AGile, a structured editor, analyzer, metric evaluator, and transformer for Attribute Grammars

André Rocha, André Santos, Daniel Rocha, Hélder Silva, Jorge Mendes, José Freitas, Márcio Coelho, Miguel Regedor[2], Daniela da Cruz, and Pedro Rangel Henriques[1]

[1] University of Minho - Department of Computer Science,
Campus de Gualtar, 4715-057, Braga, Portugal
{danieladacruz,prh}@di.uminho.pt
[2] University of Minho - Department of Computer Science,
Master Course on Language and Grammar Engineering (1.st year)

Abstract. As edit, analyze, measure or transform attribute grammars by hand is an exhaustive task, it would be great if it could be automatized, specially for those who work in Language Engineering.
However, currently there are not editors oriented to grammar development that cover all our needs.
In this paper we describe the architecture and the development stages of AGile, a structured editor, analyzer, metric calculator and transformer for attribute grammars. It is intended, with this tool, to fill the existing gap.
An AnTLR based attribute grammar syntax was used to define the input for this system. As soon as the user types the grammar, the input is parsed and kept in an intermediate structure in memory which holds the important information about the input grammar. This intermediate structure can be used to calculate all the metrics or to transform the input grammar.
This system can be a valorous tool for those who need to improve the performance or functionalities of their language processor, speeding up the difficult task of defining and managing a language. Features like highlighting, automatic indentation, on-the-fly error detection, etc., also add efficiency.

1 Introduction

Editing an Attribute Grammar is a long and complex hard task. So it is important to provide basic support to make the edition easier. However, even more important is to assist the language engineering in designing and developing a language with quality. This requires that the grammar development environment (GDE) incorporates knowledge from grammar engineering, like analysis, visualization, metric evaluation, slicing, and transformation.

This article describes the development of AGile, a text editor built to assist in the grammar writing process, comprising four main features as follows:

- *Syntax and lexical awareness*: syntax highlighting, automatic indentation, lexical integrity checking, syntactic integrity checking;
- *Metrics evaluation* for quality assessment, namely:

    - Derivation Rules analysis: both size and structural parameters should be measured and metrics evaluated;
    - Attribute analysis: evaluate attribute-related metrics and other tasks (generate dependency graph,...)
- *Derivation Rules transformation*: transformations upon types of rules, namely eliminating unitary rules, eliminate left/right recursion, ...;
- *Visualization*: dependency graphs, for both the syntactic and the semantics components, will be built and displayed.

AGile GDE is a project under development in the context of a master course on Language Engineering. The motivation for this work is the application of all the theoretical concepts on grammar engineering (quality assessment and metrics) and on software analysis and transformation (analyzers, internal representations, transformers and visualizers).

The paper has, after this, 7 sections. Section 2 gives an overview of the GDE, presenting and describing AGile architecture. Section 3 describes the syntax-directed editor. Section 4 explains how the analyzer module transforms the input grammar into an intermediate representation (IR) suitable for the processing. Section 5 discusses the metrics evaluated by AGile. Section 6 introduces the rule transformations implemented by the GDE proposed. Section 7 very briefly present the visualization provided at moment. As usual, Section 8 closes the paper.

2 AGile architecture

In this section we describe the architecture of AGile, enhancing the most important functional blocks and the information that flows between them.

When the user is editing the grammar, he can take advantage of features like syntax highlighting and automatic indentation, that clearly facilitate the edition.

AGile parses the input grammar to check its syntactic and semantics compliance with the meta-language defined (in our case, we have adopted AnTLR [1] meta-language syntax); if errors are detected during this phase, descriptive messages will be generated.

Other important AGile components are the metrics evaluator, the grammar transformation and the visualizer. To support all these operations, the system generates, during parsing, an attributed abstract syntax tree (AST) and creates an Identifier Table (IdTab)). The operations referred above will be executed on this intermediate representation.

Figure 1 depicts AGile architecture.

The component blocs—editor, analyzer, metrics evaluator, transformer, and visualizer—will be detailed in following sections.

