Looking for Errors:
A Declarative Formalism for Resource-Adaptive Language Checking

Andrew Bredenkamp, Berthold Crysmann, Mirela Petrea

Deutsches Forschungszentrum Künstliche Intelligenz (DFKI) GmbH
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
{andrewb,crysmann,mirela}@dfki.de

Abstract

The paper describes a phenomenon-based approach to grammar checking, which draws on the integration of different shallow NLP technologies, including morphological and POS taggers, as well as probabilistic and rule-based partial parsers. We present a declarative specification formalism for grammar checking and controlled language applications which greatly facilitates the development of checking components.

1. Introduction

This paper describes work aimed at developing a system for writing software components for grammar and controlled language checking. As opposed to the "error recovery" view of parsing, we adopt a phenomena (i.e. error type) oriented approach to language checking. The system is built on top of a range of existing, more or less shallow, natural language processing components, which are used to provide information (linguistic descriptions) as input to the error checking components. The error checking components themselves are expressed using a declarative rule formalism which has been developed to identify phenomena in terms of (but strictly separate from) these linguistic descriptions.

Section 2 gives a more detailed justification of our approach to language checking. We then give a detailed description of the formalism, a formal definition of the rule system which has been implemented, and a description of the procedural semantics given to the rules. Finally, we outline our plans for the future development of these technologies.

2. Phenomenon-based Language Checking

A central problem in the area of grammar checking lies in the discrepancy between the low density of grammatical errors and the high degree of processing needed for reliable error localisation and correction. Thus, checking a text against a positively defined grammar using standard parsing technology to detect "ill-formed" input (Carbonell and Hayes, 1983; Carbonell and Hayes, 1984; K. Jensen et al., 1983; Thurmair, 1990) does not appear to represent an optimal strategy, for a variety of reasons: first, given the scarce distribution of grammar errors in real-world texts, most processing effort goes on the parsing of perfectly well-formed sentences, resulting in an overall slow performance. Also, without an explicit error-model, it will not be straightforward to track down the offending input token(s) within an erroneous sentence. Furthermore, failure during parsing cannot be reliably identified with erroneous input, because it may equally well be related to the coverage of the underlying grammar. Finally, the amount of processing required for the detection and/or correction of a particular error varies considerably according to the error type involved and the linguistic context in which it occurs. Thus, a more flexible and resource-adaptive approach to the task would seem to be attractive.

We have therefore adopted a radically different approach which relies on the development of an empirically derived error-model. On the basis of a detailed error typology for German, as outlined in (Bredenkamp et al., 1999), we have found it useful to formulate a phenomenon-based approach under which an input text is systematically scanned for error candidates using a minimal amount of initial processing. This scanning is achieved by means of "triggers", e.g. token string patterns, POS tags, and morphological annotations. In order to confirm real errors and discard "false alarms", these drastically reduced sets of candidate sentences are further processed using successively elaborate technology (e.g. partial and/or full parsing). Thus, after an initial recall-oriented partitioning into error candidates and non-errors, subsequent processing levels will be concerned with narrowing down the set of error candidates to the set of real error, ensuring a sufficiently high level of precision. As a consequence, both the exact locality of an error and its type will be known quite early on, allowing subsequent processing to be well-focussed.

Rather than relying on a single integrated shallow-processing formalism, e.g. KURD (Carl and Schmidt-

Wigger, 1998) or the Xerox finite state calculus (Karttunen et al., 1996), the phenomenon-based architecture we assume draws a rather strict separation between the linguistic annotations provided by underlying NLP components and the checking components proper. This strategy enables us to combine the virtues of a wide range of existing shallow processing components for the task of grammar checking. Moreover, the use of complementary and competing technologies (e.g. two-level morphological tagging and probabilistic part-of-speech tagging) also permits a cross-validation of underlying NLP components which makes the linguistic annotation of the input text highly reliable. In addition to reliability, the integration of complementary technology guarantees a robust system: if one of the backend NLP technologies does not come up with an appropriate analysis of the input, this information can often be approximated by some other component. Thus overall robustness does not depend on the robustness of each individual module.

