EGRAM - A GRAMMAR DEVELOPMENT ENVIRONMENT AND ITS USAGE FOR LANGUAGE GENERATION

Stephan Busemann

DFKI GmbH
Stuhlsatzenhausweg 3, D-66123 Saarbrücken
busemann@dfki.de

Abstract

The development of large grammars is inherently complex and can hardly be achieved using standard text editors. Although, e.g., emacs can be programmed to support this task to a certain extent, special-purpose functionalities are indispensable. Otherwise the increasing effort for the development and maintenance of large grammars may severely limit their applicability. To avoid this pitfall in the field of language generation, eGram has been developed, which provides a developer-friendly grammar format, syntactic verification of grammar knowledge, abbreviations through metarules, and integration with grammar testing. eGram is a implemented in Java and available under research or commercial licences.

1. INTRODUCTION

The development of large grammars is inherently complex and can hardly be achieved using standard text editors. Although, e.g., emacs can be programmed to support this task to a certain extent by defining dedicated "modes", special-purpose functionalities are indispensable, and a graphical user interface is mandatory for many users. Otherwise the increasing effort for the development and maintenance of large grammars consisting of several hundreds or thousands of rules may severely limit their applicability.

Clearly, small grammars with 100 to 200 rules such as the ones underlying the generation systems in (Busemann, 1996) and (Busemann and Horacek, 1998) could safely be developed with standard text editors using the syntax exemplified in Figure 1 below. However even in this work, the lack of maintenance support and a considerable errorproneness were observed.

To avoid this pitfall in the field of language generation, the dedicated grammar development environment eGram has been developed. With eGram the implementation of a large grammar of a subset of German, which enabled the generation of cross-lingual summary texts of medical scientific reports from non-linguistic representations (Lenci et al., 2002; Busemann, 2002), was achieved in a comfortable and efficient way. The grammar comprises about 950 rules with 135 categories, 134 test predicates, many access path descriptions, and 14 features for constraints.

Major benefits of eGram include

- a developer-friendly grammar format (Section 2.),
- syntactic and semantic verification of grammar knowledge (Section 3.),
- abbreviations through metarules (Section4.), and
- integration with grammar testing (Section5.).

2. GRAMMAR FORMATS

Grammar formats used by processing components are often idiosyncratic and difficult to cope with. The editor of YAG requires the grammar writer to define Lisp expressions (McRoy et al., 2000). This may create considerable difficulties for linguists not used to bracket languages. Quite differently, eGram takes on the functionality of compiling its own, developer-friendly format into the one needed by generation systems. Currently, two so-called shallow language generation systems are supported: TG/2 implemented in Lisp (Busemann, 1996) and XtraGen (Stenzhorn, 2003), which is a sister Java implementation of TG/2.

The generation grammar formalism supported by eGram is based on the free combinations of pieces of prefabricated text and non-terminal categories on the right-hand side (RHS), thus implementing a continuum between classical templates and context-free rules. The format is augmented by feature-value pairs that can be percolated through the derivation tree to control agreement relations. The applicability of a rule is subject to boolean tests on the generation input being fulfilled (cf. (Busemann, 1996)). Thus the rules correspond to condition-action pairs, or production rules.

For instance, the rule in Figure 1 is applicable to a given input if the category to be generated from is `DECL` and if the input follows some pattern called `s2`, if it specifies a "deep subject", and if active voice is acceptable. The rule has four RHS elements, which define an argument (the subject), the finite verb, another argument (the direct object) and optionally an infinite verb constituent. The part of the input feature structure relevant when applying a particular rule is accessed using the feature path descriptions derived from expressions like `'deep-subj`. The rule has the context-free backbone (`DECL -> ARG, FIN, ARG, {INF}`). The variables starting with `X` determine the assignment of the feature constraints to the RHS elements. For instance, the subject and the finite verb agree in number and person. The rule can be used with transitive verbs, as in *Die Firma hatte 364 Arbeiter beschäftigt.* [The company had employed 364 workers].

Such a generation step forms an element of the following algorithm, which consists of the classical three-step interpretation cycle of production systems. Generation starts from a category C and a (piece of) input structure. First, a

```
(defproduction "s2 top-subj.1"
    (:PRECOND (:CAT DECL
               :TEST ((sbp ’s2) (top-deep-subj ’y) (vc-voice ’active)))
     :ACTIONS (:TEMPLATE (X1 :RULE ARG ’deep-subj)
                         (X2 :RULE FIN ’verb-complex)
                         (X4 :RULE ARG ’deep-acc-obj)
                         (X5 :OPTRULE INF ’verb-complex)
               :CONSTRAINTS ( X1.CASE := ’nom
                              X4.CASE := ’acc
                              X1.NUMBER = X2.NUMBER = X5.NUMBER
                              X1.PERSON = X2.PERSON = X5.PERSON ))))
```

Figure 1: A rule for the German transitive main clauses as processed by TG/2.

conflict set is identified by selecting all rules that have C on their LHS and whose tests are fulfilled. Second, one rule is selected from the conflict set. Third, the selected rule is applied by recursively generating from each RHS element, when C is set to the element's category, and the piece of input structure according to the element's path description is selected. If a generation step fails, the algorithm backtracks by selecting another rule from the conflict set. If the conflict set is empty, it backtracks to the next higher level.

