

Manuskripteinreichung
für die Zeitschrift für Sprachwissenschaft¹

Tim vor der Brück
Fernuniversität Hagen
Universitätsstraße 1
D-58084 Hagen
tim.vorderbrueck@fernuni-hagen.de

Stephan Busemann
DFKI GmbH
Stuhlsatzenhausweg 3
D-66123 Saarbrücken
stephan.busemann@dfki.de

¹We wish to thank our colleagues in the Language Technology departments at DFKI GmbH and the FU Hagen for their support, especially Matthias Rinck, who contributed much to developing eGram, for fruitful discussions. Thanks to Miriam Butt (Univ. Konstanz), who hinted at the relationship to Lexical Functional Grammars, and to Anette Frank (DFKI) for discussions on this topic. Moreover we are grateful for the helpful remarks of two anonymous reviewers. Obviously all errors and misconceptions possibly contained in this paper are within our own responsibility. This work was partially supported by a research grant from the German Federal Ministry of Education, Science, Research and Technology (BMBF) to the DFKI project COLLATE2 (FKZ: 01 IN C02).

Suggesting Error Corrections of Path Expressions and Categories for Tree-Mapping Grammars

Abstract Tree mapping grammars are used in natural language generation (NLG) to map non-linguistic input onto a derivation tree from which the target text can be trivially read off as the terminal yield. Such grammars may consist of a large number of rules. Finding errors is quite tedious and sometimes very time-consuming. Often the generation fails because the relevant input subtree is not specified correctly. This work describes a method to detect and correct wrong assignments of input subtrees to grammar categories by cross-validating grammar rules with the given input structures. The method also detects and corrects the usage of a category in a grammar rule. The result is implemented in a grammar development workbench and accelerates the grammar writer's work considerably. The paper suggests the algorithms can be ported to other areas in which tree mapping is required.

Keywords: *error detection, error correction, grammar workbench, language generation*

1 Introduction

In Natural Language Generation (NLG) we are often confronted with the need to map a tree-like representation of some input onto a derivation tree, from which the target text can be trivially read off as the terminal yield. This generation task is much wider in scope than those that are merely a counterpart to parsing. Two major differences are worth noting:

- The result of parsing is linguistic in nature, as the resulting semantic representation is covered by the linguistic knowledge used, whereas the input to generation is not necessarily linguistic; it may be produced by some non-linguistic application and may, for example, include environmental measurements or time series of stock prices.
- The ways in which a semantic representation may be under-specified in parsing depends on the coverage of the grammar and lexicon, and on what information should be represented at the semantic level, whereas generating a correct target sentence might start from arbitrarily incomplete input. A standard technique involves using canned pieces of text.

Clearly these differences disappear if the generator receives as its input what the parser produces as output. Then the generator's task corresponds to the syntactic realization of semantic representations, for which the same, nondirectional grammar is often proposed (cf. the different approaches in Combinatory Categorical Grammar (White, 2006), Lexical Functional Grammar (Crouch et al., 2004), or Head-Driven Phrase Structure Grammar (Copestake and Flickinger, 2000)).

```
[ ( PRED THRESHOLD-EXCEEDING )
  ( TIME [ ( PRED SEASON ) ( NAME [ ( SEASON WINTER ) ( YEAR 1996 ) ] ) ] )
  ( POLLUTANT SO2 )
  ( EXCEEDS [ ( STATUS NO ) ( TIMES 0 ) ] ) ]
```

Figure 1: Non-Linguistic Input to TG/2, Representing "In winter 1996, the threshold value for sulfur dioxide has not been exceeded."

A grammar-based tree mapping algorithm has been proposed for NLG by Busemann (1996), leading to the systems TG/2 (Busemann, 2005) and XtraGen (Stenzhorn, 2002). While these systems can indeed be used as syntactic realizers, their particular strength lies in solving some "harder" problems of verbalizing non-linguistic input such as in Figure 1 within a single component. The grammar rules include relations between the type of input and the linguistic objects appropriate to verbalizing that input. More concretely, these relations are expressed by associating right-hand side (RHS) elements with path expressions specifying the part of the input structure that is verbalized when the generator processes the RHS element. A rule used to generate from the input in Figure 1 is sketched as follows:

$$\begin{aligned}
 \textit{START} &\rightarrow \textit{TIMEX}[/\textit{TIME}] \\
 &\quad \text{", the threshold value for " } \\
 &\quad \textit{POLL}[/\textit{POLLUTANT}] \\
 &\quad \textit{STATUS}[/\textit{EXCEEDS}]
 \end{aligned}$$

In the above rule, the category *STATUS* is associated with the path expression */EXCEEDS* that specifies the input [(STATUS NO) (TIMES 0)] (cf. Figure 1). For this input, the generator can produce the string "has not been exceeded" using grammar rules with category *STATUS* on their left-hand side (LHS).

Due to their domain dependence, a new grammar must be developed for each application. The coverage of these grammars is tailored towards the intended usage and the ways in which input is expected to be under-specified. The eGram workbench (Busemann, 2004) supports the fast development of such grammars. Sample applications and domains include (for details and further references see (Busemann, 2005)):

- Describing air quality measurements for a pollutant, a time period and a region (in six

languages);

- Verbalizing results of closed-domain question answering (in German and English);
- Describing real estate on sale (in German, English, and Polish);
- Recommending, on a wearable device, places to go and people to meet in a large conference scenario (in English).

New grammars must be developed quickly in order to deliver applications fast and to limit their costs. Tree-mapping grammars may consist of thousands of rules. A first informal look at the nature of the rules shall allow us to understand why debugging is quite tedious and sometimes very time-consuming. The present paper addresses this problem by supporting the grammar writer in correcting erroneous rules.