Furthermore, the separation of error checking components from the underlying NLP backend technology in an open architecture ensures a high degree of scalability. Currently, different existing shallow processing components have been integrated, including a two-level based morphological tagger (Petitpierre and Russell, 1995), a probabilistic HMM-based POS tagger (Brants, 1996) and chunker (Skut and Brants, 1998), as well as a rule-based topological parser (Braun, 1999) which is part of the SMES information extraction architecture (Neumann et al., 1997). These underlying components can be activated on-demand, following the principle of resource-adaptivity: namely we use only the minimum amount of processing necessary to identify the particular error phenomenon in question.

Although grammar checking currently builds on information provided by shallow NLP components only, the cascaded architecture facilitates a flexible, resource-adaptive approach which will, in work now started, take advantage of annotations provided by deep NLP technology.

3. Specification of Language Checking Components

The use of multiple backend NLP components in the context of language checking necessitates the development of a sufficiently flexible and general specification language which allows the error checking components to access multi-dimensional annotations in a uniform fashion.

A major design goal in the development of the specification language was to ensure that the representation formalism be as declarative as possible, thereby supporting a division of labour between the linguistic task of developing error checking heuristics and engineering aspects, such that individual error checking modules can benefit from global optimisations.

As a second major property, the specification language should, of course, support the basic architecture assumed in the project, namely the fundamental distinction drawn between initial recall-oriented detection of error candidates and successive validation on the basis of more sophisticated information.

Third, although the straightforward procedural interpretation of the basic formalism should be as efficient as possible, the specification language should still provide means to include more complex constraints expressed in terms of "richer" descriptions.

3.1. An error description language

The error description formalism we developed permits the specification of error phenomena in terms of regular expressions over complex linguistic objects, represented as feature structures. These feature structures denote the linguistic annotations provided by different underlying NLP components, such as POS and morphological taggers. The basic regular language has been augmented with a set of relational constraints which allow for the bottom-up integration of (partial) parsing.

Each error description consists of essentially two parts: a declaration of (local) types, and a set of FSAs over the objects so defined (see Figure 1). Rules are classified into trigger rules, which serve to identify an initial set of error candidates, and evidence rules which may confirm a candidate as an error or discard a potential "false alarm".

3.1.1. Type definitions

Error descriptions in the FLAG system mainly consist of type definitions for (word-level) linguistic objects and finite state automata which are stated over the objects so defined. Linguistic objects are defined along various dimensions, including a word's shape, its part-of-speech tag, and morphological information. These dimensions correspond directly to the linguistic annotations provided by the NLP backend components, which are represented in the system as feature structures. Which values a particular feature may assume is thus entirely determined by the corresponding backend technology.

Definition of linguistic types are (partial) descriptions of these feature structures. To warrant a certain degree of generality and compactness of description,

```
#ERROR mWn
#OBJS

@meines ::= [ TOK "^[Mm]eines$" ];
@wissens ::= [ TOK "^Wissens$" ];
@nach ::=
 [TOK "^nach$"
  POS "^(APPR|PTKVZ)$"
 ];
@nach_pos ::=
 [TOK "^nach$"
  POS "^(APPO)$"
 ];



@prep ::= [ POS "^APPR(ART)?$" ];
@vfin ::= [ POS "^V[VAM]FIN$" ];

@em_dat_obj ::=
 [ TOK "em$"
   POS "^(ART|ADJA)$"
   MORPH.READING.INFLECTION.case "dat"
 ];

@er_dat_obj ::=
 [ TOK "er$"
   POS "^(ART|ADJA)$"
   MORPH.READING.INFLECTION.case "dat"
 ];

@noun_fem_dat ::=
 [ POS "^NN$"
   MORPH.READING.INFLECTION [case "dat"
                             number "singular"
                             gender "fem" ]
 ];

#RULES

TRIGGER(50) ==
 @meines^1 @wissens^2 @nach^3
 -> $meines^1, $wissens^2, $nach^3;
TRIGGER(60) ==
 @meines^1 @wissens^2 @nach_post^3
 -> $meines^1, $wissens^2, $nach^3;

POS_EV(40) == $nach @prep^1;
POS_EV(40) == $nach @vfin^1 ;

NEG_EV(40) == $nach @em_dat_obj^1 ;
NEG_EV(40) ==
 $nach @er_dat_obj^1 []* @noun_fem_dat^2
 && cin(^1,"^[PN]P$"^3) && cin(^2,^3);
NEG_EV(10) ==
 $nach @er_dat_obj^1 []* @noun_fem_dat^2;

NEG_EV(20) == @vfin^1 [-{@vfin}]* $meines
 && word($nach,^1);
```

Figure 1: Example of an error checking rule

atomic values in a type definition are interpreted as regular expression over feature values. The syntax of these regular expression is similar to the one used in Perl. In addition, type definitions may also make use of disjunction and negation over paths.

