNF notation.

Indeed, even if neither ANTLR nor Tatoo uses complete EBNF form as input, the input format of each parser generator is close to this notation. Moreover, this format has the advantage to avoid to choose between left of right recursion in the specification, since lists may be specified using star ('*') or plus ('+') constructions.

The ANTLR format has been selected because it provides a great deal of interesting existing grammars and Tatoo was chosen because of its open architecture that facilitates the computation of some of the metrics. Both tools use priorities or/and associativity rules for solving automata conflicts.

In the future, we plan to broaden our tool to support other input formats (e.g., LISA [21]). Meanwhile, users may use this tool as a Java application library. In this case, users need to implement the `IGrammar` interface, which describes grammar in our EBNF internal form.

Because ANTLR and Tatoo are both implemented in Java, the simplest choice was to implement *gMetrics* in Java. The first challenge was to create and to fill internal data structure from ANTLR and Tatoo grammar specification. More precisely, in Tatoo, we directly used the memory representation exported by Tatoo. Moreover, since Tatoo supports grammar versioning, one input grammar may include several versions (usually specified in different grammars).

The metrics implementations are divided into four groups: size metrics, structural metrics, LR-based metrics and language-based metrics.

- To calculate size metrics, we implement a visitor pattern that counts different grammar properties.
- To compute structural metrics, the call graph is derived from the productions. It is then used to calculate grammatical levels. The structure metrics [8] are deduced from this information.
- To construct the LR automaton, information about the grammar associativeness (left or right) is first established. Next, the LR table is computed by the Tatoo, engine together with the associated LR actions.
- For the language-based metrics, terminal pairs are computed, as explained in the previous section.

When dealing with metrics implementation, we try to provide as much information for interpretation as possible. For most of the metrics we provide histograms, which are used to calculate concrete metrics values. These histograms can also be used for the computation and the analysis of different statistical values.

For example, in Fig. 4 we present the metric `ltps` histogram for the ANSI C grammar. The metric `ltps` describes, for a given terminal, the number of different terminals that can follow it. In this histogram one can notice that many terminals (34 in y axle) have between 11 and 20 (x axle) terminals that can follow them and only one terminal may be followed by almost all terminals. With this histogram one can monitor the introduction of a new terminal in the grammar.

The *gMetrics* tool is an open source project and can be found on the following web page http://code.google.com/p/cfgmetrics.

5. Empirical Study on Metrics for GPL and DSL Grammars

The grammar samples used for this experiment come from examples drawn from the ANTLR [22] samples and from several versions of Java grammars for

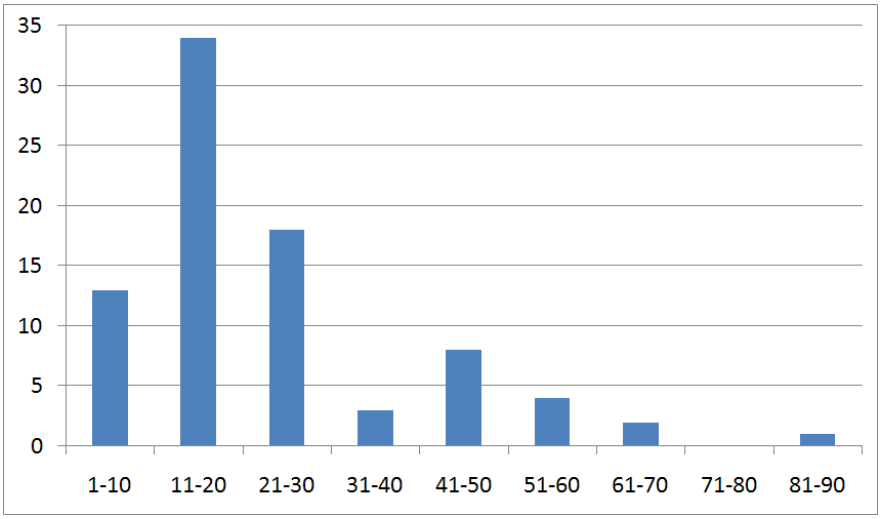

Fig. 4. Metrics `ltps` histogram for the ANSI C grammar

Tatoo [4, 5]. They cover domain-specific languages and general-purpose languages. These grammar examples are representative of current practice in grammar-ware engineering and can be considered as good benchmarks to evaluate the pertinence of metrics. They result from a collaborative work that usually involves several developers thereby ensuring their global quality.