In the tree mapping formalism proposed by Busemann (1996), grammar rules have a context-free backbone. Each RHS category carries information about which (partial) input tree it can interpret. The system constructs a derivation tree in top-down fashion, thereby visiting and verbalizing the parts of the input tree, as prescribed by the applied grammar rules.

During grammar development the generation process often fails at some stage because a wrong path expression is used, which means that the relevant input subtree is not specified correctly in the grammar rule being processed. The grammar writer must then be aware of what subtree the generation process should have been working on and verify what it actually did work on and which rule was responsible for the failure. When developing NLG grammars for TG/2 or XtraGen using the workbench eGram, it became obvious that up to 60% of the development time was used to correctly specify the mappings of subtrees.

Another relevant error type is the wrong specification of categories in a grammar rule. Consider a version of the above rule, in which instead of *STATUS*, another category of the grammar is used by mistake, e.g. *DATA*. It is quite easily possible to mix up categories, as they relate to domain-semantic entities rather than syntactic constructions.

This paper introduces a static test algorithm that identifies rules which cannot be applied at all, detects wrong assignments of input subtrees to grammar categories and makes suggestions how the respective rules could possibly be corrected. This is achieved by cross-validating RHS elements of grammar rules with the given input structures.

The runtime is proportional to the number of RHS elements in all grammar rules, i.e. it is linear to the size of the grammar. The implementation of the algorithm has been added as a module to eGram, rendering grammar development quicker and more rewarding.

We present two methods to compute a relation between categories and input substructures. The first one uses only grammar rules, while the other uses both grammar rules and a repre-

sentative set of input structures. We may safely assume that such a set of test input structures is always available at grammar development time and that these input structures are correct according to some specification. Often they are produced automatically by some other, non-linguistic system in the course of the generation process. In order to detect incorrect rules, we identify the grammar-derived relations that cannot be supported by relations that also consider the given input structures.

This paper does not discuss NLG methods (for an overview see (Busemann, 2000)). It first overviews, in Section 2, related work on grammar test methods. The formal properties of input tree structures and of grammar rules that form the basis of tree mapping are introduced in Section 3. Section 4 describes how errors in the grammar rules can be detected, whereas Sections 5 and 6 explain the respective correction methods for invalid path expressions and categories. The algorithms can be improved by adding further elements to the grammar formalism, such as rule applicability conditions carrying path expressions. This is discussed in Section 7. Some evaluation results are offered in Section 8. In Section 9, we show that the methods used are not peculiar to NLG as presented here. Their applicability to other fields than NLG is suggested, taking Extensible Stylesheet Language Transformations (XSLT) as our primary example. Section 10 draws conclusions and gives an outlook on further research.

2 Related Work

We are not aware of other work on automatic error location in tree mapping grammars. However Zeller (2005) describes a dynamic test algorithm for computer programming languages—called “Delta-Debugging”—that exactly determines the causes for a failure. This algorithm isolates the error by subsequently executing different parts of the computer program with varying program states. Errors found by this algorithm are either program crashes or cases in which the actual outcome does not match with the intended outcome. The algorithm is able to find either the exact location in the program code or an illegal variable assignment which caused the error.

Other kinds of dynamic approaches execute some specified set of test cases and compare the results with the desired outcome. A dynamic test system for natural language analysis of this kind is described by Lehmann et al. (1996).

In contrast, the algorithm described here is a static grammar test algorithm (see Daich et al., 1994 and Spillner and Linz, 2003), which does not rely on executing the underlying system.

Static tests for programming languages frequently use various patterns for potential errors that include unused variables, unreachable code segments etc. More advanced techniques try to represent the program code in complex logical representations. One common problem tackled is related to type checking of variables, which is often the cause for unpredictable behavior of

$$\left[\begin{array}{l} \begin{array}{l} \text{arg1} \left[\begin{array}{ll} \text{det} & \text{def} \\ \text{head} & \text{"man"} \\ \text{num} & \text{sg} \end{array} \right] \\ \text{pred} \text{"look - for"} \end{array} \\ \begin{array}{l} \text{arg2} \left[\begin{array}{ll} \text{det} & \text{def} \\ \text{head} & \text{"dog"} \\ \text{num} & \text{pl} \end{array} \right] \end{array} \end{array} \right]$$

Figure 2: A Sample NLG Input Tree as a Feature Structure. A possible output could be "The man looks for the dogs." The derivation is shown in Figure 3.

computer programs (Palsberg and Schwartzbach, 1991).

3 Formal Background

In the present context, an NLG input structure is an unordered tree that is represented as a feature structure, which is a set of attribute value pairs. Attributes are symbols. Values are either symbols (or strings) or feature structures. A sample input structure is given in Figure 2, using standard matrix notation.

With a set of context-free grammar rules, in which each non-terminal RHS category is assigned a substructure of the current input, a derivation tree can be generated. In Figure 3 nodes are labeled by pairs of grammar categories and input structures, while links are labeled by a path expression that specifies the input substructure relevant for expansions of the respective RHS category.¹ The empty path expression '/' leaves the current input unchanged.

This section provides some formal underpinnings. Let *first*, *last* and *rest* be functions over sequences that return the first, the last or all elements except the first, respectively. Let *A* be a set of attributes and *F* the set of all feature structures. Then a function $sel : F \times A \rightarrow F$ can be defined to extract the value of an attribute from a feature structure:

$$sel([(a_1 v_1) \dots (a_n v_n)], a_i) = v_i$$

If $a_i \notin \{a_1, \dots, a_n\}$ then v_i is the empty feature structure, denoted by $[]$.

