Metrication of SDF Grammars

Tiago Alves and Joost Visser

{tiago.alves,joost.visser}@di.uminho.pt

Technical Report DI-PURe-05.05.01

2005, May

PURe
Program Understanding and Re-engineering: Calculi and Applications
(Project POSI/ICHS/44304/2002)

Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

DI-PURe-05.05.01
Metrication of SDF Grammars by Tiago Alves and Joost Visser

Abstract

Metrication of grammars is an important instrument in grammar engineering, in particular to monitor iterative grammar development. In this report, we discuss the definition and implementation of metrics for the syntax definition formalism SDF. Two tools for SDF metrication have been developed: SdfMetz for static metrication of grammars themselves, and SdfCoverage for metrication of associated parser tests. We applied these tools to SDF grammars for a range of languages. We present and interpret the collected data.

1 Introduction

Quantification is an important instrument in software engineering. Quantification can for instance play a role in monitoring and controlling the software development process, or in specifying and improving software quality aspects such as performance or reliability. To wit, the 200 page *Guide to the Software Engineering Body Of Knowledge - SWEBOK* [1] contains about 500 references to the topic of quantification[1]. The basis of quantification is metrication: the definition and calculation of numeric measures of attributes of artifacts or processes. An extensive array of software engineering metrics have been defined and applied [6, 5].

Grammar engineering [11] is an emerging subfield of software engineering that aims to apply solid software engineering techniques to grammars, just as they are applied to other software artifacts. Such techniques include version control, static analysis, and testing. Through their adoption, the notoriously erratic and unpredictable process of developing and maintaining large grammars can become more efficient and effective, and can lead to results of higher-quality.

In grammar engineering, quantification is an important instrument for understanding and controlling grammar evolution as well as for specifying and improving grammar quality attributes, just as it is for software evolution and software artifacts in general.

Though grammars have been used in software engineering for decades, the systematic definition and application of grammar metrics is a more recent development. Purdom [21] pioneered with a *coverage* metric, which quantifies an attribute, not of the grammar *an sich*, but of the coverage of a grammar by a given set of sentences (usually a parser test suite). Recently, Lämmel provided a refined coverage metric [13]. Malloy *et al.* [20] have defined a suite of metrics for attributes of grammars themselves, such as *size*, *complexity*, and *structure*. All these authors provide definitions for grammars written in BNF, EBNF, or Yacc-style BNF dialects.

Our preferred grammar notation is the Syntax Definition Formalism SDF [9, 26]. This choice is motivated from our *grammar-centered* approach to language tool development [10], and the availability of Generalized LR parser support [22, 25] for SDF grammars, which is essential in this approach. Key differences between SDF and (E)BNF are reviewed in Section 2.

In this paper, we take the size, complexity, structure, and coverage metrics defined by others for (E)BNF, and adapt them for the SDF gram-

[1] With search keys like *quantification*, *metric*, *measure*, *statistic*, and their derivations.

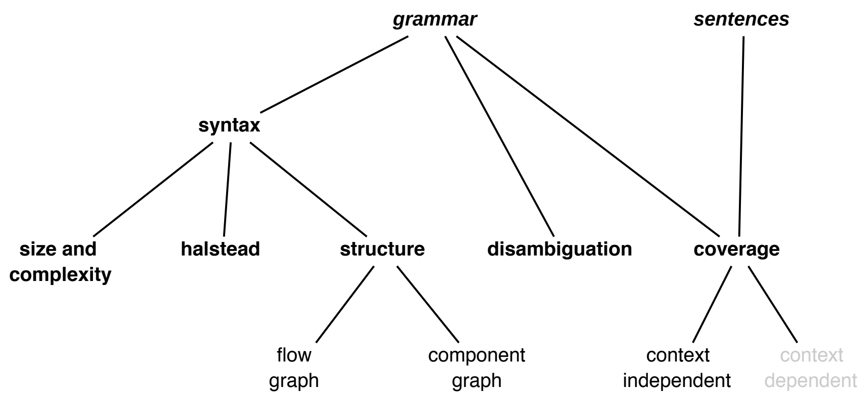

Fig. 1. Categories of grammar metrics.

mar notation. We also introduce *disambiguation* metrics, relevant for SDF only. We have implemented two tools, named SdfMetz and SdfCoverage, to support these metrics for SDF. We applied these tools to collect metrics data for a series of SDF grammars developed by us and by others. Elsewhere, we report on the use of our SDF metrication tools for monitoring the development of an industrial strength grammar of the VDM-SL language [2, 3]. Here we use excerpts from that grammar for purposes of exemplification.

Figure 1 provides a schematic overview of the metrics discussed in this paper. Sections 3, 4, 5, and 6 discuss the definition of the size, complexity, Halstead, structure, and disambiguation metrics for SDF, and their implementation in the SdfMetz tool. Section 7 discusses the definition of coverage metrics for SDF, and their implementation in the SdfCoverage tool. Section 8 presents metrics data collected with SdfMetz for a wide variety of SDF grammars, side by side with corresponding data for BNF grammars. We also comment on an extension of SdfMetz to accept a second GLR-supported grammar formalism, employed by Semantic Designs' DMS toolkit [4]. Finally, we discuss related work in Section 9 and the paper is concluded in Section 10.

2 The SDF grammar notation

The most salient difference between SDF and (E)BNF, is purely syntactical: the direction of grammar production rules. For instance:

BNF	SDF
`S ::= A B`	`A B -> S`

For the purposes of metrication, this difference is inconsequential.

But SDF differs from (E)BNF in more than pure syntax. For instance, it allows several productions for the same non-terminal, which are treated as alternatives.

BNF	SDF		

`S ::= A`
`    | B`
	`A -> S`
	`B -> S`
		or	`A | B -> S`

Thus, the number of productions in an (E)BNF grammar always coincides with the number of defined non-terminals, but not so in SDF. When adapting metrics to SDF, this forces us to choose between using the number of productions or the number of non-terminals in some definitions. Based on this possibility of distributing alternatives of a non-terminal over various productions, SDF offers supports for grammar modularity.

SDF offers more regular expression-style operators than BNF and even EBNF. An important example of this is the notation for separated lists. In SDF they can be written more concisely:

EBNF	SDF
`S ::= [A {";" A}]`	`{A ";"}* -> S`

where the curly braces in EBNF indicate zero or more repetitions and the square brackets indicate optionality. The curly braces in SDF together with the star indicate zero or more repetitions.

Furthermore, SDF grammars specify more than just context-free syntax. They also contain lexical syntax and disambiguation information. The notation for lexical syntax is the same as for context-free syntax, but grouped in `lexical syntax` rather than `context-free syntax` sections. The constructs for disambiguation include lookahead *restrictions* and chains of *priorities* which are specified in specific sections, as well as *reject* productions, and *associativity* and *preference* attributes which are specified in the `context-free syntax` sections. As explained below, we decided to apply the size, complexity, Halstead, and structure metrics only to the context-free syntax of SDF grammars (excluding reject productions), to make comparisons possible with the results of other studies, on (E)BNF grammars. For the disambiguation information, a dedicated set of metrics was defined.