Currently, the system integrates resources provided by the probabilistic POS tagger TnT (Brants, 1996), the morphological analyser MMorph (Petitpierre and Russell, 1995), and lexicalised chunking information, contributed by the probabilistic chunk tagger Chunkie (Skut and Brants, 1998). Of course, further (lexical) resources can easily be made available to the system, such as semantic sort hierarchies, or terminological information.

3.1.2. Regular feature-structure expressions

The specification language distinguishes three basic types of rules: trigger rules, which characterise the initial set of error candidates, positive evidence rules, which serve to map error candidates to confirmed errors, and negative evidence rules, which may eliminate “false alarms”, the latter two collectively known as “validation rules”. All these rules can be assigned relative weights, to signal the reliability of a rule depending on the context or the quality of the underlying information.

Error checking rules consist of a rule header, which specifies the rule type together with a confidence measure, a pattern-matching part (LHS), and an action part (RHS). The LHS of an error rule is a regular expression over complex linguistic objects. Whenever such a regular expression is matched, the position of the objects carrying a coindexation tag (indicated by caret) is saved. These positions can then be assigned to named variables on the RHS (indicated by dollar), in order to make the result of a match available to other rules. Named variables thus constitute the interface between trigger and evidence rules.

3.1.3. Constraints

To permit the integration of linguistic resources beyond finite state machines, the specification formalism introduces a finite number of relational constraints which are linked to the pattern matching by means of coreference tags (indicated by ^). These constraints thus provide the key mechanism for future enhancements to the expressive power of the formalism.

Currently, the system provides constraints for lexical lookup of complex words (e.g. the `word` constraint in Figure 1), as well as dominance and path constraints on the tree structure under which a word-level linguistic object is embedded.

As for tree-structural constraints, the system recognises 3 different types of constraint for every (partial) parser. Among these, the two-place dominance constraint (e.g. `cin`) is certainly the most versatile, because all local tree configurations, like sisterhood, mother-daughter, aunt-nephew can be derived from it. This constraint requires the terminal or non-terminal node specified in the first argument to be dominated by a node matching the regular expression in the second argument. Coindexations not only serve the purpose of linking the structural constraints to the pattern matching, but they can also be used to express token-identity of phrasal nodes. In the example in Figure 1, the tokens matching `@er_dat_obj` and `noun_fem_dat` are required to be contained in the same minimal NP or PP chunk. The `cin` constraint, which always binds its second argument to the first matching node up the tree, is complemented by a three-place constraint `cancestor`, determining the first ancestor of a particular category shared by two nodes in the tree. This permits the specification of less local configurations, including c-command. Which category labels are allowed to hierarchically intervene between any two phrase structure nodes can be further restricted by means of a path constraint (e.g. `cpath`). All these constraints can of course be negated, thus offering additional expressive power.

3.1.4. Rule interaction

As we have already stated above, the crucial mechanism for rule interaction is provided by means of error-local variables.

Whenever a sentence is checked for an error, all trigger rules are applied to it, saving the results of each successful match, i.e. the locality (and probability) of an error candidate, as the values of named variables. For each error candidate, all negative and positive evidence rules will be applied in a subsequent step. If a negative evidence rule matches, its confidence measure will be subtracted from the value assigned by the trigger rule. Similarly, if a positive evidence rule fires, its confidence measure is added to the confidence measure of the appropriate trigger rule.

Evidence rules are used to further constrain the linguistic context of an already identified error candidate (or trigger). Although such contexts may, of course, be specified by means of regular expressions over word-level linguistic objects, this is certainly not the only method. As the locality of an error is already determined by the trigger rule, evidence rules may consist solely of relational constraints.