More precisely, the DSL grammars studied are:

- EXPR, a grammar for arithmetic expressions [2].
- FDL, a grammar that enables the specification of sets of features [10].
- EBNF, a grammar for grammar specifications in Extended Backus Normal Form [1].
- CFDG, a grammar of a simple programming language for generating pictures [11].
- GAL, a grammar to describe video devices [26].
- ANTLR V3, a grammar for grammar definitions in ANTLR version 3 format [22].

General-purpose language grammars studied are those from

- Ruby 1.8.5 [27];
- ANSI C [14];
- Python 2.5 [19], and
- versions of Java [12] from 1.0 to 1.6.

The version 1.6 of the Java grammar comes from ANTLR samples, whereas the other versions come from Tatoo samples.

The results shown in Table 2 show the outcome of the size metrics for the different kinds of grammars. This table indicates that some of these metrics (e.g.,

Table 2. Results for classical metrics

Lang	term	var	mcc	avs	hal
Expr	9	5	1.6	4	1
FDL	14	6	2.17	6.5	2.63
EBNF	12	7	1.71	3.29	1.17
CFG Design	24	13	2.39	6	6.57
GAL	71	74	1.2	3.88	33.36
ANTLR V3	49	45	2.42	4.98	29.55
Ruby 1.8.5	88	83	2.61	4.74	54.44
Java 1.6	98	110	2.46	5.96	122.66
ANSI C	83	66	2.21	5.09	42.34
Python 2.5	85	86	2.22	4.93	63.41
Java 1.5	102	129	1.75	5.85	140.38
Java 1.4	100	116	1.75	5.8	118.21
Java 1.3	99	114	1.76	5.84	116.85
Java 1.2	99	114	1.76	5.84	116.85
Java 1.1	98	114	1.75	5.83	116.98
Java 1.0	98	112	1.63	5.38	98.54

`mcc`, `avg`) can not be used to differentiate DSL from GPL grammars. Some DSLs (e.g., GAL) are comparable to GPLs in relation to the number of terminals, nonterminals, and `hal` indicates that the size of such DSLs grammars can be comparable to GPLs.

It is noteworthy that all of the metrics for Java versions are extremely stable, with the exception of the `mcc` metrics, which is much larger for Java 1.6. This is without a doubt because it is an LL grammar, designed for ANTLR, whereas other versions of Java are LR, designed for Tatoo.

Table 3 provides the results of the structural metrics for the grammars. Among these results, it is interesting to note that the `clev` metrics indicates that the first versions of Java support have better modularization than the newer versions, probably due to new constructions such as internal classes. However, it is surprising that DSL, such as Expr, also has large values.

Most of the structural metrics are not very relevant where they concern the difference between DSL and GPL, with the exception of the `dep` metrics. However, there is marked variation between the values of this metrics between GPL, in particular in Java. This provides a good means for the interweaving of the grammar. From our point of view, these structural metrics are difficult to understand for grammar developers.

The results of Table 4 show that the values of the metrics `lrs` mainly depend on the type of the language. The DSL grammars have smaller values than GPL grammars, below 1000 states. Second, this metrics is not directly connected to the size of the grammar since grammars with similar numbers of terminals or non-terminals produce completely different values (e.g. Ruby vs. Python from