3 Size and Complexity Metrics

Table 1 lists a number of size and complexity metrics for grammars. Most of these metrics were formally defined for BNF in [20]. Below we will

Size and complexity metrics		(E)BNF	SDF
TERM	Number of terminals	x	x
VAR	Number of defined non-terminals	x	x
PROD	Number of productions	x	x
MCC	McCabe's cyclometric complexity	x	x
AVS	Average size of RHS	x	
AVSn	Average size of RHS per non-terminal		x
AVSp	Average size of RHS per production		x

Table 1. Size and complexity metrics for grammars. The last two columns indicate which are defined in [20] for (E)BNF and adapted to SDF by us, and which were introduced specifically for SDF by us.

briefly discuss the adaptation of these BNF metrics to SDF as well as a few additional metrics specific to SDF. We will briefly remark on the implementation of these metrics in the SdfMetz tool.

TERM - Number of terminals The TERM metric is the count of unique terminals in a context-free grammar. In SDF, three types of terminals occur: literals (e.g. `";"` or `"begin"`), lexical non-terminals (i.e. non-terminals defined in a lexical syntax section, rather than in a context-free syntax section), and character classes (e.g. `[a-z]`). The latter are rarely used in context-free productions.

VAR - Number of non-terminals The VAR metric is the count of unique *defined* non-terminals in a context-free grammar.

Example 1. Consider the following grammar fragment:

```
QuoteLiteral                              -> Type
"compose" Identifier "of" FieldList "end" -> Type
Type "|" Type                             -> Type
Type "*" Type                             -> Type
```

Here, the only defined non-terminal is `Type`, and hence the value of VAR is 1. The non-terminals `QuoteLiteral`, `Indentifier`, and `FieldList` are *used*, but not defined in the above fragment.

Non-terminals that are used, but not defined in context-free syntax can either be *lexical* non-terminals, i.e. defined with productions in a lexical syntax section, or they can be undefined, i.e. *bottom* non-terminals. Bottom non-terminals occur in incomplete grammars, or grammar fragments. Since we want our metrics to be applicable also for incomplete grammars, we need to be clear about counting bottom non-terminals. We prefer to keep bottom non-terminals out of the VAR metric. Hence the restriction to *defined* non-terminals.

PROD - Number of productions In (E)BNF, the number of defined productions is equal to the number of defined non-terminals. But, as noted in Section 2, this is not the case in SDF. For this reason, we use the metric PROD for the number of context-free productions.

All three metrics TERM, VAR, and PROD are calculated by SdfMetz by collecting the relevant items into corresponding sets and subsequently taking the cardinality of those sets.

MCC - McCabe's cyclometric complexity The MCC metric is an interpretation of McCabe's cyclometric complexity [27] for grammars, based on an analogy between grammar production rules and program procedures.

The cyclometric complexity of a program procedure is defined as the number of possible non-circular execution paths, and is generally implemented as the number of branching control constructs (e.g. `if`, `while`), plus 1. There is always at least one execution path, i.e. MCC ≥ 1.

Following [20], MCC for a complete grammar is computed by summing the values for individual production. The value of an individual production is the number of branching operators (alternatives, repetitions, optionals) that it contains. Note that according to this definition, the value of MCC is always one below the number of paths, and the minimum value of MCC is not 1, but 0.

As in BNF, alternatives in SDF give rise to additional paths. Consider the following production from our SDF grammar for VDM-SL [2]:

```
Type | ("(" ")") -> DiscretionaryType
```

The value of *MCC* for this production is 1 since it contains a single alternative operator (`|`).

In addition to the alternative operator, in SDF different alternatives can be expressed with multiple production rules. The following example is equivalent to the excerpt presented before:

```
Type    -> DiscretionaryType
"(" ")" -> DiscretionaryType
```

This fragment has value 1, just as its equivalent.

There are more operators that create branches and thus affect the *MCC* value: optional and repetition operators. Consider the following productions:

```
"types" { TypeDefinition ";"}+ -> TypeDefinitions
Identifier "=" Type Invariant? -> TypeDefinition
```

Both production rules have a *MCC* value of 1. The first production uses the repetition operator `+` which allows two possible paths: a single `TypeDefinition` or several `TypeDefinitions` separated by a semi-colon. The second production uses the optional operator (`?`), again allowing two paths corresponding to zero or one occurrences of Invariant.

SdfMetz calculates the MCC value for each production by counting occurrences of alternative, optional, and repetition operators. The total MCC value of a grammar is computed as the sum of all MCC values of the individual productions. To account for the occurrence of more than one production per non-terminal, the number of such additional productions (PROD minus VAR) is added to that sum.

AVS - Average size of RHS The AVS metric represents the average size of the right-hand sides (RHS) of context-free grammar productions. Given the inverted direction of SDF productions versus BNF production, we adopt the direction-neutral term of *body* of a production, rather than right-hand side.

In (E)BNF the definition of this metric is trivial: count the number of terminals and non-terminals in the body of each production, and divide by the total number of productions (PROD) or, equivalently by the number of defined non-terminals (VAR).

Since in SDF the values of PROD and VAR are not necessarily equal, two interpretations of the metric are possible: average production body size *per production* (AVSp) and average production body size *per non-terminal* (AVSn).

Example 2. Consider the following fragment:

```
"-" | Identifier                    -> Pattern
"(" Expression ")"                  -> Pattern
SymbolicLiteral                     -> Pattern
"{" PatternList "}"                 -> Pattern
Pattern "union" Pattern             -> Pattern
"[" PatternList "]"                 -> Pattern
Pattern "^" Pattern                 -> Pattern
"mk_" "(" Pattern "," PatternList ")" -> Pattern
"mk_" Name "(" PatternList? ")"     -> Pattern
```

The body sizes of these productions are 2, 3, 1, etc., totalling to 29. The number of productions is 9, and the number of defined non-terminals is

Halstead metrics (primitive)		
$\mathbf{u}_1$	Number of distinct operators	
$\mathbf{u}_2$	Number of distinct operands	
$\mathbf{n}_1$	Total number of operators	
$\mathbf{n}_2$	Total number of operands	
Halstead metrics (derived)		Formula
n	Program vocabulary	$n = u1 + u2$
N	Program length	$N = n1 + n2$
V	Program volume	$V = N \times log_2 n$
D	Program difficulty	$D = \frac{u_1}{2} \times \frac{n_2}{u_2}$
E	Program effort	$E = D \times V$
L	Program level	$L = \frac{2}{u_1} \times \frac{u_2}{n_2}$
T	Program time	$T = \frac{E}{18}$

Table 2. Halstead metrics, primitive and derived.

1. Hence, the value of AVSp is $29/9 \approx 3.22$ and the value of AVSn is 29. This is exactly how SdfMetz computes these metrics.

4 Halstead Metrics

The classical Halstead effort metric [8] has also been adapted for (E)BNF grammars in [20]. We adapted the metric to SDF and implemented it in SdfMetz. Moreover, we compute values not only for the Halstead effort metric but also for some of its ingredient metrics and related metrics. Figure 2 shows a list. There are four primitive metrics, counting distinct and total numbers of operands and operators in a software artifact. Using various formulas, seven more metrics are derived from the primitive ones.

In [20], the Halstead effort metric (denoted there by HAL) is categorized with the size metrics. In Figure 1 we devote a separate category to Halstead metrics, because we discuss the entire range, not just effort.