The separation of error descriptions into trigger rules and evidence rules is mainly motivated by three considerations: efficiency, resource-adaptivity and robustness. Trigger rules should be formulated in such a way as to identify an initial set of error candidates with the minimal amount of linguistic resources. Evidence rules will then operate on a drastically reduced set of candidate sentences, allowing for the efficient use of more expensive machinery, such as partial parsing. The division of labour between recall-oriented trigger rules and more sophisticated, precision-oriented evidence rules also maintains a certain degree of resource adaptivity and robustness, even if some of the backend NLP components are non-robust: if some resource specified in an evidence rule is not available, error status can still be determined on the basis of the trigger rule. Similarly, it is always possible to introduce additional low-confidence evidence rules which build on less sophisticated, but more robust backend technology.

3.1.5. An example

Before we proceed to a formal specification of the error description language, we will briefly discuss the example of a grammar error. The grammar error described by the rules in Figure 1, is a typical, lexically anchored grammar error of German. Speakers often use an erroneous variant of the formulaic expressions *meines Wissens* \`my.GEN knowledge.GEN' or *meinem Wissen nach* \`my.DAT knowledge.DAT according' by blending the two grammatical variants into the ungrammatical **meines Wissens nach* \`my.GEN knowledge.GEN according'. The complexity of this error derives mainly from the fact that the adposition *nach* is highly ambiguous: it can have a dative complement either to its left or to its right, and, it can also be a verbal particle, stranded in sentence-final position. Among these, only the postpositional use gives rise to ungrammaticality.

In order to check this error reliably, we will have to eliminate all those sentences as false alarms, where it is highly probable that *nach* does not form a PP constituent with the preceding genitive NP.

The trigger rules identify occurences of *meines Wissens nach* as potential error candidates and save the position of the respective matches in the three error-local variables. Depending on the POS tag assigned by the probabilistic tagger, i.e. whether *nach* is tagged as a postposition or not, the result of a successful match is associated with a higher or a lower confidence measure.

Positive evidence rules specify some of the linguistic contexts where occurrence of *nach* is most certainly an error: if it is followed by another preposition, or the finite verb, it is most likely we are confronted with the erroneous postpositional use.

The negative evidence rules try to eliminate as false alarms those cases where either *nach* is a preposition with a dative NP to its right, or a stranded verbal particle. Thus, if such an element exists in a sufficiently local context, we take it as negative evidence for error status. As for the identification of datives, not all forms are equally conclusive: while determiners or adjectives ending in *-em* are unambiguous, feminine singular datives (ending in *-er*) are actually identical to genitive singular fenminine and nominative singular masculine forms. Thus, in these cases, we cannot check dative case on the basis of the shape of the determiner alone. Thus, it is further required, that there is a feminine singular dative noun further to the right and that the determiner or adjective and the head-noun are part of the same minimal NP or PP (`cin`). In case the partial parser does not deliver any (useful) results, a weakened version of this rule is included, which does not require locality. Consequently, its confidence measure is considerably low. The last negative evidence rule addresses the separable particle reading of *nach*: if a verb is present in the sentence which may combine with *nach*, we also regard this as evidence in favour of a false alarm. Lexical lookup of combined forms is performed here by means of a builtin constraint (`word`).

3.2. Formal specification

Grammar checking components are built up by means of a rule system, a device to describe different types of grammar errors and a control engine to match rules against annotated input. The system is expressive enough

- to allow the minimum amount of linguistic knowledge to be identified for a particular type of error and the minimal processing needed to acquire it (resource adaptivity),
- to describe an error's triggers (including lexical anchors) and different layers of evidence (positive or negative),
- to allow constraints specification.

As a result of matching an error description, error candidates are signaled out and their locality is reported.

The main components of an error type specification are the fs-patterns (feature structure patterns) and fs-regular expressions (feature structure regular expressions). Informally, a fs-pattern is a feature structure without variables whose features may introduce values of type string-regular expressions (similar to Perl 5 regular expressions). A fs-regular expression is a regular expression over fs-patterns enriched with a special case of back referencing (Aho, 1990).