Table 3. Results for structural metrics

Lang	timp	clev	nslev	vhei	dep
Expr	56.25	60	1	3	3
FDL	32	83.33	1	1	2
EBNF	69.44	42.86	1	3	5
CFG Design	31.25	100	0	1	1
GAL	14.6	95.95	1	1	4
ANTLR V3	21.69	75.56	2	1	8
Ruby 1.8.5	62.37	37.35	1	1	53
Java 1.6	72.35	25.46	2	1	80
ANSI C	67.93	30.3	3	1	41
Python 2.5	44.39	46.51	3	1	35
Java 1.5	76.53	22.48	2	1	100
Java 1.4	59.59	40.52	2	1	69
Java 1.3	58.98	41.23	2	1	67
Java 1.2	58.98	41.23	2	1	67
Java 1.1	58.98	41.23	2	1	67
Java 1.0	33.29	65.18	3	1	24

Table 4. Results for LR based metrics

Lang	lrs	lat	lrtla	lat/lrs	lcc
Expr	59	18	3.05	0.31	0
FDL	115	20.13	2.63	0.18	0
EBNF	129	63.08	6.36	0.49	2
CFG Design	151	45.72	7.57	0.32	0
GAL	873	40.31	3.14	0.05	1
ANTLR V3	958	165.92	8.66	0.17	60
Ruby 1.8.5	13474	3509.2	23.18	0.26	7446
Java 1.6	6244	1107.34	17.56	0.18	658
ANSI C	2512	448.04	14.98	0.18	165
Python 2.5	3909	646.98	14.23	0.17	57
Java 1.5	7741	1342.88	17.87	0.17	5715
Java 1.4	7183	1279.29	17.99	0.18	5715
Java 1.3	6698	1189.24	17.76	0.18	5144
Java 1.2	6698	1189.24	17.76	0.18	5144
Java 1.1	6693	1193.28	17.65	0.18	5144
Java 1.0	5611	901.02	15.9	0.16	5144

Table 2). However, the evolution of this metrics is also directly connected to the complexity of the different versions of Java, but varies smoothly. Finally, the metrics value for the Java 1.6 grammar is not comparable to Java 1.5 grammar value, even if the language is the same. This is due to the fact that Java 1.6 grammar is designed for LL parsing and LL grammars are known to be more complex than LR ones. It is also probably due to a important use of conflict resolution mechanism in the Java 1.5 version (measured by the `lcc` metrics) that simplifies the grammar. From this result, it would appear that the metrics `lrs` is a good measure of the complexity of the grammar.

For each terminal in the grammar the `lat` metrics sums, the number of states in the LR automaton that does not lead to an error when this terminal is in the lookahead, and it is normalized by the number of terminals. Because `lat` metrics depends on the number of states, it is also interesting to normalize it by the number of states (`lrs`).

Indeed, the value for different versions of Java is very stable. It is also comparable to C, Java and Python grammar values. On the contrary, the value for the language GAL is very low. A low value for this metrics indicates that each terminal only appears in few of the grammar's states (5% of the states for GAL) and thus that the language is very controlled and probably easy to learn. On the other hand, a high value for this metrics, such as 49% for EBNF, indicates that the language may accept any terminal in approximately every state.

The results show that the metrics `lrtla` is closely related to the type of language. DSL grammars have smaller values than GPL grammars. Normalizing this metrics by the number of terminals in the grammar, it produces the same results as the normalized value of `lat`. This is to be expected since those two normalized metrics compute respectively the probability of being able to shift a given terminal in a given state and the probability of a given state to be able to shift a given terminal.

Results for the metrics `lcc` indicate that for most of the tested DSLs we do not have a LR conflict. Priorities/associativities are expressed in a recursive manner. If we compare ANTLR grammar for Java 1.6 and Tatoo grammar for Java 1.5, we can notice big difference in metrics `lcc` value. Value of Tatoo Java grammars for metrics `lcc` is high, because priorities/associativities are massively used in expressions.

In Table 5, the metrics `ss` provides results that are not related to the size of the grammar nor to the expressive power of the language. GPL and DSL grammars have similar values as well. This metrics evolves moderately with the different versions of Java. It seems to measure the verbosity of the grammar. Indeed, C or Python are known to be less verbose than Java. One surprising result is again, the smaller value of this metrics for the 1.6 version of Java. This is probably due to the larger number of productions for the LL version. Although these complementary productions have small sample sizes, the average value is smaller.