$\mathbf{u_1}$, $\mathbf{u_2}$, $\mathbf{n_1}$, $\mathbf{n_2}$ - Primitive Halstead metrics The essential step in adapting Halstead's metrics to grammars is to interpret the notions of *operand* and *operator* in the context of grammars. In [20] the operands and operators are defined for (E)BNF. We have extended that interpretation to arrive at a definition of these notions for SDF.

For a first simple example, consider the following production rule from our VDM-SL grammar:

```
    "forall" BindList "&" Expression  -> QuantifiedExpression
```

Operand		Non-terminal
terminal	e.g. `";"` or `begin`	`Literal`
non-terminal	e.g. `Expression`	`Sort`
character class	e.g. `[a-z]`	`CharClass`
number	e.g. 2	`NatCon`
Operator		Syntax
label	`:`	`Literal ":" Symbol`
epsilon	`()`	`"(" ")"`
sequence	`(···)`	`"(" Symbol Symbol* ")"`
optional	`?`	`Symbol "?"`
alternative	`|`	`Symbol "|" Symbol`
repetition (≥ 0)	`*`	`Symbol "*"`
repetition (≥ 1)	`+`	`Symbol "+"`
repetition ($\geq n$)	`+`	`Symbol NatCon "+"`
repetition w. separator (≥ 0)	`{···}*`	`"{" Symbol Symbol "}" "*"`
repetition w. separator (≥ 1)	`{···}+`	`"{" Symbol Symbol "}" "+"`
repetition w. separator ($\geq n$)	`{···}·+`	`"{" Symbol Symbol "}" NatCon "+"`
set	`Set [···]`	`"Set" "[" Symbol "]"`
pair	`#`	`Symbol "#" Symbol`
function	`(···=>·)`	`"(" Symbol* "=>" Symbol ")"`
permutation	`<<···>>`	`"<<" Symbol* ">>"`

Table 3. Operands and operators of SDF. Note that the syntax `Symbol*` gives rise to implicit juxtaposition operators for symbol lists longer than 1.

In this production, all terminals and non-terminals are regarded as operands, and they happen to be all distinct. Thus, $\mathrm{u}_2 = \mathrm{n}_2 = 5$. Furthermore, the arrow (`->`) that joins production body and defined non-terminal, is regarded as an operator, and so are the 'implicit' juxtaposition operator that compose the various terminals and non-terminals into a production body. If we would turn this implicit juxtaposition into an explicit concatenation operator (`.`), the production would look as follows:

```
"forall" . BindList . "&" . Expression  -> QuantifiedExpression
```

With this auxiliary transcription in mind, it is easy to see that $\mathrm{u}_1 = 2$ and $\mathrm{n}_1 = 4$.

SDF offers a wide range of operators beyond the juxtaposition and arrow operators, and a few additional operands beyond non-terminals and terminals as well. Table 3 offers a more complete overview.

Example 3. For an example, consider the following two productions from the VDM-SL grammar:

```
"let" { LocalDefinition "," }+ "in" Expression     -> Expression
"let" Bind ("be" "st" Expression)? "in" Expression -> Expression
```

Structure metrics	
TIMPi	Tree impurity of immediate successor graph (%)
TIMP	Tree impurity of transitive successor graph (%)
LEV	Count of levels
CLEV	Normalized count of levels (%)
NSLEV	Number of non-singleton levels
DEP	Size of largest level
HEI	Maximum height

Table 4. Structure metrics for grammars.

In this excerpt, is possible to observe that besides juxtaposition and `->` operators, other operators are present: iterative, sequence and optional operators. Thus, the values of the primitive Halstead metrics are u_1=5, u_2=8, n_1=13, and n_2=14.

In SdfMetz, the count of operators and operands is implemented through a recursive algorithm that traverses the parse trees of all context-free productions. The algorithm returns a pair of two lists: operators and operands. The length of these lists are the values of n_1 and n_2. To obtain u_1 and u_2, the cardinality is computed of sets created from these lists.

n, N, V, D, E, L, T - Derived Halstead metrics The derived Halstead metrics are computed from the primitive ones using the formulas in Table 2. SdfMetz computes and reports all of them. For the two productions of Example 3, their values are: n=15, N=27, V=99.91, D=4.375, E=4371, L=0.2286, and T=24.28.

The theory of software science behind Halstead's metrics has been widely questioned. In particular, the meaningfulness and validity of the effort (E) and time (T) metrics have been called into question [6]. Below, we will still discuss and report values of Halstead effort for purposes of comparison to data reported in [20].

5 Structure Metrics

Table 4 shows a list of structure metrics defined for (E)BNF in [20]. We will discuss the adaptation of these metrics to SDF.

In contrast to the size, complexity, and Halstead metrics shown before, structure metrics are not computed by counting simple observations in a grammar. Rather, they presuppose the construction of an *immediate*

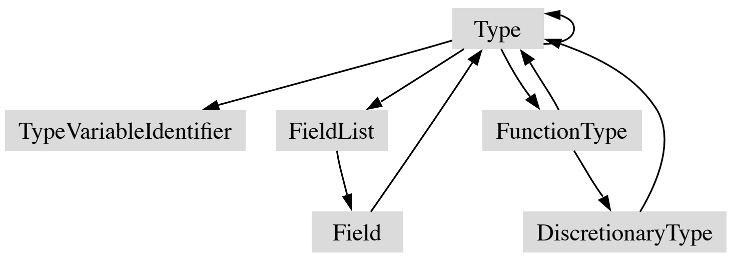

Fig. 2. Immediate successor graph for Example 4.

successor graph from which they are computed. This is a directed graph which has context-free non-terminals as nodes, and which contains an edge between two non-terminals whenever one occurs in the body of a production that defines the other.

Example 4. Consider, as an example, the following excerpt from the VDM-SL grammar:

```
Name                                         -> Type
TypeVariableIdentifier                       -> Type
"@" Identifier                               -> TypeVariableIdentifier
"compose" Identifier "of" FieldList "end"    -> Type
Field*                                       -> FieldList
(Identifier ":")? Type                       -> Field
"set" "of" Type                              -> Type
FunctionType                                 -> Type
DiscretionaryType "->" Type                  -> FunctionType
DiscretionaryType "-t>" Type                 -> FunctionType
Type | ("(" ")")                             -> DiscretionaryType
```

The immediate successor graph corresponding to this excerpt is shown in Figure 2. Note that lexical non-terminals, such as `Name` and `Identifier` do not appear as nodes in the graph. Also, multiple immediate successor connections between the same pair of non-terminals are summarized as a single edge.

As we will discuss below, the tree impurity metrics can be calculated directly from the immediate successor graph or its transitive closure, while the remaining structure metrics presuppose the transformation of the immediate successor graph into a so-called *level graph*.