¹Obviously this is a very simple example used for expository purposes. Real world input may require quite complex mappings onto linguistic levels.

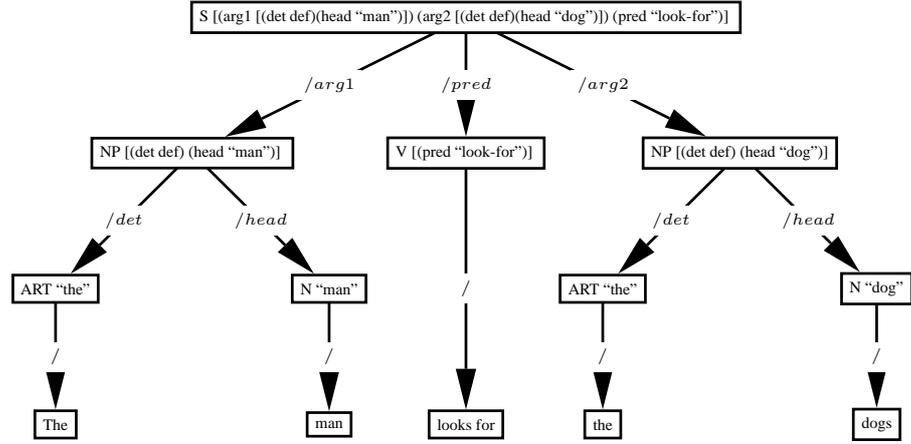


Figure 3: Derivation Tree Generated Using the Input From Figure 2.

The function sel can be recursively extended to include a sequence of attribute names, called a path expression. This will allow us to access part of an input feature structure that is located at the end of the path expression. We define $psel : F \times A^* \rightarrow F$ as follows.

$$psel(s, p) = \begin{cases} s, & \text{if } p = / \\ [], & \text{if } s = [] \\ psel(sel(s, first(p)), rest(p)), & \\ \text{otherwise} & \end{cases}$$

Instead of writing $psel(s, p)$ we also use the infix notation $p \bullet s$.

An attribute-value pair (a, v) is defined as being *contained* in a feature structure s ($(a, v) \in^R s$) if there exists some path expression $p \in A^*$ with $p \bullet s = v$, $last(p) = a$.

Let us now specify in more detail the context-free grammar rules used for tree mapping. Any RHS element is either a terminal symbol (a character string) or a non-terminal element $A[p]$ including a path expression p . A is either a category symbol or a string-valued function over some input structure, giving rise to a terminal element of the derivation tree. String-valued functions are often used to transfer material from the input structure directly to the output string, e.g. names or numbers. If A is a category, p defines a part of the input structure that must be accessed by the rule selected by the generation component to further expand that RHS category in the derivation tree.

Consider some node n in a derivation tree with category C . Let $[a_1 / \dots / a_m]$ be the path

expressions assigned to each RHS element in the course of the derivation from the root node to node n . Then a rule applied to category C can access the feature structure s contained in the input structure according to $a_m \bullet \dots \bullet a_1 \bullet s$. This behavior is illustrated in the sample derivation tree in Figure 3. The nodes are labeled with pairs (C, s) of the category name and the associated part of the input structure.

Furthermore, a grammar rule $R : C \rightarrow A_1[p_1], \dots, A_n[p_n]$ ² can only be applied to a pair (C, s) of category and input structure if none of the path expressions leads to the empty feature structure: $\forall i \in \{1, \dots, n\} : p_i \bullet s \neq []$.

4 Detection of Errors

For the automatic correction, we will compare the attributes specified by path expressions with those that may occur in some input structure. Since path expressions are associated with RHS elements, the algorithm will be centered around grammar categories in order to synchronize the ways in which the grammar is interpreted and the input structure is accessed.

Note that currently path components not located directly at the beginning or the end of the path expression are ignored. Thus results involving path expressions of a length > 2 may become inexact. However, such path expressions hardly occur in practice, and we decided to leave it to future work to cover such cases as well.

We use the following grammar rules—which contain an error—to illustrate the algorithm:³

$$\begin{aligned} R_1 : START &\rightarrow \text{ "from " } TIME[/from] \text{ "to " } TIME[/to] \\ R_2 : TIME &\rightarrow toString[/hour] toString[/from] \end{aligned}$$

We take the input structure in Figure 4 as given and note that the grammar developer should have written $/min$ instead of $/from$ in rule R_2 . This error should be corrected by our algorithm.

4.1 Determining left and right attributes of a category

For the automatic correction we investigate the top-level attributes of the kind of input structure that is associated to a category. We call the attributes of these input structures *right attributes*

²We ignore terminal elements as they do not carry path expressions and are thus not subject to the error correction procedure.

³To save space, the path expressions are included into the rules.

$$\left[\begin{array}{l} \textit{from} \\ \textit{to} \end{array} \left[\begin{array}{ll} \textit{hour} & \textit{"12"} \\ \textit{min} & \textit{"20"} \end{array} \right] \right]$$

Figure 4: Input Structure Guiding Further Examples

of that category. Similarly we call the set of attributes leading to an input structure related to a RHS category *left attributes* of that category.

As mentioned in the introduction, a grammar-based method will be introduced and validated by a method based on both the grammar and the input structures. Thus we define the left and right attributes first as grammar and then as validation attributes.

Consider all rules with LHS C that contain one or several RHS elements with path expressions. The right attributes of a category C , derived from the grammar, are defined as the set of the first components of these path expressions. They are called *right grammar attributes* of a category. If the path expression of a RHS category is empty, additionally the right grammar attributes of that category are also considered as right grammar attributes for C .