Definition 1 (fs-pattern syntax) *Let $\mathcal{F}$ be a finite set of feature symbols. A fs-pattern is an expression of one and only one of the following constructs:*

1. *a string-regular expression (atomic pattern);*
2. $\{f_1\ \pi_1, f_2\ \pi_2, \ldots, f_n\ \pi_n\}$*, where f_i belong to $\mathcal{F}$ and π_i are fs-patterns, $1 \leq i \leq n$ (structural fs-pattern);*
3. $\pi_1 \mid \pi_2 \mid \ldots \mid \pi_n$ *(disjunctive fs-pattern).*

A fs-pattern denotes the set of all feature structures which the pattern approximates, according to the definition of fs-matching below:

Definition 2 (fs-matching) *A fs-pattern π matches a feature structure ψ, $\pi \cong \psi$ if:*

1. *π is an atomic fs-pattern, ψ is an atomic feature structure and the exact string pattern matching of π against ψ is successful;*
2. *for every f defined in π, f is also defined in ψ and $val(f,\pi) \cong val(f,\psi)$*
3. *π is a disjunction, $\pi = \pi_1 \mid \pi_2 \mid \ldots \mid \pi_m$ and there is at least a $1 \leq j \leq m$ such that $\pi_j \cong \psi$.*

Definition 3 (syntax of a fs-regular expression) *Let $\mathcal{N}$ be a finite set of names. A fs-regular expression E is one of the following constructs:*

1. π *(fs-pattern);*
2. $E1.E2$ *(concatenation)*
3. $\{E1 \mid E2\}$ *(alternation)*
4. $[E]*$ *(Kleene star),* $[E]+$ *(positive closure),* $[E]?$ *(optional)*
5. $-\{E\}$ *(negation)*
6. $\pi : n$ *(back referencing)*
7. n *(variable)*

where $n \in \mathcal{N}$.

The procedural semantics is based on an adaptation of the Knuth-Morris-Pratt algorithm for exact string pattern matching (Charras and Lecroq, 1997; Aho, 1990) embedded in a backtracking procedure. The main difference is that the test for character equality is replaced by fs-matching. The pattern preprocessing, used for computing the number of positions the pattern must be shifted to the right, makes use of strong fs-pattern subsumption, a relation between fs-patterns defined as follows:

Definition 4 (strong fs-pattern subsumption) *A fs-pattern π subsumes a fs-pattern π', $\pi \subseteq \pi'$ if:*

1. *π is an atomic fs-pattern, π' is an atomic fs-pattern and they are equal with respect to string comparison;*

2. *for every f defined in π, f is also defined is π' and $val(f,\pi) \subseteq val(f,\pi')$;*

3. *π' is a disjunction, $\pi' = \pi'_1 \mid \pi'_2 \mid ... \mid \pi'_m$ and there is at least a $1 \leq j \leq m$ such that $\pi \subseteq \pi'_j$.*

During processing, a fs-regular expression is matched against a sequence of feature structures built on resources provided on demand by distinct NLP components. Whenever a feature structure in the input sequence is matched with a named fs-pattern and the name is *n*, a new variable with name *n* is created and the value assigned to it is the feature structure's position in the sequence. That is, the use of variables in our system is different from the back referencing mechanism in Perl with respect to value assignment, but, as will be shown further, also with respect to the matching of bound variables, their naming and scope.

An error type description consists of a set of objects (macros) and a set of rules. A macro is an atomic fs-pattern enriched with an identifier for fs-patterns which are often used, offering a useful construct which also makes testing fs-pattern subsumption easier (i.e. reduces it to identifier comparison).

The complete specification of a rule consists of a fs-pattern, an optional list of constraints and an action:

$$Pattern\ [Constraints]\ \rightarrow Action$$

where *Pattern* is a fs-regular expression and *Constraints* denotes a conjunction of built-in constraints which must be satisfied in order to have the rule fire. The arguments of a constraint are restricted to variables bound during the matching of *Pattern*. The *Action* part is used to save bound variables in a book-keeping structure for further processing.

An error type may have two types of rules: trigger (or lexical anchor) rules and evidence (or context) rules. The trigger rules define the first layer for "candidacy" checking, that is, if they fire, they set the locality of a hypothesis. The evidence rules check for special conditions over a hypothesis' context and they can be of two types: positive evidence or negative evidence, i.e. they collect information used to respectively confirm or refute, with some degree of confidence, that a candidate is an error.

The control engine proceeds as follows: if at least one of the trigger rules belonging to a description is matched with success, all the context rules are applied. The only means to collect and to transmit information among rules belonging to the same error description, as well as to save the matching results for the decision module, is variable binding.