Figure 3 shows the level graph obtained from the immediate successor graph of Figure 2. The level graph contains as nodes the strongly connected components of the immediate successor graph, which are called

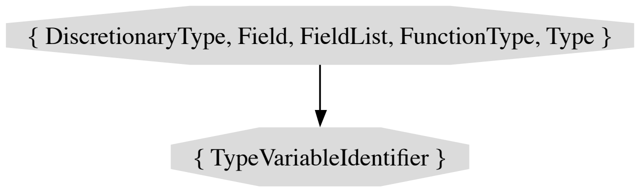
{ DiscretionaryType, Field, FieldList, FunctionType, Type }
{ TypeVariableIdentifier }

Fig. 3. Level graph for Example 4.

the *levels* of the grammar. These strongly connected components are sets of non-terminals that can be reached from each other, i.e. for which the immediate successor graph contains a path from one to the other and *vice versa*. The level graph contains an edge between two strongly connected components whenever at least one edge exist between a non-terminal in one component an a non-terminal in another component. By definition, the level graph is acyclic.

SdfMetz implements construction of immediate successor graphs and level graphs, and can export both such graphs to the *dot* format of GraphViz [12].

TIMP - Tree impurity of transitive successor graph (%) Fenton *et al.* [6] define tree impurity (TIMP) for undirected graphs without self-edges as $\frac{2(e-n+1)}{(n-1)(n-2)} \times 100\%$, where n is the number of nodes, and e is the number of edges. TIMP indicates to what extent the shape of a graph deviates from a tree shape. A tree impurity of 0% means that the graph is a tree and a tree impurity of 100% means that it a fully connected graph.

Power *et al.* [20] apply the tree impurity metric to the *transitive closure* of the immediate successor graph, i.e. to the (non-immediate, transitive) successor graph. Since tree impurity is defined for undirected graphs without self-edges, this requires that self-edges and multiple edges between non-terminals should be removed from the edge count before applying the formula.

Since, when we ignore edge direction, a path exists between any two non-terminals in the immediate successor graph of Example 4, its transitive closure is a fully connected graph. Correspondingly, the value of TIMP is 100%.

TIMPi - Tree impurity of immediate successor graph (%) We propose to apply the tree impurity metric also to the *immediate* successor graph, and we call this metric TIMPi.

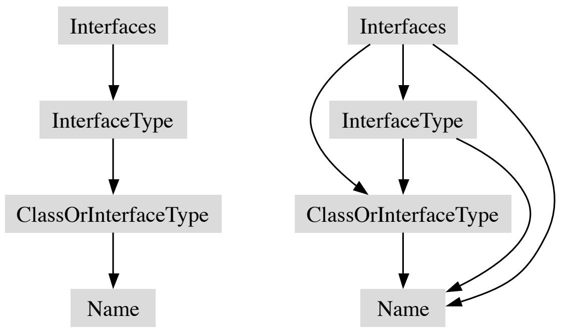

Fig. 4. Immediate and transitive successor graphs for Example 5.

The value of TIMPi for Example 4 is 20.0%, much smaller than its value for TIMP. In fact, even purely tree-shaped graphs will drastically increase in tree impurity when we take their transitive closure.

Example 5. For an extreme example, consider the following fragment from a Java grammar:

```
"implements" {InterfaceType ","}+ -> Interfaces
ClassOrInterfaceType              -> InterfaceType
Name                              -> ClassOrInterfaceType
{Identifier "."}+                 -> Name
{Identifier "."}+ "." "class"     -> Name
```

The immediate and non-immediate successor graphs of this fragment are shown in Figure 4. The tree impurity values for this fragment are TIMPi=0% and TIMP=100%.

More generally, all pure trees that have internal nodes (nodes with both incoming and outgoing edges) will lose their tree impurity to some degree after transitive closure. In Section 8 we will compare values for TIMP and TIMPi for a range of grammars.

SdfMetz reports values for both tree impurity metrics, TIMP and TIMPi. To compute these values, the transitive and immediate successor graphs are constructed, and the tree impurity formula is applied after removing direction and self-edges from each graph.

LEV - Number of levels The LEV metric represents the number of nodes in the level graph. For Example 4, and as can be seen in Figure 3, the value of LEV=2.

CLEV - Normalized count of levels (%) The CLEV metric, or normalized count of levels, expresses the number of nodes in the level graph (LEV) as a percentage of the number of nodes in the successor graph (VAR). A normalized count of levels of 100% means that there are as many levels in the level graph as non-terminals in the successor graph. In other words, there are no circular connections in the successor graph, and the level graph only contains singleton components. A normalized count of levels of 50% means that about half of the non-terminals of the successor graph are involved in circularities and are grouped into non-singleton components in the level graph. For Example 4, the value of CLEV=33.33%.

SdfMetz reports both the absolute count of levels (LEV) and the normalized count (CLEV).

NSLEV - Number of non-singleton levels The NSLEV metric indicates how many of the grammar levels contain more than a single non-terminal. SdfMetz reports both the number and, optionally, exports a list of the non-singleton levels. For Example 4, there is exactly one non-singleton level, containing the five non-terminals `DiscretionaryType`, `Field`, `FieldList`, `FunctionType`, and `Type`. Thus, the value of NSLEV=1.

DEP - Size of largest level The DEP metric measures the depth of the level graph as the maximum number of non-terminals per level. For Example 4, the value of DEP is 5.

HEI - Maximum height Maximum height measures the height of the level graph as the longest vertical path through the level graph, i.e. the biggest path length (in number of nodes) from a source of the level graph to a sink. In the level graph of Example 4, there is only a single path, connecting two nodes. Thus, HEI=2.

6 Disambiguation Metrics

In SDF, disambiguation constructs are provided in the same formalism as the syntax description itself. To quantify this part of SDF grammars, we defined a series of metrics, which are shown in Table 5. These metrics are simple counters for each type of ambiguity construct offered by the SDF notation.

Ambiguity metrics	
FRST	Number of follow restrictions
ASSOC	Number of associativity attributes
REJP	Number of reject productions
UPP	Number of unique productions in priorities
PREF	Number of preference attributes

Table 5. Disambiguation metrics for SDF grammars.

FRST - Number of follow restrictions Follow restrictions specify that some non-terminals can not be followed by particular characters. This can be useful, for instance, to specify the longest match of lexicals. For example, the following restrictions appear in the VDM-SL grammar:

```
Identifier -/- [a-zA-Z0-9]
NumericLiteral -/- [0-9]
```

They specify that an `Identifier` can not leave trailing alpha-numeric characters and that a `NumericLiteral` can not leave trailing digits. For the entire VDM-SL grammar, the value of FRST=4.

ASSOC - Number of associativity attributes The associativy attributes of a production specify how that production should associate with itself. For instance consider the sum operator of an expression:

```
Expression "+" Expression -> Expression { left }
```

Because the associativity attribute is `left`, the expression $2 + 3 + 4$ is recognized as $(2 + 3) + 4$. For the entire VDM-SL grammar, the value of ASSOC=35.

REJP - Number of reject productions Reject productions are a disambiguation instrument commonly used to specify reserved keywords. For example:

```
"let" -> Identifier { reject }
```

This specifies that the `let` keyword is not allowed as `Identifier`. More generally, the body of a reject production is not necessarily a single non-terminal, but can be any piece of syntax, which basically enables grammar *subtraction*. For the entire VDM-SL grammar, the value of REJP=99.

UPP - Number of unique productions in priorities In SDF, priorities between productions are specified with chains of priority groups. For example:

```
  "inverse" Expression -> Expression
>
{ left:
  Expression "*" Expression   -> Expression
  Expression "/" Expression   -> Expression
}
>
{ left:
  Expression "+" Expression    -> Expression
  Expression "-" Expression    -> Expression
}
```

Here, the first group consists of a single production, for the `inverse` operator. It takes higher priority than multiplication (`*`) and division (`/`), which have equal priority to each other, and associate to the left. The last group, of addition (`+`) and subtraction (`-`) takes lowest priority.