It should be noted that the formalism boils down to context-free grammars when feature constraints and tests are left unspecified.

Further elements of the formalism also covered by eGram, but not discussed further in this paper, include

- use of interface functions in rules to trigger external components (e.g., morphological inflection),
- the specification of meta symbols expanding into either HTML, LaTeX or ASCII formatting directives,
- a preference assignment for the elements of the conflict set, guiding step 2 of the interpretation cycle.

3. DESIGN PRINCIPLES AND CONSISTENCY ISSUES

A major difficulty in the course of developing complex grammars is to maintain consistency. Every-day practice shows that features used are sometimes not defined, values are not sufficiently restricted, or certain categories do not occur in any other rule. When such grammars are interpreted, errors occur that can be difficult and time-consuming to trace. eGram verifies that every new piece of grammar knowledge is fully consistent with what already exists, thus eliminating many obvious sources of mistake.

eGram was designed to enforce a consistent way of defining grammar objects by allowing the definition of complex objects only after all their elements are defined. Before a rule may be entered, the categories, test predicates, access paths and constraints used must be defined. The GUI offers dynamically generated menus for more complex elements in addition to textual input windows, where these remain necessary. For the definition of e.g. a constraint, a menu would offer all defined features, and for the selected feature, all defined values. Definitions of test predicates must be entered as text.

Different working styles are supported: either the user pro-actively plans her work by first defining all low-level elements and then proceeding to higher-level ones, or she prefers to add missing elements "on the fly", i.e. when eGram complains.

Defining a set of similar elements such as categories or features quickly is supported by options that save definitions without closing the window, allowing existing entries to be reused for editing.

eGram's main pane contains a set of tabs corresponding to the different elements. Clicking on a tab opens a new screen with all the tabs remaining available at any moment (see Figure 2). A set of tabs opens separate subpanes allowing for the definition of the tests, RHS elements, and constraints of rules.

Moving from basic to more complex elements together with the dynamic menu based methodology minimises errors. Definitions are guaranteed to be syntactically complete.

Although correctness at the semantic level can not be ensured in general, eGram has built-in means to control some effects of definitions. For every element that occurs as part of other elements – e.g. a category occurring in different rules – the containing elements can be visualised. This way the rules applicable to a given category can be overviewed and inspected. In addition, derivations induced by the context-free backbone can be interactively expanded and visualized as a tree. The choice of applicable rules is left to the user. The effects of feature percolation through the derivation tree can be visualized by coloured links between the nodes involved. This way, the user can easily detect missing or ill-formed constraints.

4. METARULES

As the kind of grammars encoded in eGram involves a context-free backbone, rules cannot easily express certain linguistic phenomena, such as word order variation, pronominalization, voice, the relation between sentential structures and relative clauses, or verb positions. To express these phenomena, several hundreds if not thousands of different rules must be defined. Every-day practice involces copy-and-paste approaches that are error-prone. Moreover such phenomena are often captured only partially, leaving unexpected gaps in the coverage of the grammar.

eGram is equipped with a metarule mechanism that is technically similar to that of Generalized Phrase Structure