3 Editor

The first component of AGile is a text editor. Editors are tools that give us the ability to collect (type), prepare (delete, replace, substitute, add) and arrange (indent, itemize,

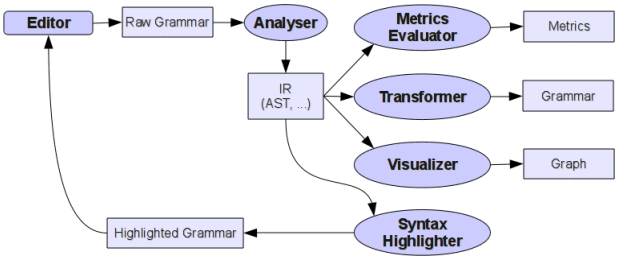

Fig. 1. AGile architecture

etc.) material for storing objects in a computer to be processed afterwards (published, searched, transformed, . . .).

In our project the objects we want to edit are structured documents instead of unstructured text; actually, we need to process structured texts representing grammars. So we decided to developed an editor that could help to create and maintain that structure.

Structure text editors can be classified into different categories [2–5]:

- **structure-aware editors** — that have knowledge about the file structure and provide an interface where the user can see and manipulate that structure; no special rules are checked but the hierarchy of text components is explicit and guides the edition.
- **syntax-directed editors** — that have knowledge about the grammar of the language being able to validate the syntax or even the semantics of text during a free file manipulation, and offering extra information and editing guidelines.
- **language-based editors** — that also know the grammar of the language but do not allow free edition; on the other way around, these editors guide de edition according to grammar syntactic and semantic rules; actually, the user manipulates the internal representation of the object under edition and what he sees is just one possible view of that object, obtained by the application of unparsing rules to the intermediate representation.

Our editor falls on the second category and it treats grammar rules in an analytical way, meaning that we are using an intermediate representation to offer the user the functionality described bellow.

To help in the process of writing a grammar, our editor uses the knowledge about the meta-grammar to offer the following set of features, as can be seen in Figure 2:

Line numbering: It is specially useful for debugging, to allow a quick identification of the line where the error was detected.

Syntax highlighting: The editor sends the text to the analyzer to produce an intermediate structure obtained from the abstract syntax tree generated in the parsing process (more details in the next section); then this structure is traversed producing tokens colored differently according to their lexical/syntactic role.

automatic indention of the text according to the hierarchical relationship between components.

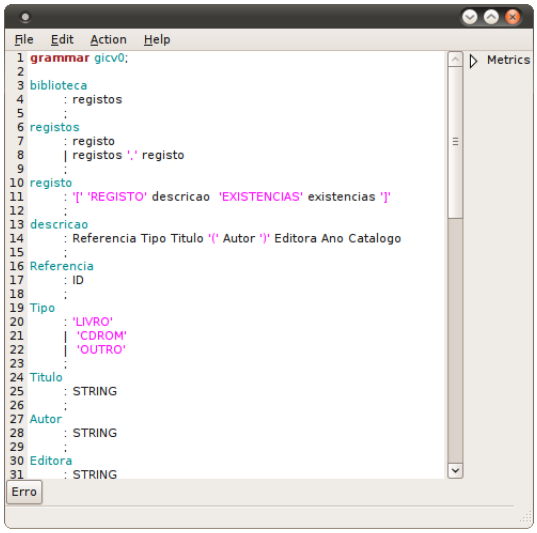

Fig. 2. AGile Editor: an illustrative screenshot

4 Analyzer

After creating or modifying a grammar, it is necessary to process that grammar. For that, the textual description entered with the Editor must be checked for correctness and, if it is a valid grammar specification, it should be transformed into a convenient internal representation.

This is the objective of the second AGile component that analysis the raw grammar transforming it into the chosen intermediate structure, an abstract syntax tree (AST). Figure 3 shows the structure of the Analyzer implemented, and Figure 4 describes the intermediate representation of data that flows between components.

AnTLR compiler generator [6–8] was used to develop this component.

4.1 Lexer

As everybody knows, a Lexical Analyzer, or simply a Lexer, transforms one character or a sequence of characters into tokens, to be further used by the Parser. The main purpose of the Lexer is to match the raw grammar with our AnTLR-based grammar for grammars, generating the set of tokens if the matching is successful or returning an error otherwise.

Assuming that the matching returns successfully the next step of this component its the Parser.