The `ltps`, `lptsm` and `ltpsa` metrics are directly related to the type of language. DSL have values below 1000, whereas GPL have values above 1000.

Table 5. Results for language based metrics

Lang	ss	ssm	ltps	ltpsm	ltpsa	ltpsn
Expr	1.56	4	35	6	4.29	0.43
FDL	2.53	6	47	8	4.34	0.23
EBNF	1.59	3	102	11	5.56	0.70
CFG Design	2.33	7	82	17	12.72	0.14
GAL	2.73	13	349	43	35.34	0.06
ANTLR V3	1.73	8	435	29	24.97	0.18
Ruby 1.8.5	1.47	7	3200	88	44.87	0.41
Java 1.6	2.04	10	2691	92	48.61	0.28
ANSI C	1.73	7	1777	81	41.92	0.25
Python 2.5	1.73	8	1576	61	48.33	0.21
Java 1.5	2.98	14	2370	83	50.06	0.22
Java 1.4	2.94	14	2272	81	49.78	0.22
Java 1.3	2.95	14	2239	80	49.25	0.22
Java 1.2	2.95	14	2239	80	49.25	0.22
Java 1.1	2.96	14	2200	79	48.81	0.22
Java 1.0	2.89	14	1734	78	48.77	0.18

They increase smoothly with the version of Java. The `ltpsn` value is very comparable to `lat/lrs`, since it measures the constraints on the language. The only language that gives different results is CDFG: the `ltpsn` stresses that the language is moderately constrained (14%) whereas the other metrics indicates 30%, which is quite high. Since, these two metrics are not exactly related, it is normal, that they produce different results, although large differences probably indicate an interesting property of the grammar. At this time we are not able to explain this behavior.

6. Usage Guidelines

There is currently no empirical study for grammar based metrics, which would ascertain or suggest when and how to use them. But from an in-depth understanding of metric design, we can give some useful usage guidelines. To do this we describe the advantages/disadvantages for each proposed metrics.

Metrics can be used for different purposes and in different stages of the grammar/language development life-cycle. The importance of metrics and their resulting interpretation is also dependent on their purpose. This is why it is important to identify the objective of the use. Three usages of proposed metrics are discussed in detail below.

Grammar-based Language Comparison

This is most common scenario in which we have different grammars and would like to compare them. First, we need to be aware that the same language can

be described with different grammars, and a grammar can be described with different forms, or it can be specialized for different parsing techniques. The best way to compare grammars is to compare grammars that are in the same form and that they use the same parsing technique. In this case, the simplest metrics to use are `lrs`, `lat` and `ltpsm`. These metrics indicate the size and complexity of grammars and because of that, they can be used to rank the grammars. For a precise comparison with common languages we suggest using the results from Table 4 and Table 5. For quick reference of grammar size and complexity we propose four different groups.

- Tiny, mainly toy grammars,
- Small, mainly DSL grammars,
- Intermediate, 3rd generation GPL languages grammar,
- Big, modern object-oriented language grammars.

The groups are defined using an analysis of tested grammars. The intent of these groups is not to classify grammars by size or complexity, but merely for quick reference in order to have a first impression about the grammar size.

Table 6. Metrics orientation values

Grammar size	lrs		lat		ltpsm	
	from	to	from	to	from	to
Tiny	0	100	0	20	0	6
Small	101	1000	21	200	7	50
Intermediate	1001	3000	201	500	51	70
Big	3001	$\geq$ 3001	501	$\geq$ 501	71	$\geq$ 71

Orientation values for each group and metrics are stated in Table 6.

Metrics that are less size-dependent are `lrtla` and `lrpsa`. `lrtla` indicates average complexity of each state and `lrpsa` indicates complexity of terminal pairs. In the third group we find metrics and normalizations that are not size-dependent `lat/lrs`, `lcc`, `ss` and `ltpsn`. In most cases these values are averaged by size. Because of this, they are less adapted to comparing larger grammars.