For the entire VDM-SL grammar, the total number of unique productions appearing in priority chains (UPP) is 72.

PREF - number of preference attributes The preference attributes `prefer` and `avoid` can be placed on a production to specify that it takes preference over all other productions, or conversely that all other productions take preference over the attributed production. The VDM-SL grammar, for example, contains the following production:

```
PatternBind "=" Expression    -> EqualsDefinition {prefer}
PatternBind "=" CallStatement -> EqualsDefinition
```

Some call statements in VDM can also be parsed as expressions, and both can occur in the right-hand side of an equals definition. Thus, in this case, the `prefer` attribute serves to resolve an ambiguity between `Expression` and `CallStatement` that occurs in the specific context of an `EqualsDefinition`.

7 Coverage Metrics

To determine how well a given grammar has been tested, a commonly used indicator is the number of non-empty lines in the test suites. A more reliable instrument to determine grammar test quality is coverage analysis. We have adapted the BNF rule coverage (RC) metric [21] for this purpose. The RC metric simply counts the number of production rules used during parsing of a test suite, and expresses it as a percentage of the total number of production rules of the grammar.

RC, NC - rule and non-terminal coverage In the case of SDF, several interpretations of RC are possible, due to the fact that a single non-terminal may be defined by multiple productions. One possibility is to count each of these alternative productions separately. We will reserve the RC name for this form of rule coverage. Another possibility is to count different productions of the same non-terminal as one. For this metric, we introduce the name *non-terminal coverage* (NC). RC gives a more accurate indication than NC of how extensively a grammar is covered by a given test suite. However, for comparison with rule coverage for BNF grammars, NC is more appropriate.

An even more accurate indication of coverage can be obtained with context-dependent rule coverage [13] (CDRC). This metric takes into account not just whether a given production is used, but also whether it is used in each context where it can actually occur. However, implementation, and especially computation, of this metric is more involved.

We have implemented the tool SdfCoverage to compute RC and NC for SDF. The tools takes parse trees produced by the SGLR parser as input. The nodes of these parse trees are labeled with ASTs of the SDF productions that were involved in creating them. In addition, the SdfCoverage tool receives the original SDF grammar as input. It simply collects all productions from the given parse trees into a set of *used* productions, and all productions from the grammar into a set of *defined* productions. Subsequently, the RC percentage is computed by dividing the cardinality of the first set (used) by the cardinality of the second (defined). To compute the NC percentage, these sets are first transformed into sets of defined non-terminals. Then, again the quotient of their cardinalities is computed.

Table 6 lists the various coverage metrics for (E)BNF and SDF. In [2], we report on measured values of NC and RC during all 48 steps of our VDM-SL grammar development project.

8 Data collection

We have sampled a series of grammars with our SdfMetz tool. In this section, we present the collected data and provide some observations regarding their interpretation. A full statistical analysis of the collected data is beyond the scope of this report.

Coverage metrics		(E)BNF	SDF
RC	Rule coverage	x	
RC	Rule coverage per production		x
NC	Rule coverage per non-terminal		x
CDRC	Context-dependent rule coverage	x	

Table 6. Coverage metrics for grammars. The last two columns indicate which are defined elsewhere for (E)BNF, and which were introduced specifically for SDF by us.

8.1 Sampled grammars

We have used SdfMetz to collect metrics data for a series of SDF grammars from various origins. From the Grammar Base (an online repository of SDF grammars [7]), we obtained SDF grammars for Yacc, BibTex, Fortran 77, Toolbus, Stratego, SDF itself, Java, AT&T SDL, C, and Cobol. From the Software Improvement Group [24], we obtained grammars for DB2/SQL and PL/SQL. From source distributions of SDF-related tools, PGEN [19] and Strafunski [16], we obtained grammars for Risla, Casl, and again Cobol. Finally, from local projects, we took grammars for VDM-SL and VB.Net.

We collected data also from grammars written in the syntax notation of the Design Management System (DMS) toolkit [4] of Semantic Designs [23]. The DMS toolkit supports GLR parsing, which makes grammars written in its syntax notation interesting for comparison to SDF grammars. For this purpose, we adapted the front-end of SdfMetz to accept DMS grammars as well as SDF grammars. The DMS notation has a minimal design. Similar to basic BNF, the DMS notation lacks operators for repetition or optionals. In fact, it even dispenses with the alternative operator. Instead, the DMS notation allows several productions for a single non-terminal, as does SDF. Semantic Designs allowed us to perform measurements on their DMS grammars for Java, ECMAScript, PHP 5, C^{++}, and Verilog 2001.

Apart from the data collected by us on SDF and DMS grammars, we will reproduce, for comparison purposes, some data from the paper from which we adopted the various grammar metrics [20]. This data concerns grammars for C, Java, C^{++}, and $C^{\#}$, which were developed in BNF-like notation. Note that for these grammars, the AVSn and AVSp metrics are always equal, since the number of productions and non-terminals is always equal in BNF grammars.

Table 7 lists the various grammars that were sampled, together with some of their properties. Two versions of the same Cobol grammar are

Language	Origin	Notation	Purpose	Paradigm
Yacc	gb	SDF	specify syntax	declarative
BibTex	gb	SDF	specify bibliographic data	declarative
Fortran 77	gb	SDF	programming	procedural
Toolbus	gb	SDF	specify protocols	declarative
Stratego	gb	SDF	programming	declarative
SDF 2.4	gb	SDF	specify syntax	declarative
SDF 2.3	gb	SDF	specify syntax	declarative
Java	gb	SDF	programming	object-oriented
AT&T SDL	gb	SDF	specification	declarative
C	gb	SDF	programming	procedural
Cobol (*alt*)	gb	SDF	programming	procedural
DB2/SQL	sig	SDF	querying	declarative
PL/SQL	sig	SDF	programming & querying	procedural
Risla	pgen	SDF	specify financial products	declarative
Casl	pgen	SDF	specification	declarative
Cobol	sfi	SDF	programming	procedural
VDM-SL	ta	SDF	specification	declarative
VB.net	jv	SDF	programming	object-oriented
ECMAScript	sd	DMS	web programming	scripting
PHP 5	sd	DMS	web programming	scripting
Java 5	sd	DMS	programming	object-oriented
Verilog 2001	sd	DMS	hardware description	mixed
C^{++}	sd	DMS	programming	object-oriented
C	pm	BNF	programming	procedural
Java	pm	BNF	programming	object-oriented
C^{++}	pm	BNF	programming	object-oriented
$C^{\#}$	pm	BNF	programming	object-oriented

gb	= The online Grammar Base [7].	jv	= Joost Visser.
sig	= Software Improvement Group [24].	ta	= Tiago Alves.
pgen	= Parsetable Generator source distribution [19].	sd	= Semantic Designs [23].
sfi	= Strafunski source distribution [16].	pm	= Power and Malloy [20].

Table 7. Sampled grammars and some of their properties.

listed: the one marked *alt*, from the Grammar Base, makes heavy use of nested alternatives, while in the other one, such nested alternatives have been folded into new non-terminals.