Formally the right grammar attributes of a category are defined as follows:

$$\begin{aligned} \textit{attr}_{r,g}(C) = & \{a | \exists R \in \textit{Rules} : R : C \rightarrow A_1[p_1] \dots A_n[p_n] \wedge \\ & (\textit{first}(p_i) = a \vee p_i = / \wedge A_i \in \textit{Categories} \wedge a \in \textit{attr}_{r,g}(A_i)) \wedge \\ & 1 \leq i \leq n\} \end{aligned}$$

In the derivation tree (see Figure 3) the expressions attached to the edges that are leaving a category are a subset of the right grammar attributes of this category.

Now consider RHS elements with a category A_i , which are associated with path expressions. The left attributes of a category A_i , derived from the grammar are defined as the last elements of these path expressions. Those attributes are called *left grammar attributes* of a category; they are formally defined as follows:

$$\begin{aligned} \textit{attr}_{l,g}(A_i) = & \{a | \exists R \in \textit{Rules} : R : C \rightarrow A_1[p_1] \dots A_n[p_n] \wedge \\ & (\textit{last}(p_i) = a \vee p_i = / \wedge A_i \in \textit{Categories} \wedge a \in \textit{attr}_{l,g}(C)) \wedge \end{aligned}$$

category	$attr_{r,g}$	$attr_{l,g}$
START	{from, to}	\emptyset
TIME	{hour, from}	{from, to}

Table 1: The Right and Left Grammar Attributes of (Erroneous) Sample Grammar.

$$1 \leq i \leq n\}$$

In the derivation tree, the left grammar attributes of a category contain all last path components of the path expressions attached to the edges that are leading to that category. In our (erroneous) sample grammar the right and left grammar attributes can be determined according to Table 1.

Let us now turn towards the definition of validation attributes. To derive the attributes of a category from both grammar and input structures, we need a single representation of all available input feature structures.

Let Inp be the set of all input feature structures available for the given grammar. We define a function $children$ to denote the set of top-level attributes that may occur in a given attribute's feature value: $children : A \rightarrow 2^A$

$$b \in children(a) \Leftrightarrow \exists s \in Inp, f \in R : sel(f, a) = [...(b, v) ...]$$

We further introduce an additional attribute name top which has as its children all attributes that do not have a parent. We thus have

$$children(top) := \{a \mid \exists s \in Inp \wedge (a, b) \in R \wedge \nexists c : a \in children(c)\}$$

b is called a child of a (and a is called the parent of b) if $b \in children(a)$. Instead of referring to the input structures directly we use the function $children$ to introduce the facts about input structures into the checking procedure. In our sample input we have e.g. $hour \in children(from)$.

As a side note, we do not rely on NLG input being typed in the sense of typed feature structures (cf. e.g. Krieger and Schäfer, 1994), which would make life simpler and more elegant by allowing us to abstract away from $children$ and use type definitions instead. However, in practice input to NLG systems originates from non-linguistic sources and is usually untyped.

We now describe the attributes associated with some category A_i (right attributes) and their parent attributes (left attributes). Let R be a grammar rule containing a RHS element A_i and

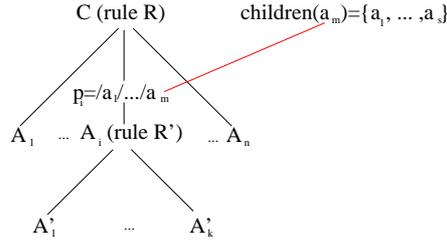


Figure 5: Right Validation Attributes: Retrieving the children of a_m .

R' a rule that expands A_i (cf. Figure 5). Using the last element a_m of the (non-empty) path expression p_i , $children(a_m)$ determines a superset of the top level attributes of the kind of input structure s the rule R' operates on. If p_i is the empty path expression, s is identical to the input structure the rule R is associated with.

For a given category A_i and for all rules with A_i as a RHS element, we build the union of all supersets of top-level attributes as described above. We call this set the *right validation attributes* of A_i .

Formally the right validation attributes of a category are defined as follows:

$$\begin{aligned}
 attr_{r,v}(A_i) = & \{a | R \in Rules : R : C \rightarrow A_1[p_1] \dots A_n[p_n] \wedge \\
 & (a \in children(last(p_i)) \vee p_i = / \wedge a \in attr_{r,v}(C)) \wedge \\
 & 1 \leq i \leq n\}
 \end{aligned}$$

Note that the set of right validation attributes of the top-most category in a derivation tree, $START$, contains all attributes without parents: $attr_{r,v}(START) \supset children(top)$. It might contain more than these if recursive rules are used in the grammar.

To elucidate the relation between grammar and validation attributes in a derivation tree, let us consider a pair of a category C and some input structure (cf. Figure 3), as well as the underlying rule R with LHS category C . Note that the top level attributes of that input structure should always be a subset of the right validation attributes of C . The right grammar attributes of C derived directly from R must appear in the right validation attributes of C . Otherwise R can never be applied, and the RHS element expanded by C is a potential error candidate.

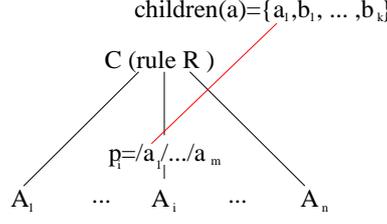


Figure 6: Left Validation Attributes: Retrieving the parents of a_1 .

We now define left validation attributes in a similar way. Consider a rule

$$R : C \rightarrow A_1[p_1] \dots A_n[p_n]$$

with $p_i = /a_1/\dots/a_m$ (cf. Figure 6), where $p_i \neq /$. The top-level attributes of the input structures rule R operates on are a subset of all parents a of a_1 ($a_1 \in \text{children}(a)$). For a given category C and for all rules with C as their LHS category, we build the union of all attributes a that are parents of a_1 , as described above. We call this set the *left validation attributes* of C .