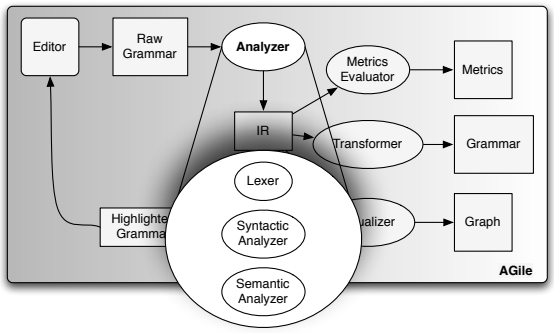

Fig. 3. AGile architecture: the Analyzer component

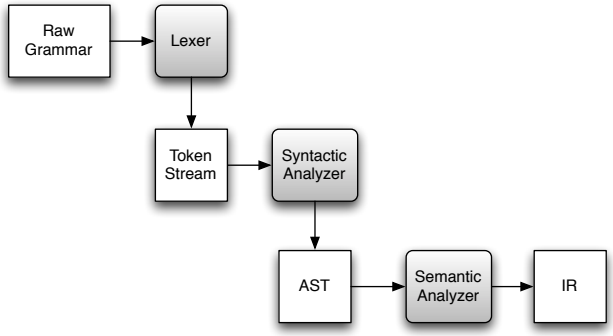

Fig. 4. AGile Analyzer dataflow

4.2 Parser

Having now the set of tokens generated by the Lexer, the in-memory construction of an AST is delegated to the Syntactic Analyzer, or simply Parser, that will group tokens finding non-terminal symbols according to the meta-grammar productions. AnTLR allows us to add to the meta-grammar productions rules to specify how to build the most convenient AST.

After generating this AST everything is prepared to continue to the next analysis step: the semantic analysis.

4.3 Semantic Analyzer

Parsing only verifies that the input grammar consists of tokens arranged in a syntactically valid combination. However, syntactic rules do not specify that names can not be repeated, because this is not important from a structural point of view; but such a

constraint is relevant for further processing (translation, etc.), this is from a semantic perspective.

So, the semantic analyzer, activated after the parser, traverses the syntax tree: (i) to compute the attributes in each node to associate semantic values to the symbols; and (ii) to verify if all nodes are compliant with the respective semantic rules that constraint attribute values.

In our editor, the following semantic rule are verified:

- symbol/attribute names can not be duplicated;
- a symbol/attribute name can not be used to name a symbol/attribute in another class;
- terminal symbols should be defined using regular expression;
- attributes should have a valid type.

After all these steps, that guarantee that the raw grammar is syntactic and semantically correct, and Intermediate Representation of this grammar is constructed, as can be seen in Figure 4.

5 Metrics Evaluator

Among other factors that affect the software quality, its complexity has a large influence, and thus part of the referred metrics were designed to assess the software complexity. If the software is very complex, this complexity potentiates run-time errors; to fix those failures, a significant maintenance effort will be required. Avoiding that effort before software delivery, saves time and money. This statement justifies the importance of measuring complexity and quality during software development.

In the context of grammar quality, Power and Malloy have defined in [9] a set of metrics, derived from the above mentioned software metrics. They categorize the metrics in two types: *size metrics* (adapted from the standard metrics for programs and procedures described in [10] and [11, 12]); and *structural metrics* (adapted from those used to measure discretional complexity of context free grammars, as presented in [13]).

In this paper, we do not follow strictly Power and Malloy's approach. We took there classification as a basis, but we go further, defining a set of 10 metrics classified in 3 types: size metrics (2); shape metrics (4); and lexicography metrics (4).

One of the main components of AGile is the evaluation of size metrics on a context-free grammar, edited under this grammar processing environment.

Notice that we consider that a grammar is a four-tuple (N, T, S, P), where N and T are disjoint sets of symbols, known as non-terminal and terminal, respectively. S is a distinguished element of N known as the start symbol or axiom of the grammar. P is a relation between elements of N and $(N \bigcup T)^*$; the elements (pairs) of P are known as the production rules.

The first set of metrics is related with the size of the grammar G, and the second one is concerned with the size the Parser derived from the grammar G.

The following table presents the size metrics evaluated by AGile.

Size Metric	Description
#T	Number of terminal symbols
#NT	Number of non-terminal symbols
#P	Number of productions
#PU	Number of unitary productions
**$RHS**$_{avg}$	Average size of the right hand side
**$Alt**$_{avg}$	Average number of alternatives for each symbol
Fan-in & Fan-out	Number of dependencies between symbols from the dependency graph
#TabLL(1)	Size of the LL(1) parse table
#RD	Size of the Recursive Descendant parser

Table 1. Description of the size metrics implemented in AGile.

In this way the user obtains easily and in an interactive mode immediate feedback about his grammar. This way, the user is aware of the options that is doing when constructing the grammar.