Developing a New Language

In this scenario, we develop a new language from scratch. In practice, this means that in most cases we develop new DSL (because they are most common [16, 24]). In order to compare the new language with others, we can use metrics as suggested in the previous section. In addition, the developer can monitor metrics after every incremental developing stage to evaluate its influence on the complexity and the verbosity of grammars. For this purpose, all of

the suggested metrics are appropriate, but if the developer wants to measure the verbosity of the language, he/she should look at the `lat/lrs`, `ss` or `lptsn` metrics, whereas, if the developer is interested in the complexity of the grammar he/she should look at other metrics. For developers it is also interesting to monitor number of conflicts `lcc`.

Sample Based Language Comparison

In this case, languages are compared based on samples/sentences. Two scenarios are possible: either formal grammar does not exist yet or a sample-based comparison is carried out in addition to a grammar-based comparison. To be able to compare or evaluate the language, we calculate the metrics `ltpsm`, `ltpsa` and `ltpsn` (`ltpX`). These metrics can be calculated directly from samples, with the condition that samples involved all allow combinations of terminals. In this scenario, there are two main problems. First, to identify all terminals and second to get some degree of confidence that our set of samples (S) is diverse enough. The first problem is lexically-related and solvable. The second problem can neither be tested nor computed. To overcome this obstacle, we use the relation between metrics with the unknown grammar G and metrics calculated from samples S. Value of `ltpX(G)` is always greater or equal to `ltpX(S)`. In practice this means that we can compare the language to a language that has smaller metrics values. If our goal is to infer grammar from samples, metrics `ltpsn` can help to evaluate a number of different non-terminals. Higher numbers mean fewer constraints in terms of language; this usually means less differing non-terminals. With this information we have more direct grammar inference search [6, 7].

7. Conclusion and Future Work

This paper explores the usefulness of several new metrics for grammar engineering. It presents experimental results for traditional metrics and for these new metrics on several grammars. These grammars cover domain-specific languages and general-purpose languages. Existing metrics are directly computed from the grammar itself. A first set of new metrics uses the LR automaton produced from the grammar. A second is related to the language recognized. We believe that these metrics provide interesting results that are not all covered by existing metrics. From this point of view, LR-based metrics are probably more suitable for grammar experts familiar with LR parsing, whereas other metrics could also be applicable for non-specialists in grammar development.

From experimental results, we see that some metrics are directly linked to the size or the complexity of the grammar whereas others remain stable even if the size or complexity of the grammar varies. Our findings show that the metrics in both cases qualify for evaluating the quality of the grammar. However, we consider that the quality of the grammar cannot be captured by a single metrics

but by a range of the metrics explored in this paper. Moreover, this quality is not an absolute value but is relative to other grammars.

In this paper we only explore the metrics of the grammar portion of the analyzer, without looking at the lexing portion of the analyzer. However, the complexity of these two parts is closely related. For instance, one could specify in the lexer a different token for `true` and `false` or establish a generic token for the booleans. In this case, the grammar would necessarily be different and may produce different metrics. Therefore, we believe that metrics on token definitions could also be useful to capture the entire complexity of a language analyzer. An analyzer with complex lexer and simpler parser may be less maintainable than a complex parser with a simple lexer.