Formally the left validation attributes of a category are defined as follows:

$$\begin{aligned} \text{attr}_{l,v}(C) = & \{a \mid \exists R \in \text{Rules} : R : C \rightarrow A_1[p_1] \dots A_n[p_n] \wedge \\ & (\text{first}(p_i) \in \text{children}(a) \vee p_i = / \wedge a \in \text{attr}_{l,v}(A_i)) \wedge \\ & 1 \leq i \leq n\} \end{aligned}$$

Note that in the (normal) case no recursion on the category $START$ is used; there is no left validation attribute of this category: $\text{attr}_{l,v}(START) = \emptyset$.

In our sample grammar the right and left validation attributes can be determined as in Table 2.

category	$\text{attr}_{r,v}$	$\text{attr}_{l,v}$
START	{from, to}	\emptyset
TIME	{hour, min}	{from, to}

Table 2: The Right and Left Validation Attributes of (Erroneous) Sample Grammar.

category	$attr_{r,err}$	$attr_{l,err}$
START	\emptyset	\emptyset
TIME	{from}	\emptyset

Table 3: Identifying the Incorrect Attribute for the Sample Grammar.

4.2 Identifying incorrect path expression and category occurrences

Basically a path expression in some RHS element is considered incorrect if a grammar attribute of some category was derived but could not be verified by some validation attribute of that category.

However, there is one exception to this basic rule. Consider the case that both the set of right validation attributes and the set of right grammar attributes of some category are empty. Without right grammar attributes no left validation attributes can be derived for this category, and hence the left grammar attributes for this category cannot be checked by any validation attributes.

The sets of possibly incorrect grammar attributes for some category C can be defined as follows:

$$\begin{aligned}
 attr_{l,err}(C) &:= \begin{cases} \emptyset, & \text{if } attr_{r,g}(C) = \emptyset \\ attr_{l,g}(C) \setminus attr_{l,v}(C), & \text{otherwise} \end{cases} \\
 attr_{r,err}(C) &:= attr_{r,g}(C) \setminus attr_{r,v}(C)
 \end{aligned}$$

In order to identify an incorrect RHS element, each grammar attribute is assigned to the RHS elements it was derived from.

With our sample grammar, this algorithm would evaluate to the attribute *from* of category *TIME* being incorrect, as shown in Table 3.

Usually this method identifies the actual error location. However, if empty path expressions are used in a sequence of rule applications, an error can be located at any rule in such a sequence. We currently use a heuristic to solve such ambiguities, which is described in Section 5.2.

5 Corrections for Invalid Path Expressions

This section describes how the information about a possibly incorrect path expression can be used to correct most grammar errors pertaining to path expressions and categories automatically. The correction information should contain the following information:

- incorrect rule;
- incorrect RHS element of that rule;
- wrong path expression appearing in that element;
- possible correct path expressions.

A correct path expression must fulfill the following conditions:

- The first element must be contained in the right validation attributes of the LHS category of the rule containing the incorrect RHS element.
- The last element must be contained in the left validation attributes of the incorrect RHS element.

Let P be the set of path expressions and $lhs(A_i)$ be the LHS category of the rule with RHS element A_i . The set P_c of possible correct path expressions can formally be described as follows:

$$P_c(A_i) = \{p \in P : first(p) \in attr_{r,v}(lhs(A_i))\} \\ \cap \{p \in P : last(p) \in attr_{l,v}(A_i)\}$$

Note that theoretically an infinite number of path expressions can exist if recursion is used. However, in the grammar formalism used recursion is represented within the rules rather than within path expressions. Thus we may assume a finite set of path expressions used in grammar rules. In eGram, path expressions are defined independently of grammar rules using path variables, which in turn are used to define a grammar rule. Our finite set of path expressions is thus the union of all values of path variables defined in the grammar. To find the correct path expression for some erroneous action, we only have to iterate over this set.

Recall that a terminal RHS element (a string-valued function) is not assigned to any category. In this case we just have

$$P_c(A_i) = \{p \in P : first(p) \in attr_{r,v}(lhs(A_i))\}$$

The algorithm stated above will never find the empty path expression as a potential correction. Thus this case has to be handled in a special way. Let us consider first if the empty path expression can be used for as a correction for a *non-terminal* RHS element A_i . Obviously, in order for the modified grammar to be recognized by our algorithm as correct, the following conditions have to be fulfilled:

$$C_1 : attr_{r,g}(A_i) \subset attr_{r,v}(A_i)$$

$$\begin{aligned}
C_2 &: attr_{l,g}(A_i) \subset attr_{l,v}(A_i) \\
C_3 &: attr_{r,g}(lhs(A_i)) \subset attr_{r,v}(lhs(A_i)) \\
C_4 &: attr_{l,g}(lhs(A_i)) \subset attr_{l,v}(lhs(A_i))
\end{aligned}$$

Note that the modified RHS element A_i can only influence the part of the right grammar attributes of $lhs(A_i)$ that is derived from A_i . Since A_i is associated with the empty path expression, this part corresponds to $attr_{r,g}(A_i)$ (cf. Section 4.1). Thus, condition C_3 can be changed to

$$C'_3 : attr_{r,g}(A_i) \subset attr_{r,v}(lhs(A_i))$$

Analogously condition C_2 can be changed to

$$C'_2 : attr_{l,g}(lhs(A_i)) \subset attr_{l,v}(A_i)$$

The fulfillment of conditions C'_2 and C'_3 actually implies that conditions C_1 and C_4 hold as well. This follows from the definition of validation attributes. From the definition of the right validation attributes we know that

$$attr_{r,v}(A_i) \supset attr_{r,v}(lhs(A_i))$$

With C'_3 , condition C_1 follows. Using the definition of the left validation attributes, the following assertion can be made:

$$attr_{l,v}(A_i) \subset attr_{l,v}(lhs(A_i))$$

With C'_2 , condition C_4 follows.