In Table 2 we show the metrics evaluated by AGile for the grammar defined in Table 3.

Size Metric	Value obtained
#T	21
#NT	20
#P	28
#PU	9
**$RHS**$_{avg}$	1.96
**$Alt**$_{avg}$	1.4
#TabLL(1)	420
#RD	21

Table 2. Values of the size metrics evaluated by the AGile for the *Biblioteca* grammar.

6 Transformer

When we are writing a grammar for a certain language, the one that we wrote is not always the best when taking into account factors like: grammar metrics, grammar comprehension, or even the efficiency of the future parser. In this context, it is important to consider some transformation techniques in order to make our grammar a better one.

In AGile were introduced two different types of transformations.

1. The *replacement of the name* of a Terminal, a Non-Terminal, or Attributes.
   This transformation can increase significantly the clarity of the grammar, making easier the comprehension and maintenance tasks – it increases the grammar quality. When we have a long grammar and we want to change the name of a symbol that is mentioned in a lot of productions, it would be a time consuming and exhausting

biblioteca	→ registos
registos	→ registo
	| registos ',' registo
registo	→ '[' 'REGISTO' descricao 'EXISTENCIAS' existencias ']'
descricao	→ Referencia Tipo Titulo '(' Autor ')' Editora Ano Catalogo
Referencia	→ ID
Tipo	→ 'LIVRO'
	| 'CDROM'
	| 'OUTRO'
Titulo	→ STRING
Autor	→ STRING
Editora	→ STRING
Catalogo	→ 'BGUM'
	| 'ALFA'
	| 'OUTRO'
existencias	→ 'LOCAL' Local '(' estados ')'
Local	→ STRING
estados	→ estado
	| estados ',' estado
estado	→ Codigobarras disponib
Codigobarras	→ ID
disponib	→ 'ESTANTE'
	| 'PERMANENTE'
	| 'EMPRESTADO' datadev
datadev	→ Ano '-' Mes '-' Dia
Ano	→ INT
Mes	→ INT
Dia	→ INT

Table 3. The *Biblioteca* grammar.

task to change all of them by hand. Automatizing it, reduces effort and at the end assures a complete replacement.

2. The *elimination of Unitary Productions*.
   This transformation produces a grammar more difficult to understand, but the advantage is that from the reduced grammar (with less Productions and Non-Terminals) we can generate a more efficient Processor.

The way how these transformations are done is the same: the intermediate representation created by the Analyzer module (remember Section 4) is traversed taking into consideration the transformation that we want to perform as explained in the following paragraphs.

If we want to rename a symbol: if it is a Non-Terminal we search for the productions that hold that symbol in their left-hand side and rename it; then we search in every right-hand side its occurrence and rename it (the same process for Terminals). For attribute renaming we apply a similar algorithm.

If we want to remove all the Unitary productions, the first thing to do is to identify the set of such rules. Choosing one of the rules on that set, next step is to find the Non-Terminal on the left, in the right-hand side of the other grammar productions, replacing them with the symbol on the respective right-hand side of the Unitary production to remove.

This process is repeated until there is no more Unitary productions. In the end we obtain a small and simplified grammar.

AGile computes automatically the metrics for the new grammar, allowing the user to compare both version (the original and the transformed).

At this moment, these are the available transformations, but the architecture conceived for the tool allows us to add other transformations, simply by extending the interface with a set of new traversal methods that implement the new transformation.

This AGile feature has another outcome: with these transformation techniques we have no longer need to write non-readable grammars. We can use our tool to transform them in a more efficient one as we can automatize this process. This way, we can preserve two versions of the grammar: one to specific the language (the version that is used by the language engineer for maintenance) and another one to generate the processor.

7 Visualizer

To complement the comprehension of the grammar under development, this AGile component produces and displays visual representation of all or parts of the intermediate representation.

It is well known from the literature and practice that the visualization of dependency graphs offers an effective help for program comprehension. Extending the same idea to grammars, our GDE also offers a functionality to show the Dependency Graphs between Symbols or Attributes as illustrated in Figure 5.

Figure 6 shows the Dependency Graph for the Biblioteca grammar used in example of Section 5 (see its definition in Table 3).