References

1. International standard EBNF syntax notation. ISO/IEC standard n° 14977 (1996)
2. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compiler: Principles, Techniques, and Tools. Addison Wesley, 2nd edn. (2007)
3. Alves, T.L., Visser, J.: A case study in grammar engineering. In: Proceedings of the 1st International Conference on Software Language Engineering (SLE 2008). pp. 285–304. Lecture Notes in Computer Science Series, Springer Verlag (2008)
4. Cervelle, J., Forax, R., Roussel, G.: Tatoo: An innovative parser generator. In: 4th International Conference on Programming Principal and Practice in Java (PPPJ'06). pp. 13–20. ACM International Conference Proceedings, Mannheim, Germany (Aug 2006)
5. Cervelle, J., Forax, R., Roussel, G.: A simple implementation of grammar libraries. Computer Science and Information Systems 4(2), 65–77 (2007)
6. Črepinšek, M., Mernik, M., Javed, F., Bryant, B.R., Sprague, A.P.: Extracting grammar from programs: evolutionary approach. SIGPLAN Notices 40(4), 39–46 (2005)
7. Črepinšek, M., Mernik, M., Žumer, V.: Extracting grammar from programs: brute force approach. SIGPLAN Notices 40(4), 29–38 (2005)
8. Csuhaj-Varjú, E., Kelemenová, A.: Descriptional complexity of context-free grammar forms. Theoretical Computer Science 112(2), 277–289 (May 1993)
9. Denny, J.E., Malloy, B.A.: The ielr(1) algorithm for generating minimal lr(1) parser tables for non-lr(1) grammars with conflict resolution. Science of Computer Programming (September 2009), http://dx.doi.org/10.1016/j.scico.2009.08.001
10. Deursen, A. van., Klint, P.: Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology 10(1), 1–17 (2002)
11. Elliott, C., Finne, S., Moor, O.D.: Compiling embedded languages. In: Proceedings of the Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG'00). pp. 9–27. Springer-Verlag (Sep 2000)
12. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java Language Specification, Second Edition: The Java Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
13. Halstead, M.H.: Elements of Software Science. Elsevier, New York (1977)
14. Harbison, S.P., Steele Jr, G.L.: C A Reference Manual, Fourth Edition. Prentice-Hall, Upper Saddle River, NJ 07458, USA (1995)
15. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism sdf—reference manual—. SIGPLAN Not. 24(11), 43–75 (1989)

16. Javed, F., Mernik, M., Bryant, B., Sprague, A.: An unsupervised incremental learning algorithm for domain-specific language development. Applied Artificial Intelligence 22(7), 707–729 (2008)
17. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Transactions on Software Engineering and Methodology 14(3), 331–380 (2005)
18. Lämmel, R.: Grammar Testing. In: Proc. of Fundamental Approaches to Software Engineering (FASE) 2001. LNCS, vol. 2029, pp. 201–216. Springer-Verlag (2001)
19. Lutz, M., Ascher, D.: Learning Python, Second Edition. O'Reilly Media, Inc., Sebastopol, CA (2003)
20. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (2005)
21. Mernik, M., Korbar, N., Žumer, V.: LISA: A tool for automatic language implementation. ACM SIGPLAN Notices 30(4), 71–79 (Apr 1995)
22. Parr, T.J., Quong, R.W.: ANTLR: A Predicated-LL(k) Parser Generator. Software Practice and Experience 25(7), 789 – 810 (1995)
23. Power, J.F., Malloy, J.F.: A metrics suite for grammar-based software. Journal of Software Maintenance and Evolution: Research and Practice 16(6), 405–426 (2004)
24. Rebernak, D., Mernik, M., Wu, H., Gray, J.G.: Domain-specific aspect languages for modularising crosscutting concerns in grammars. IET software 3(3), 184–200 (2009)
25. Slivnik, B., Vilfan, B.: Producing the left parse during bottom-up parsing. Information Processing Letters (96), 220–224 (Dec 2005)
26. Thibault, S., Marlet, R., Consel, C.: Domain-specific languages: from design to implementation – application to video device drivers generation. IEEE Transactions on Software Engineering 25(3), 363–377 (May 1999)
27. Thomas, D., Fowler, C., Hunt, A.: Programming Ruby. The Pragmatic Programmer's Guide. Pragmatic Programmers (2004)
28. Watt, D.A.: Programming Language Concepts and Paradigms. Prentice-Hall (1990)

Appendix

7.1. Computation of closure application of rules

The first two steps of the computation of the shortest sample for a production require a same closure mechanism which is already implemented in the parser generator Tatoo to compute the first and the follow set [2].