Thus, to decide whether or not A_i can be assigned the empty path expression, we only have to verify conditions C'_2 and C'_3 . Furthermore, the sets compared in both conditions are not affected by modifications of A_i , which means that they need not to be recalculated.

For a terminal RHS element, we suggest the empty path expression if the right validation attributes of this element are empty. In this case a leaf node in the input structure must have been reached and the empty path expression is the only possible solution.

In our sample grammar, the set of possible correct path attributes is evaluated to

$$attr_{r,v}(TIME) = \{hour, min\}$$

Therefore the path expression */from* occurring in rule R_2 has to be replaced by either */hour* or */min*. The next section explains the choice among alternative corrections.

5.1 Choosing among alternative corrections

The candidate set of possible corrections as described in the last section may contain multiple elements as a unique solution cannot always be found. In this case several heuristics may be applied to rule out unwanted candidates. We used the following heuristics:

Avoiding Double Generation: The same path expression should not occur twice in connection with the same category in a single rule, as this would result in repeating a piece of text in the generated output. Such candidates are thus not suggested for correction.

Analogously we define a heuristic for path expressions appearing in terminal RHS elements of the same rule. Applying this heuristic to our example yields the unique solution */min*, which is actually correct.

Preference of Frequent Candidates: This heuristic assumes that most of the grammar rules are correct. To disambiguate the candidate set we count the occurrences of path expressions per rule context and use this information to derive a rating for each of the path expressions in the candidate set. A path expression is preferred if its rating related to the RHS element in question significantly exceeds the average value, which is determined by a statistical test.

5.2 Identifying the exact error location

As mentioned in Section 4.2, the actual error is—in the case when empty path expressions are used—not always located at the action indicated by the difference set of right/left validation and grammar attributes. We demonstrate this in more detail using a slightly modified sample grammar:

$$\begin{aligned} R_1 : START &\rightarrow \text{"at" } TIME[/time] \text{" at" } LOC[/loc] \\ R_2 : TIME &\rightarrow toString(/hour) \text{" :"} toString(/min) \text{" h" } \\ R_3 : LOC &\rightarrow toString(/street) \text{" in" } toString(/city) \end{aligned}$$

Generating from the input structure in Figure 7 using the above rules could give rise to a phrase like “at 12:20h at Poststr. 32 in Bonn”.

Assume, however, that the grammar developer erroneously defined the rule $R_{1,err}$, which differs from R_1 in the path expression assigned to $TIME$:

$$R_{1,err} : START \rightarrow \text{"at" } TIME[/] \text{" at" } LOC[/loc]$$

$$\left[\begin{array}{l} time \\ loc \end{array} \left[\begin{array}{ll} hour & "12" \\ min & "20" \end{array} \right] \right. \\ \left. \left[\begin{array}{ll} city & "Bonn" \\ street & "Poststr. 32" \end{array} \right] \right]$$

Figure 7: Input Structure for Modified Sample Grammar.

Now the attributes *hour* and *min* can no longer be verified by the right validation attributes of the category *TIME* and are indicated as error candidates. Actually there are other possibilities to modify the grammar in a way that it would be recognized as correct by our algorithm, as discussed so far. One of them is to change two path expressions in rule R_2 , yielding $R_{2,mod}$:

$$R_{2,mod} : TIME \rightarrow toString(/loc/city) " : " toString(/loc/street) " : "$$

In this case, the right grammar attributes of the category *TIME* are changed to the set {time} which is actually covered by the right validation attributes.

To account for such errors, we use the heuristics to change as few path expressions as possible, assuming that errors usually are a result of local mistakes. Changing R_2 to $R_{2,mod}$ needs two changes, whereas changing $R_{1,err}$ to R_1 needs only one change. Hence the latter correction is preferred.

In our sample grammar, $R_{1,err}$ will be correctly changed into R_1 since the category *TIME* is assigned a set of two wrong features, namely {*hour*, *min*}.

5.3 Interdependencies of errors

An incorrect RHS element may result into deriving incorrect right validation attributes for other RHS elements of that rule as well as deriving incorrect left validation attributes at the LHS category of that rule. Therefore some errors may not be found, or multiple corrections are suggested.

Since the right validation attributes of the category *START* are always⁴ correct, the algorithm determines the errors in the right grammar attributes of that category correctly. If errors are found, the associated RHS elements are excluded from determining left validation attributes

⁴Assuming that *START* never occurs on the RHS of a grammar rule, which is usually the case.

of the child categories of *START*, thus maintaining a correct set of attributes for further processing. However, some right grammar attributes of a child category may no longer be covered by associated right validation attributes, and therefore, new errors can eventually be found in these right attributes. This in turn can prevent determining incorrect right validation attributes of grandchildren etc.

To detect all such errors the categories are ordered top-down according to their appearance in the derivation tree and processed in this order.

6 Corrections of Invalid Categories

Sometimes the algorithm specified so far indicates a grammar error although the grammar developer specified the correct path variable but used instead a wrong category. An incorrectly specified category may result in the generation system failing, or producing ungrammatical sentences.