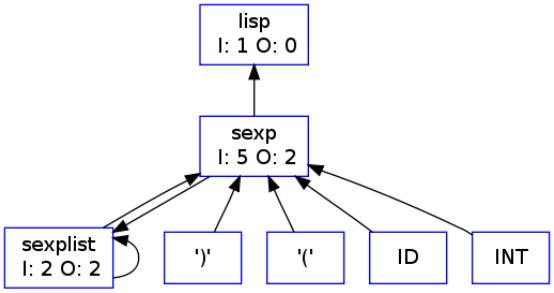

Fig. 5. AGile Visualizer: Dependency Graph for Lisp grammar

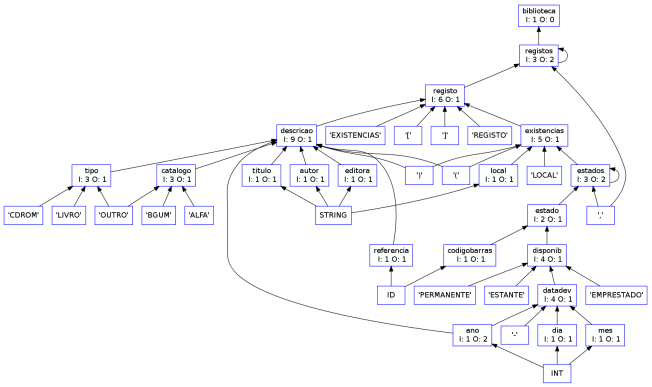

Fig. 6. AGile Visualizer: Dependency Graph for Biblioteca grammar

8 Conclusion

This paper introduced AGile, a structured editor, analyzer, metric calculator and transformer for attribute grammars. Mainly, this tool allows the user to write a grammar, based on its specific syntax, and apply several actions over it.

Features like syntactic-directed edition, metrics evaluation and form transformations make AGile a useful tool for studying and formal grammars.

According to [14], grammar metrics have been introduced to measure the quality and complexity of a given grammar in order to direct the grammar engineering. Computing metrics since the early stages of the development of a given grammar, allows to improve the grammar and to avoid some undesirable features that otherwise would only be perceived later on.

Dependency graphs, for syntactic dependencies between grammar symbols or semantic ones between attributes, can also be generated and displayed to simplify the understanding of the grammar edited.

A lot of work still needs to be done to realize all our ideas, however the theoretical foundations are established and the prototype is promising.

References

1. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. First edn. Pragmatic Programmers. Pragmatic Bookshelf (Maio 2007)
2. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B.: Programming environments based on structured editors: the mentor experience. Rapport de Recherche 26, INRIA, Rocquencourt (July 1980)
3. Teitelbaum, T., Reps, T.: The cornell program synthesizer: A syntax-directed programming environment. Communications of the ACM **24**(9) (Setembro 1981)
4. Reps, T.: Generating Language-Based Environments. PhD thesis, Cornell University (1982)
5. Reps, T., Teitelbaum, T.: The Synthesizer Generator: A System for Constructing Language-Based Editors. Texts and Monographs in Computer Science. Springer-Verlag (1989)
6. Parr, T., Quong, R.W.: Antlr: A predicated-ll(k) parser generator. Software Practice and Experience **25**(7) (July 1995) 789–810
7. Parr, T.: Practical computer language recognition and translation – a guide for building source-to-source translators with antlr and java. http://www.antlr.org/book/index.html (1999)
8. Parr, T.: An introduction to antlr. http://www.cs.usfca.edu/ parrt/-course/652/lectures/antlr.html (Jun. 2005)
9. Power, J.F., Malloy, B.A.: A metrics suite for grammar-based software: Research articles. J. Softw. Maint. Evol. **16**(6) (2004) 405–426
10. Hoare, C.A.R.: Hints on programming language design. Technical report, Stanford, CA, USA (1973)
11. Fenton, N.E.: Software Metrics: A Rigorous Approach. Chapman & Hall, Ltd., London, UK, UK (1991)
12. Coleman, D., Ash, D., Lowther, B., Oman, P.: Using metrics to evaluate software system maintainability. Computer **27**(8) 44–49
13. Power, J.F., Malloy, B.A.: Metric-based analysis of context-free grammars. In: IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension, Washington, DC, USA, IEEE Computer Society (2000) 171
14. Cervelle, J., Crepinsek, M., Forax, R., Kosar, T., Mernik, M., Roussel, G.: On defining quality based grammar metrics. In: Proceedings of the International Multiconference on Computer Science and Information Technology – 2nd Workshop on Advances in Programming Languages (WAPL'2009), Mragowo, Poland, IEEE Computer Society Press (October 2009) 651–658