The problem this mechanism solves can be formalized in the following way: we associate to each non-terminal X a mathematical object (a set of terminals for first and follow set, but a word in the process describe below) which we note $\mathcal{M}[X]$. The process fills the map $\mathcal{M}$ starting from some non-terminals and updates it using rules until the whole map is filled. In particular, an object associated to a non-terminal can change during the process. For instance, in the sequel, we try to compute the shortest words; in these cases, a word can be replaced by a shorter one.

The problem is expressed giving two rules. The first one, the *initiation rule*, tells how to initiate the process by giving an answer for some non-terminals and putting it in the map $\mathcal{M}$. The second one, the *iteration rule*, gives how to

construct new objects from others, leading to the construction of dependency maps which store, for a non-terminal X:

i. the non-terminals Y such that $\mathcal{M}[Y]$ changes when $\mathcal{M}[X]$ is updated
ii. the non-terminals Y such that $\mathcal{M}[Y]$ has to be computed in order to get $\mathcal{M}[X]$.

Note that map *i.* is easily computed from iteration rules and map *ii.* is the reverse of map *i.* In order to compute the object associated to X, the solver first recursively uses map *ii.* to get all the words that have to be computed and then uses the iteration rule in a loop until no more changes are made into the map $\mathcal{M}$.

7.2. Computation of a shortest word generated by X

We note this map $\mathcal{M}_1$.

The rule for the computation are the following:

- **initiation rule** : if $X \to \alpha$ is a production such that α is the shortest only made of terminals, X generates α, $\mathcal{M}_1[X] = \alpha$.
- **iteration rule** : if $X \to \alpha$ is a production such that α does not contains X, then, if smaller or not yet defined, $\mathcal{M}_1[X]$ is replaced by the word obtained replacing each non-terminals Y of α by $\mathcal{M}_1[Y]$.

Note that cycles in dependency map are not a problem since they always lead to longer words.

7.3. Computation of a shortest word leading to X

We note this map $\mathcal{M}_2$.

The rule for the computation are the following:

- if S is an axiom, S is a shortest word leading to S, that is $\mathcal{M}_2[S] = S$.
- if $X \to \alpha Y \beta$ is a production, then, if smaller or not yet defined, $\mathcal{M}_2[Y]$ is replaced by the word $\mathcal{M}_2[X]$ where X is replaced by $\alpha' Y \beta'$, α' [resp. β'] being the word α [resp. β] where all non-terminals Z in this word are replaced by $\mathcal{M}_1[Z]$.

Here again, cycles are not a problem for the same reason.

Matej Črepinšek received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research interests include grammatical inference, evolutionary computations, object-oriented programming, compilers and grammar-based systems. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Tomaž Kosar received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research is mainly concerned with design and

implementation of domain-specific languages. Other research interest in computer science include also domain-specific visual languages, empirical software engineering, software security, generative programming, compiler construction, object oriented programming, object-oriented design, refactoring, and unit testing. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Marjan Mernik received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently a professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is also an adjunct professor at the University of Alabama at Birmingham, Department of Computer and Information Sciences. His research interests include programming languages, compilers, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Julien Cervelle received the Ph. D. degree in computer science at Provence University, Marseille, France in 2002 and its habilitation thesis at Paris-Est University, France in 2007. His research interest are grammar based systems, parser generators, dynamical systems and cellular automata. He is currently Professor at Paris-Est University, France and adjunct professor at Ecole Polytechnique, Palaiseau, France.

Rémi Forax received the Ph. D. degree in Computer Science at Paris-Est University, France in 2001. He is currently a Teaching Assistant at Paris-Est University and a Java Community Process Expert for JSR 292. His main research areas concern design and implementation of programming languages, compiler construction, parser generators, virtual machines and executing environment.

Gilles Roussel received in computer science at UPMC, Paris, France in 1994 and its habilitation thesis at University of Marne-la-Vallée, France in 2003. He is currently professor at Paris-Est University and is deputy director of LIGM computer laboratory at Paris-Est University. His research interests are programming languages parsing, object-oriented design, program plagiarism detection, network programming and network routing algorithms.

Received: November 15, 2009; Accepted: March 30, 2010.