In this section, we demonstrate how our algorithm can be extended to the correction of categories. Note however that it is not able to detect errors of the kind where two categories are specified in the wrong order, e.g., $NP \rightarrow N ART$, which could result in the generation of a sentence like “woman the goes into town.” Such errors are usually easily traced down to their source.

Consider a rule R with a wrong LHS side category C_{err} . For a correct category C we require that the right grammar attributes of C_{err} that are derived from R be a subset of the right validation attributes of C : $attr_{r,g,R}(C_{err}) \subset attr_{r,v}(C)$.

The correction of incorrect RHS side categories works in an analogous way. Let R be a rule with a wrong RHS category C_{err} . For a correct category C the left grammar attributes that are derived from R must be contained in the left validation attributes of the correct category C : $attr_{l,g,R}(C_{err}) \subset attr_{l,v}(C)$.

To illustrate this algorithm, we use the grammar and input structure from Section 5.2. Consider the case that the grammar developer accidentally used the category *LOC* instead of *TIME* on the LHS of the rule R_2 :

$$R_{2,err} : LOC \rightarrow toString(/hour) " : " toString(/min) "h"$$

As Table 4 shows, the difference set subtracting right grammar from right validation attributes is not empty for the category *LOC*. Therefore we can assume that the LHS category of rule $R_{2,err}$ is not correctly specified. The right validation attributes of a possible LHS replacement category must contain the right grammar attributes of *LOC* derived by the rule $R_{2,err}$,

	START	TIME	LOC
Right grammar attributes	time, loc		hour, min, city, street
Left grammar attributes		time	loc
Right validation attributes	time, loc	hour, min	city, street
Left validation attributes			time, loc
Difference right			hour, min
Difference left		time	

Table 4: Attribute Sets for the Categories of the Sample Grammar Consisting of the Rules R_1 , $R_{2,err}$, and R_3

which are the attributes *hour* and *min*. The only category of the grammar in Table 4 fulfilling this condition is the category *TIME*. Therefore *TIME* is proposed as the replacement for the erroneous category *LOC* in rule $R_{2,err}$, which is indeed correct.

An alternative solution is triggered by the fact that the left difference set of *TIME* is not empty as well. This suggests that the RHS category *TIME* in R_1 can be considered erroneous—it is the only occurrence in the grammar—and be replaced by *LOC*. While using the same category to specify time and location information is awkward from a mnemotechnical point of view, it just works perfectly.

It is up to the grammar writer to decide which correction she prefers.

7 Extending the Grammar Formalism: Introduction of Tests

For text generation, introducing applicability conditions into the rules may increase efficiency or prevent unwanted results. The formalism used in TG/2 and XtraGen allows for the definition of Boolean predicates in propositional logic. Among the more frequently used predicates are

- *equal* : returns true iff both arguments are identical.
- *featureExists* : returns true iff the given path expression applied to the current input structure does not return the empty feature structure.

The arguments of a predicate can be either constants or path expressions. In the latter case, the predicate is evaluated on the input substructure referred to by the given path expression.

We observe that path expressions appearing in a predicate can be subjected to our correction algorithm, too. Basically test predicates in a rule can be treated in the same manner as terminal RHS elements.

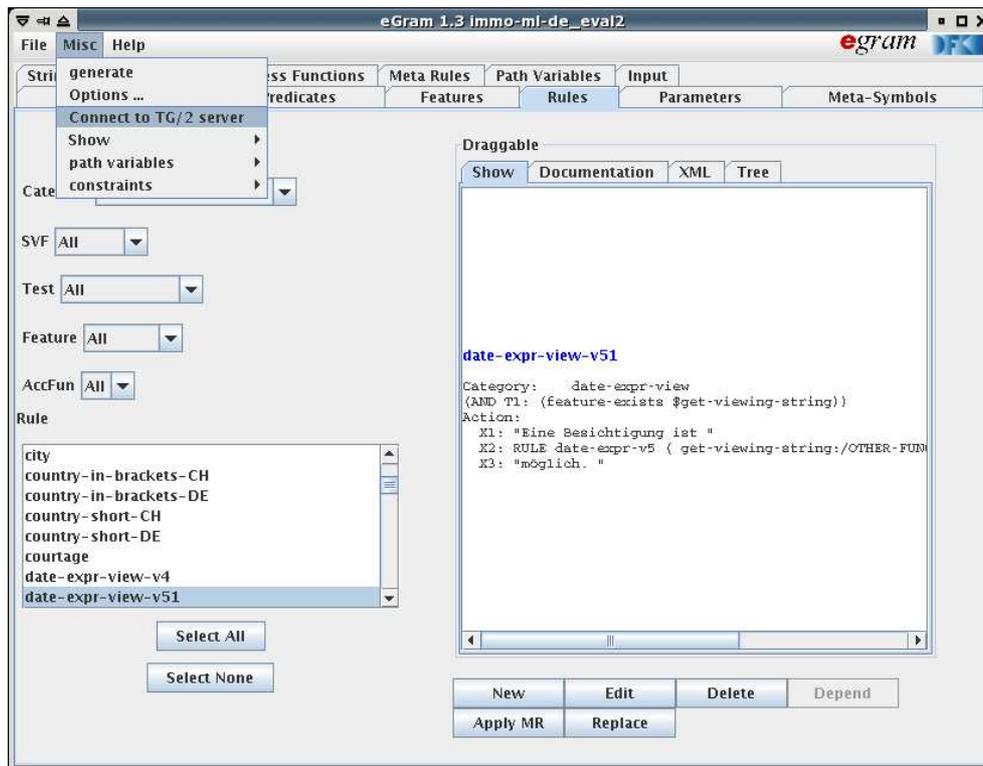


Figure 8: A Screenshot of eGram with the Rules Pane Displaying a Rule.