8.2 Comparing size, complexity, and structure metrics

Size, complexity, and structure data have been obtained for all sampled grammars, as shown in Table 8. Data for the Halstead effort metric (E) is also listed, and the grammars are sorted by their value for that metric.

Grammar	TERM	VAR	PROD	MCC	AVSn	AVSp	E	TIMPi	TIMP	LEV	CLEV	NSLEV	DEP	HEI
Yacc	25	13	26	34	5.1	2.5	8.4	4.5	42.4	11	84.6	2	2	6
BibTex	20	6	16	21	12	4.4	8.9	20	100	6	100	0	1	6
Fortran	41	16	41	32	8.8	3.4	19	4.8	42.9	15	93.8	1	2	7
Toolbus	44	23	59	47	7.3	2.9	21	4.8	64.1	22	95.7	1	2	10
C (pm)	*86*	*65*		*149*	*5.9*		*51*		*64.1*	22	*33.8*	*3*	*38*	*13*
Stratego	66	25	106	112	13	3.2	75	4.7	94.9	13	54.2	1	12	7
Sdf (2.4)	70	43	131	107	6.9	2.3	79	2.3	68.3	38	90.5	3	3	21
ECMAScript	144	90	344	254	9.6	2.5	81	1.6	83.7	29	32.2	3	57	10
Risla	71	53	106	87	7.7	3.9	82	3.3	41.9	47	92.2	2	4	12
Sdf (2.3)	74	44	137	108	6.9	2.2	83	2.4	68.1	38	88.4	3	4	21
Casl	79	95	222	190	6.0	2.6	94	2.2	65.6	68	72.3	4	13	24
Java (pm)	*100*	*149*		*213*	*4.1*		*95*		*32.7*	89	*59.7*	*4*	*33*	*23*
PHP	159	112	418	306	8.8	2.4	97	1.6	76.1	55	49.1	2	37	15
Java (sfi)	101	105	239	197	5.0	2.2	132	1.6	83.6	44	41.9	1	62	14
AT&T SDL	89	91	174	170	5.0	2.6	133	1.1	39.8	76	83.5	2	13	14
Java (sd)	99	144	460	316	7.8	2.5	138	1.3	93.9	41	28.5	2	87	17
C (gb)	91	73	190	190	6.1	2.3	143	2.8	89.9	30	41.7	2	39	11
C^{++} (pm)	*116*	*141*		*368*	*6.1*		*173*		*85.8*	21	*14.9*	*1*	*121*	*4*
DB2/SQL	214	98	292	311	7.9	2.6	185	1.0	42.6	69	71.1	1	29	16
$C^{\#}$	*138*	*245*		*466*	*4.7*		*228*		*29.7*	159	*64.9*	*5*	*44*	*28*
VDM-SL	143	71	227	232	10	3.3	248	2.8	78.7	35	49.3	3	27	13
VB.Net	169	233	473	469	4.7	2.3	294	0.84	48.4	156	67.0	6	43	26
Cobol (sfi)	338	493	829	739	3.2	1.9	306	0.24	12.6	465	94.3	3	20	26
Verilog	232	488	1248	760	6.6	2.6	456	0.45	46.1	266	54.5	10	117	19
Cobol (gb)	379	185	231	1158	10.4	8.3	578	1.2	22.1	150	81.1	5	21	14
PL/SQL	456	499	1094	888	4.5	2.1	711	0.34	24.5	434	87.0	2	38	29
C^{++} (sd)	173	436	2122	1686	12	2.5	1301	0.67	96.3	42	9.63	2	393	10

Table 8. Values of size, complexity, and structure metrics. The slanted grammars are in BNF, whose values are reproduced from [20], except LEV, which is computed by us as VAR×CLEV. The remaining grammars are in SDF and DMS. Rows have been sorted by Halstead effort (E), which is reported in thousands.

The value of McCabe's cyclomatic complexity (MCC) is below 500 for 22 of our 27 grammars. From the remaining ones, the two highest scoring grammars are the Cobol grammar from the Grammar base (MCC=1158) and the C++ grammar from Semantic Designs, which double and triple this value, respectively. An interest observation about MCC can be made when we divide it by the number of non-terminals (VAR) or by the number of productions (PROD). Most grammars turn out to fall within a small bandwidth with respect to these ratios ($0.6 \leq \frac{\text{MCC}}{\text{PROD}} \leq 1.4$ and $1.5 \leq \frac{\text{MCC}}{\text{VAR}} \leq 3.5$). Only the Cobol grammar from the Grammar Base exceeds by several factors the upper bounds of these intervals ($\frac{\text{MCC}}{\text{PROD}} = 5.0$ and $\frac{\text{MCC}}{\text{VAR}} = 6.3$), which is again due to its style of alternative usage.

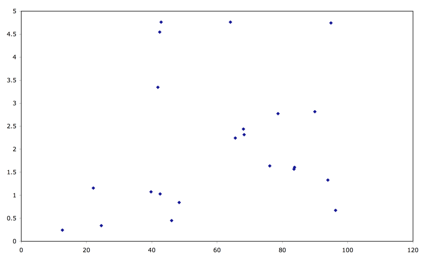

Fig. 5. Scatter plot that shows the two tree impurity metrics, TIMP and TIMPi, not to be correlated.

Note that the average size of production body per production (AVSp) is remarkably constant throughout all of the SDF and DMS grammars, *viz.* mostly between 2 and 4. Apparently, most grammar writers tend to organize their grammars in such a way that production size averages to such a value. A notable outlier in this respect is the Cobol grammar from the Grammar Base (AVSp=8.3), which makes heavy use of nested alternatives. This tends to increase the size of production bodies. The Cobol grammar from the Strafunski bundle, where such nested alternatives have been removed, actually has a value at the low end (AVSp=1.9).

The ranges of values of the two tree impurity metrics are quite different. For immediate successor graphs (TIMPi), the values lie between 0.24% and 4.8%, with the exception of the smallest grammar (in terms of TERM, VAR, and PROD), i.e. BibTex (TIMPi=20%). For transitive successor graphs (TIMP), the values range between 12.6% and 96.3%, again excluding BibTex (TIMP=100%). One may also observe that there is no clear correlation between the two, as illustrated by the scatter plot in Figure 5.

The two C^{++} grammars exhibit the lowest values for the normalized count of levels metric (CLEV), indicating a high degree of circularity in their immediate successor graphs. The BNF and DMS grammars tend to have lower CLEV values (all below 65), then the SDF grammars (most above 65). This may be due to SDF's larger repertoire of repetition operators, which allows to prevent various kinds of recursion in grammars. The absolute count of levels shows that most grammars score below 100, except for 6 grammars, that have values between 150 and 465. The number of non-singleton grammars is quite limited for most grammars (NSLEV $\leq$ 5), except for the VB.Net grammar (6) and the Verilog grammar(10).