To ensure that the use of variables inside of an error type description is well-defined, the following restrictions must be fulfilled:

- a back reference may occur only at the upper level of a fs-regular expression, i.e. no subexpression of type closure, alternation or negation is allowed to contain a back reference.

- a rule whose pattern contains a back reference may contain later occurrences of variable's name only as arguments to constraints or in the action side.

- the set of variables saved by the trigger rules must be the same for all the triggers belonging to the same error type description.

- for an error description type, any variable occurrence inside the left hand side of an evidence rule, must belong to the set of variables saved by the trigger rules.

Thus, in addition to the efficient implementation of error checking components, the rule formalism just described supports the grammar writer already in the process of development.

4. Future work

Whilst the implementation of this formalism and its integration in language checking demonstrators has proved that the basic concept is sound, there remain some open questions which are still under investigation.

Firstly, the information available to the error checking component is rather shallow. In principle, however, the error formalism can make use of the results of any depth of processing. We therefore intend to experiment with integrating focussed deep processing of sections of the input, to assess whether this provides any significant benefit in relation to the additional cost in time which it will entail.

Furthermore, we intend to implement some rule systems which are of realistic size for practical applications. This means, for controlled language applications, describing some 40-50 phenomena, and for grammar checking some two hundred would be appropriate. It remains to be seen if the approach outlined here behaves well with rule sets of this size - it will be particularly interesting to see to what extent the rules can usefully interact (e.g. share information) and where this interaction may have negative effects.

5. Conclusion

We have argued that the traditional approach to language checking is best seen, not in terms of error recovery for some parsing system, but rather should focus on real errors. The system described here separates the NLP resources, which provide robust and arbitrarily rich descriptions of input text, and the checking components which use this information in an optimal, focussed fashion, to identify errors. The system has been proved to work well using relatively shallow processing technologies and with small numbers of error types, it remains to be investigated to what extent "deep processing" has a role to play in this model, and how well the approach holds up when a full error typology is implemented.

6. References

Aho, Alfred V., 1990. Algorithms for finding patterns in strings. In Jan van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, volume A. Elsevier Science Publishers.

Brants, Thorsten, 1996. TnT – a statistical part-of-speech tagger. Technical report, Universität des Saarlandes, Computational Linguistics.

Braun, Christian, 1999. *Flaches und robustes Parsen deutscher Satzgefüge*. Master's thesis, Computational Linguistics, University of the Saarland.

Bredenkamp, Andrew, Berthold Crysmann, and Judith Klein, 1999. Annotation of German news corpus. In Anne Abeillé (ed.), *ATALA workshop: Corpus annotés pour la syntaxe/Treebanks*. Université Paris 7.

Carbonell, J. G. and P. J. Hayes, 1983. Recovery strategies for parsing extragrammatical language. *American Journal of Computational Linguistics*, 9(3–4):123–146.

Carbonell, J. G. and P. J. Hayes, 1984. Coping with extragrammaticality. In *Proceedings of COLING 84*. Stanford.

Carl, Michael and Antje Schmidt-Wigger, 1998. Shallow post morphological processing with KURD. In *Proceedings of NeMLaP*. Sydney.

Charras, Christian and Thierry Lecroq, 1997. Exact string matching algorithms. Technical report, Laboratoire d' Informatique de Rouen, Université de Rouen.

K. Jensen et al., 1983. Parse fitting and prose fixing: Getting a hold on illformedness. *American Journal of Computational Linguistics*, 9(3-4):161–177.

Karttunen, Lauri, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller, 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Neumann, Günter, Ralf Backofen, Judith Baur, Markus Becker, and Christian Braun, 1997. An information extraction core system for real world german text processing. In *Proceedings of the 5th Conference on Applied Natural Language Processing (ANLP)*.

Petitpierre, Dominique and Graham Russell, 1995. *MMorph — The MULTEXT Morphology Program*. ISSCO, Geneva.

Skut, Wojciech and Thorsten Brants, 1998. Chunk tagger – statistical recognition of noun phrases. In *Proceedings of the ESSLLI Workshop on Automated Acquisition of Syntax and Parsing*. Saarbrücken, Germany.

Thurmair, G., 1990. Parsing for grammar and style checking. In *Proceedings of COLING-90*. Helsinki.