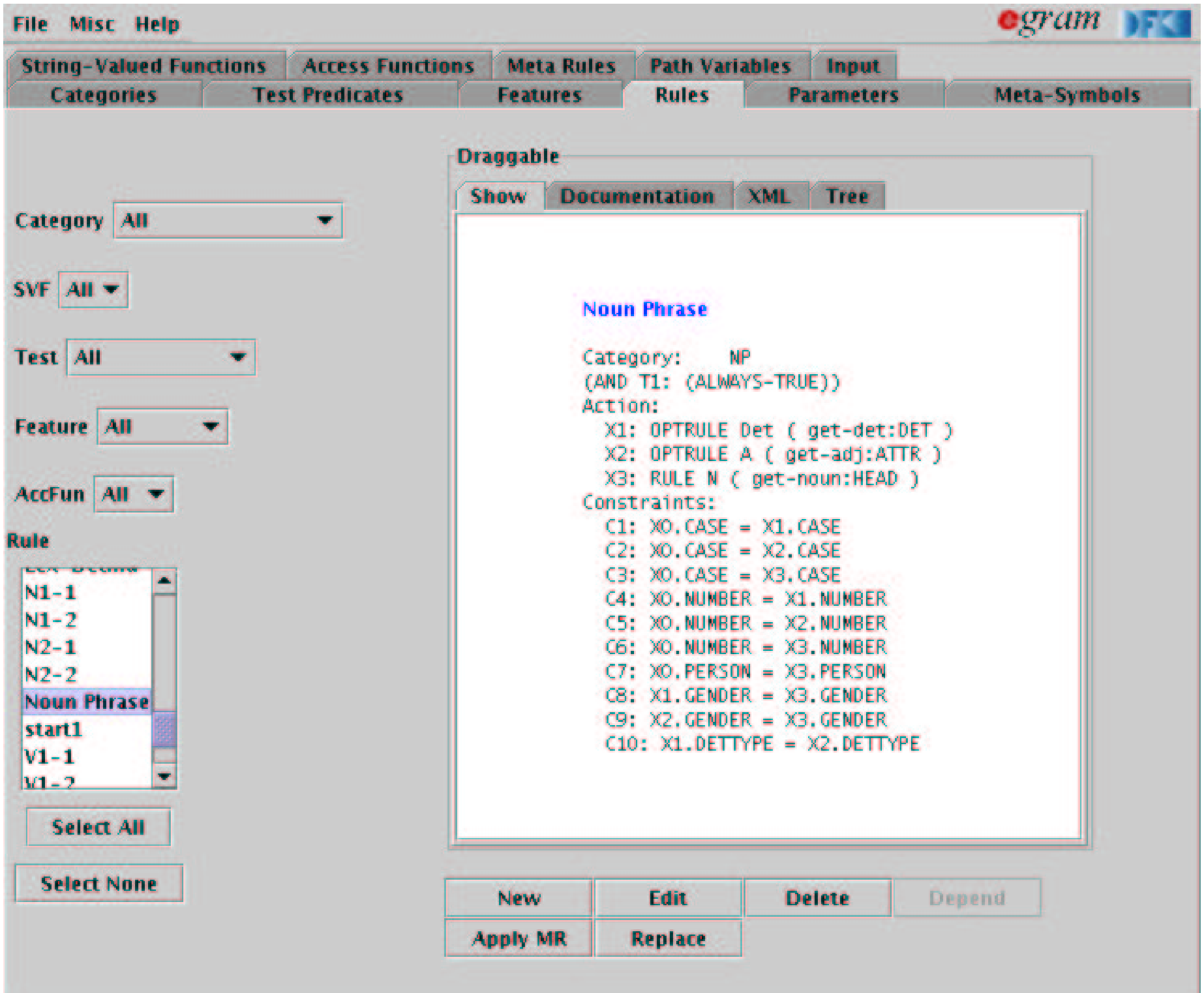

Figure 2: A Screenshot of eGram with the Rule Pane Active. It displays a simple NP rule for German. The feature constraints express various agreement relations. The variable X0 refers to the mother node. The rule window can be dragged to some other location on the screen, allowing to view multiple objects at the same time. The rules names shown on the left-hand side can be filtered by the elements contained in the rules. For instance by selecting category NP, only the rules with NP as their LHS category are shown.

Grammars (Gazdar et al., 1985). Metarule expansion starts with a set of base rules and then applies to the set of base rules and derived rules. The derivational history of a derived rule may contain each metarule at most once, thus guaranteeing termination of the expansion process. Like in GPSG, the metarule mechanism does not augment the power of the formalism, i.e. if the base rules are context-free, the set of derived rules will be context-free as well. Linguistic phenomena are conveniently encoded by base rules and metarules, leaving comparably less coding work for the grammar writer.

A metarule consists of a LHS defining a pattern for input rules, and a RHS specifying the resulting rule(s) for each matched input rule. Clearly the resulting rule will contain components of the input rule, but also skip components or introduce new components. Reused components are bound by variables on the LHS that are used on the RHS, skipped components are specified on the LHS and ignored on the RHS, and new components are specified only on the RHS.

The user can control the applicability of metarules by restricting the allowed derivation histories of derived rules. In order to suppress unwanted derivations, metarule application can be limited to base rules only or after some other metarules have been applied.

Derived duplicates should be removed since they do not add to the coverage of the grammar, but simply introduce spurious ambiguities. eGram recognizes duplicates and eliminates them. Rules differing only wrt. their tests are combined by using a disjunction on the tests.

Derived rules cannot be edited in eGram. Rather the underlying base rules or the metarules must be modified.