The size of the largest grammar level (DEP) tends to be smaller for SDF grammars (most below 30) than for BNF and DMS grammars (all above 30). We can hypothesize that this is due to the different approaches to disambiguation, in particular of *expression* syntax. In SDF, the typical approach is to have a single non-terminal for expressions, and use priority chains to disambiguate the various productions that define this non-terminal. In BNF, the typical approach is to divide expressions over as many non-terminals as there are priority distinctions, and to build disambiguation into the syntax itself by appropriately nesting these non-terminals in each other's productions. Inspection of the grammars could shed light on this hypothesis, but only the SDF grammars are available to us, at least in non-obfuscated form. Grammar height (HEI) seems to be fairly evenly distributed between 4 and 29.

8.3 Comparing Halstead metrics

For the SDF and DMS grammars we have computed the full range of Halstead metrics, not just the effort metric. Table 9 shows the values. Rows are again sorted by Halstead effort (E).

Note that the value of n_1 is 2 for all DMS grammars, which reflects the fact that the only operators of this notation are sequencing and production construction (*cf.* `->` in SDF). For SDF grammars the value of n_1 lies between 4 and 9, indicating that not all grammar authors take advantage of SDF's full operator repertoire. Among these, the grammars for AT&T SDL and C from the Grammar Base score highest (n_1=9), by using nested sequences and alternatives, optionals, and four operators for separated and non-separated repetition. None of the SDF grammars use the epsilon operator (`()`), repetition with an explicit lower bound (e.g. `A 2+` or `{A ";"}3+`), sets, pairs, functions, or permutations (*cf.* Table 3). The total number of operators and operands (N), seems significantly correlated with the number of non-terminals (VAR), as illustrated by Figure 6. Semantic Designs' C^{++} grammar scores highest, with a value of 12670, and seems to be an outlier regarding this correlation.

8.4 Comparing disambiguation metrics

Disambiguation metrics have been computed for the SDF grammars only. The data is shown in Table 10, again ordered by Halstead effort (E), though this metric is not shown.

The VB.Net grammar we measured is from a project in progress, and has not been fully disambiguated yet. This explains the absence of associativity attributes and the low number of follow restrictions (FRST=1) and reject productions (REJP=3).

Grammar	u_1	u_2	n_1	n_2	n	N	V	D	E	L	T
Yacc	7	37	85	92	44	177	966	8.70	8.4	0.11	467
BibTex	6	24	79	87	30	166	815	10.88	8.9	9.2e-2	492
Fortran	6	57	148	182	63	330	1973	9.58	19	0.10	1050
Toolbus	5	67	174	228	72	402	2480	8.51	21	0.12	1172
C (pm)									51		
Stratego	6	91	346	440	97	786	5188	14.51	75	6.9e-2	4180
Sdf (2.4)	7	137	379	496	144	875	6274	12.67	79	7.9e-02	4417
ECMAScript	2	250	879	1211	252	2090	16673	4.84	81	0.21	4487
Risla	6	124	424	516	130	940	6601	12.48	82	8.0e-2	4578
Sdf (2.3)	7	141	389	512	148	901	6496	12.71	83	7.9e-2	4586
Casl	4	174	582	794	178	1376	10287	9.13	94	0.11	5216
Java (pm)									95		
PHP	2	286	1005	1403	288	2408	19673	4.91	97	0.20	5362
Java (gb)	7	206	565	759	213	1324	10241	12.90	132	7.8e-2	7337
AT&T SDL	9	182	504	626	191	1130	8562	15.48	133	6.5e-2	7363
Java (sd)	2	251	1142	1588	253	2730	21794	6.33	138	0.19	7660
C (gb)	9	166	486	632	175	1118	8330	17.13	143	5.8e-02	7929
C^{++}(pm)									173		
DB2/SQL	7	326	872	1063	333	1935	16214	11.41	185	8.8e-2	10280
$C^{\#}$									228		
VDM-SL	8	214	788	968	222	1756	13687	18.09	248	5.5e-2	13758
VB.Net	6	402	1324	1571	408	2895	25107	11.72	294	8.5e-2	16353
Cobol (sfi)	5	831	1971	2397	836	4368	42402	7.21	306	0.14	16987
Verilog	2	734	3341	4487	736	7828	74550	6.11	456	0.16	25318
Cobol (gb)	7	565	2573	2155	572	4728	43308	13.35	578	7.5e-2	32119
PL/SQL	7	955	2512	3343	962	5855	58022	12.25	711	8.2e-2	39493
C^{++}(sd)	2	676	5292	7378	678	12670	119163	10.91	1301	9.2e-02	72254

Table 9. Values of all the Halstead metrics for SDF and DMS grammars. The values of the Halstead effort metric E for BNF grammars are reproduced from [20].

With large margins, our VDM-SL grammar scores highest in terms of associativity attributes (ASSOC=35) and priorities (UPP=72). This can be explained from the exceptionally large range of built-in operators of the VDM-SL language, which need disambiguation. The absence of associativity attributes and priorities (ASSOC=UPP=0) in the two Cobol grammars and the C grammar is due to the adoption of Yacc-style disambiguation, where as many hierarchically layered expression non-terminals are introduced as there are operator precedences. In the Yacc grammar, which is quite small, the absence of all disambiguation constructs except follow restrictions is explained simply by the lack of nestable operators in the language.

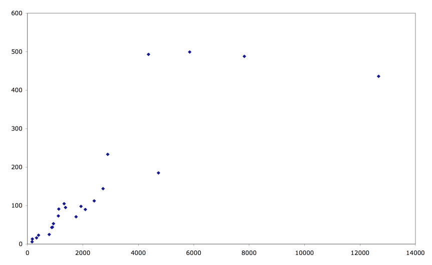

Fig. 6. Scatter plot that shows likely correlation between the number of non-terminals (VAR) and Halstead's total number of operators and operands (N).

Grammar	FRST	ASSOC	REJP	UPP	PREF
Yacc	8	0	0	0	0
BibTex	4	1	3	0	0
Fortran	1	5	0	10	4
Toolbus	2	4	0	4	1
Stratego	6	8	27	15	0
Sdf (2.4)	6	8	40	13	2
Risla	4	11	15	17	0
Sdf (2.3)	6	8	46	13	1
Casl	5	1	23	4	1
Java (gb)	9	12	48	24	4
AT&T SDL	6	7	49	2	0
C (gb)	8	0	33	0	1
DB2/SQL	9	7	3	7	8
VDM-SL	4	35	99	72	4
VB.Net	1	0	3	10	0
Cobol (sfi)	5	0	320	0	0
Cobol (gb)	7	0	701	0	0
PL/SQL	16	9	414	8	31

Table 10. Values of disambiguation metrics for SDF grammars.

The large values for the number of reject productions (REJP) for the two Cobol grammars and the PL/SQL grammars indicate that these languages contain high numbers of reserved keywords. The difference of rougly a factor two in values (REJP=320 and 701) between the two Cobol grammars is due to a simple refactoring. The version from the Grammar Base rejects each keyword for two non-terminals, leading to two reject productions for each reserved keyword. The refactored version from the Strafunski distribution defines on of the non-terminals in question in terms of the other, and therefore needs only a single reject per keyword.

The `prefer` and `avoid` attributes of SDF are most intensively used in the PL/SQL grammar. Many of these are placed on *context-free* productions that introduce keywords which are explicitly rejected from identifier nonterminals at the *lexical* level. A possible motivation for this encoding could be that it facilitates recognition of these keywords in the abstract syntax tree produced after parsing.

9 Related work

As said, the starting point of our work has been the 10 grammar metrics formally defined by Power *et al.* in [20]. We have adapted their metrics to SDF where appropriate, and we have introduced further SDF-specific metrics, for measuring disambiguation constructs in particular. We have adopted their tree impurity metric for transitive successor graphs (TIMP), and we have proposed a second tree impurity metric, for immediate successor graphs (TIMPi). They presented values of their 10 grammar metrics for 4 grammars. We provided values for 29 grammar metrics of 23 additional grammars. Whereas the grammars sampled by them all fall within the family of C-like programming languages, our grammars cover a wider variety of languages, including mainframe legacy languages, specification languages, scripting languages, and domain-specific languages.

Power *et al.* implemented the SynQ tool to measure the metrics. They present the tool's architecture and some implementation details. The implementation language is C^{++}, using the visitor design pattern for AST walking. The graph algorithms use a matrix representation of graphs. The architecture of our tool SdfMetz is similar, though very different on the implementation level. We implemented SdfMetz in the functional programming language Haskell, using strategic programming techniques [17]. The underlying graph representation is a finite map-based set of pairs. Since grammar graphs are generally sparse, these implementation choices have proven to allow excellent performance.

Malloy *et al.* have applied various software engineering techniques during the development of a LALR parser for $C^{\#}$ [18]. Their techniques include versioning, testing, and the grammar size, complexity, and structure metrics that we adopted [20]. They do not measure coverage. We have used a similar suite of techniques, including grammar metrication, for metrics-based monitoring of a grammar development project for the VDM-SL language. We report on this elsewhere [2, 3].

Lämmel et. al. have advocated derivation of grammars from language reference documents through a semi-automatic transformational process [15, 14]. In particular, they have applied their techniques to recover the VS COBOL II grammar from railroad diagrams in an IBM reference manual. They use metrication on grammars, though less extensive than we. Coverage measurement nor unit tests are reported.

Klint et. al. provide an survey over grammar engineering techniques and a agenda for grammar engineering research [11]. Our work fits into this agenda.

10 Concluding remarks

We have shown that the grammar metrics of Power *et al.* [20] and the coverage metrics of Purdom [21] can be generalized to richer syntax notations, and to SDF in particular. In addition, we have defined some SDF-specific metrics, for measuring disambiguation constructs. We have implemented these generalized grammar metrics in the SdfMetz and SdfCoverage tools. We have applied these tools to perform measurements on a suite of medium to large-sized grammars from various academic and industrial origins, and we have presented the collected data side by side with previously published data for BNF grammars.

10.1 Future work

The validation of grammar metrics, their connection to external grammar attributes (e.g. quality, modifiability, modularizability), and the embedding of their use in grammar engineering processes are important topics that go beyond the scope of this report. Definition of metrics and data collection, as presented here, are initial steps to enable addressing these issues. An important next step is comprehensive statistical analysis of measurement results. Such analysis should shed light on questions regarding how well-behaved the metrics are and to what extent they are independent of each other. The suite of sampled grammars should be further extended to ensure the validity and representativeness of such statistical analysis.

Our SdfMetz tool currently accepts two grammar notations: SDF and DMS. We intend to add front-ends for common grammar notations, such as Yacc, ANTLR, and various BNF dialects commonly employed in language reference documents. Apart from grammar notations *per se*, we consider providing front-ends for grammar-like schema notations, such as XSD.

Quantitative data about grammars should be related to nominal data to guide the interpretation of metrics values. In table 7, nominal data is presented for a tentative initial set of nominal external attributes (grammar *origin*, the *purpose* and *paradigm* of the language they generate) and a nominal internal attribute (*notation*). It could be worthwhile to elaborate this set into a more comprehensive questionnaire to be used in a wider 'grammar census'.

Lämmel generalized the notion of rule coverage and advocates the uses of coverage analysis in grammar development [13]. SDF tool support for his *context-dependent* rule coverage metric has yet to be developed.

Availability Source code distributions of the SdfMetz and SdfCoverage tools are freely available from the web pages of the authors. From the same pages, all data presented in Section 8 is available in raw form.

Acknowledgments We thank Tobias Kuipers of the Software Improvement Group and Ira Baxter of Semantic Designs for graciously allowing us to perform measurement on grammars developed by their respective companies.

References

1. A. Abran, J.W. Moore, P. Bourque, and R. Dupuis (*eds.*). Guide to the Software Engineering Body of Knowledge - SWEBOK, Version 2004. IEEE Computer Society, www.swebok.org.
2. T. Alves and J. Visser. Development of an industrial strength grammar for VDM. Technical Report DI-PURe-05.04.29, Universidade do Minho, 2005.
3. T. Alves and J. Visser. Grammar engineering applied for development of a VDM grammar. Manuscript in preparation, 2005.
4. I. Baxter, P. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *Proceedings of the International Conference on Software Engineering*. IEEE Press, 2004.
5. R. Dumke. Software metrics - a subdivided bibliography. http://irb.cs.uni-magdeburg.de/sw-eng/us/bibliography/bib_main.shtml.
6. N. Fenton and S.L. Pfleeger. *Software metrics: a rigorous and practical approach*. PWS Publishing Co., Boston, MA, USA, 1997. 2nd edition, revised printing.
7. The online Grammar Base. http://www.cs.uu.nl/ mdejonge/grammar-base/.
8. M.H. Halstead. *Elements of Software Science*, volume 7 of *Operating, and Programming Systems Series*. Elsevier, New York, NY, 1977.
9. J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
10. M. de Jonge and J. Visser. Grammars as contracts. In *Proceedings of the Second International Conference on Generative and Component-based Software Engineering (GCSE 2000)*, volume 2177 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2000.

11. P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. Draft, Submitted for journal publication; Online since July 2003, 42 pages, July11 2004.
12. E. Koutsofios. Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, November 1996.
13. R. Lämmel. Grammar Testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.
14. R. Lämmel. The Amsterdam toolkit for language archaeology (Extended Abstract). In *Proceedings of the 2nd International Workshop on Meta-Models, Schemas and Grammars for Reverse Engineering (ATEM 2004)*, October 2004.
15. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.
16. R. Lämmel and J. Visser. Strafunski home page. http://www.cs.vu.nl/Strafunski/.
17. R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.
18. B.A. Malloy, J.F. Power, and J.T. Waldron. Applying software engineering techniques to parser design: the development of a C# parser. In *SAICSIT '02: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 75–82. South African Institute for Computer Scientists and Information Technologists, 2002.
19. PGEN: Parse table generator home page. http://www.cwi.nl/projects/MetaEnv/pgen.
20. J.F. Power and B.A. Malloy. A metrics suite for grammar-based software. In *Journal of Software Maintenance and Evolution*, volume 16, pages 405–426. Wiley, November 2004.
21. P. Purdom. Erratum: "A Sentence Generator for Testing Parsers" [BIT **12**(3), 1972, p. 372]. *BIT*, 12(4):595–595, 1972.
22. J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992.
23. Semantic Designs, Inc., home page. http://www.semdesigns.com/.
24. Software Improvement Group home page. http://www.sig.nl/.
25. M.G.J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.
26. E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997.
27. A.H. Watson and T.J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. Technical Report 500-235, NIST Computer Systems Laboratory, 1996.