The basic metarule mechanisms and their integration into eGram are described in detail in (Rinck, 2003). A redesign of the grammar mentioned led to a reduction of the 950 rules to 569. Applying to these base rules 19 metarules modelling the above phenomena resulted in 2.435 derived rules, demonstrating that the original grammar did not systematically cover all the phenomena represented by the metarules.

5. INTEGRATION WITH GENERATION SYSTEMS

Although eGram knows about the logical dependencies between the elements of the formalism, can show the user in which parts of a grammar an element is used, supports the interactive generation of derivation trees these mechanisms cannot replace online testing through the generation components.

Integrating grammar development and grammar testing is crucial to verify the effects of modifying a grammar. eGram is integrated with TG/2 via a client-server interface and with XtraGen via a Java API. Since both systems use different input formats – XtraGen uses XML encodings of grammars, whereas TG/2 uses expressions as shown in Figure 1 – eGram provides suitable export formats for both. Calls to the generators can be issued from within eGram. A call to a running generation system consists of an input structure that can be defined within eGram, and the modifications of the grammar since the last call. The generator either returns the generated string or an error message.

Ongoing development concentrates on rendering the interface between eGram and the generation systems supported more comfortable. In particular, processing errors of the generation systems should be interpreted in a useful manner. First steps have been implemented.

In many cases the error usually occurs at different location in the derivation tree than the object causing the error was used. The ultimate goal is to pinpoint the grammar object that most likely caused a derivation to fail. Typical errors include missing feature specifications, the failure of two features to unify, the failure to apply any rule for a given input, and the non-existence of expected input.

6. CONCLUSION

eGram successfully answers the need for a comfortable editor for large sets of context-free grammar rules that optionally can be augmented with feature constraints and with conditions on applicability. It ensures that grammars are syntactically correct,

eGram provides a formalism for metarules, applies metarules recursively according to a specified order, and it checks for, and removes, derived duplicates. These algorithms are to our knowledge completely novel and render metarules manageable in practice. incorporates a practically useful mechanism for metarules and integrates generation functionality for grammar testing. eGram is implemented in Java and can be licensed for research and commercial purposes.

ACKNOWLEDGMENTS

This work has partially been funded by the European Union under contracts no. MLIS-5015 to the project MUSI and no. IST-2000-25045 to the project MEMPHIS. I am indebted to Ana Água, Tim vor der Brück and Matthias Rinck who implemented large parts of the eGram system. I also thank Joachim Sauer, Holger Stenzhorn and Feiyu Xu for fruitful discussions and for their collaboration.

7. References

Stephan Busemann and Helmut Horacek. 1998. A flexible shallow approach to text generation. In Eduard Hovy, editor, *Nineth International Natural Language Generation Workshop. Proceedings*, pages 238–247, Niagara-on-the-Lake, Canada. Also available at `http://xxx.lanl.gov/abs/cs.CL/9812018`.

Stephan Busemann. 1996. Best-first surface realization. In Donia Scott, editor, *Eighth International Natural Language Generation Workshop. Proceedings*, pages 101–110, Herstmonceux, Univ. of Brighton, England. Also available at the Computation and Language Archive at `http://xxx.lanl.gov/abs/cmp-lg/9605010`.

Stephan Busemann. 2002. Language generation for cross-lingual document summarisation. In Huanye Sheng, editor, *International Workshop on Innovative Language Technology and Chinese Information Processing (ILT&CIP-2001), April 6-7, 2001, Shanghai, China*, Beijing, China. Science Press, Chinese Academy of Sciences. Also available at `http://www.dfki.de/lt/publications/index.php3`.

Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. 1985. *Generalized Phrase Structure Grammar*. Basil Blackwell, London.

Alessandro Lenci, Ana Água, Roberto Bartolini, Stephan Busemann, Nicoletta Calzolari, Emmanuel Cartier, Karine Chevreau, and José Coch. 2002. Multilingual summarization by integrating linguistic resources in the MLIS-MUSI project. In *Proc. Third International Conference on Language Resources and Evaluation (LREC)*, pages 1464–1471, Las Palmas, Canary Islands, Spain, May.

Susan W. McRoy, Songsak Channarukul, and Syed S. Ali. 2000. Text realization for dialog. Bangkok, Thailand, December. Also in Working Notes of the 2000 AAAI Fall Symposium on Building Dialogue Systems for Tutorial Applications, North Falmouth, MA, November 2000.

Matthias Rinck. 2003. Ein Metaregelformalismus für TG/2. Master's thesis, Department for Computational Linguistics, University of the Saarland.

Holger Stenzhorn. 2003. XtraGen. A natural language generation system using Java and XML technologies. Master's thesis, Department for Computatinal Linguistics, University of the Saarland.