We use tests to resolve ambiguities in the candidate set of possibly correct path expressions. Consider an erroneous rule with a test that contains the non-negated test predicate *featureExists* and some path expression as its argument. We prefer attributes in the candidate set that are prefixes of such path expressions over those that are not, as it is more likely that *featureExists* asks for a substructure that is afterwards accessed in the course of the derivation. Of course this assumes that the argument to the test predicate is correct. The interaction of errors, in particular, of errors in test predicates with those in RHS elements, is left to future work.

8 Implementation and Evaluation

This work has been implemented as a Java plugin to the editor eGram (Busemann, 2004). eGram is a development environment for grammars and input structures, as they are used by the NLG systems TG/2 (Busemann, 2005) and XtraGen (Stenzhorn, 2002). The GUI of eGram is illustrated in Figure 8.

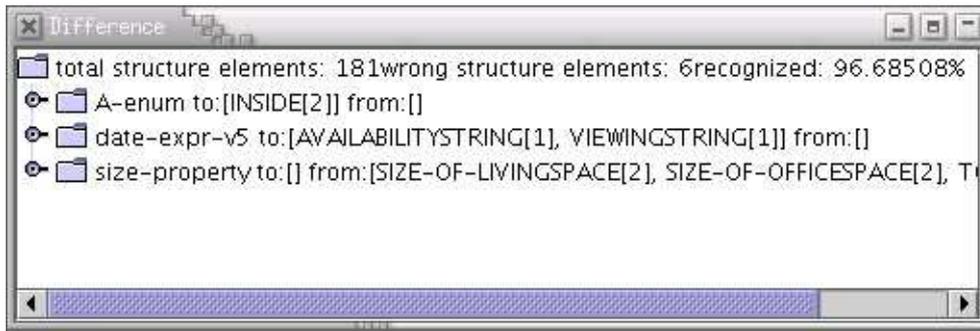


Figure 9: Difference Set

rule name	action name	old path variable	new path variables
outside-listing2	X13: OPTRULE street-faci...	get-street-facing	[get-balcony, get-bathrooms, g...
storeys-on-deal	X1: RULE rent-or-sale-n (...	get-real-type	[get-address-country, get-avail...
outside-listing2	X12: OPTRULE rear-facing...	get-rear-facing	[get-balcony, get-bathrooms, g...
other-stmts	X9: OPTRULE owner (get-...	get-owner	[get-address-country, get-avail...
outside-listing2	X10: OPTRULE lorry (get-l...	get-lorrys-in	[get-balcony, get-bathrooms, g...
sizes-business-ho...	X11: OPTRULE land-reg-d...	get-land-reg-desc	[get-address-country, get-avail...
sizes-business-ho...	X9: OPTRULE land-reg (g...	get-land-reg	[get-address-country, get-avail...
inside-props-floor...	X2: OPTRULE A-enum (ge...	get-inside	[get-address-country, get-avail...
inside-props-floor...	X4: RULE A-enum (get-in...	get-inside	[get-address-country, get-avail...
availability-v5 1	X1: RULE date-expr-v5 (...	get-availability-string	[get-availability-date]
date-expr-view-v5 2	X2: RULE date-expr-v5 (...	get-viewing-date	[get-availability-date]
date-expr-view-v5 1	X2: RULE date-expr-v5 (...	get-viewing-string	[get-availability-date]

Figure 10: The Set of Corrections Suggested After Checking the Grammar.

The plugin offers menu items for displaying the set differences between validation and grammar attributes (see Figure 9) as well as the suggested corrections (see Figure 10). The right and left grammar and validation attributes together with the RHS elements they are derived from can be displayed as well. Suggested corrections are entered by the grammar writer from within eGram.

The algorithm was evaluated on a grammar with 270 rules and 111 input structures. The grammar was assumed to be correct. We randomly changed a path expression (a category) of that grammar and investigated in 200 trials if our algorithm was able to detect the grammar modification. The runs took 22 seconds on an AMD Athlon XP 2200+ with 1 GB of main memory.

We evaluated precision and recall for the path expression/category error detection and correction. For our evaluation detection recall is the percentage of cases, in which the error could actually be found, and correction recall the percentage of cases in which an error was corrected.

Criterion	Path Expressions	Criterion	Categories
Recall (1 error)		Recall (1 error)	
Detection	90,0%	Detection	69,5%
Correction	76,0%	Correction	80,0%
Precision (1 error)		Precision (1 error)	
Detection	62,5%	Detection	52,3%
Correction	55,0%	Correction	73,4%

Table 5: Recall and Precision for the Error Detection and Correction of both Path Expressions and Categories. Category correction values are based on the smaller grammar.

Detection precision, on the other hand, is the percentage of all proposed error detections that were actually correct, and correction precision the percentage of all proposed error corrections that were actually correct. Detecting a path expression error is considered successful if the incorrect RHS element as well as the wrong path expression(s) could be determined. Regarding categories, an error detection is considered successful if the mismatching LHS could be found. For the calculation of recall and precision of the correction part, we only considered suggestions relating to the actual error. Hence the results of the error detection do not influence the score of the correction algorithm.⁵ The evaluation results are given in Table 5.

We leave it to future work to design heuristics for the disambiguation of categories. If a grammar contains a large amount of empty path expressions, as does our evaluation grammar, many unwanted candidates are suggested, yielding bad precision values.⁶ To exclude this factor we evaluated category correction on a smaller grammar that contains only a few empty path expressions. The recognition part was, however, still evaluated on our original grammar.

Table 6 shows the considerable influence the various heuristics have on the precision gain of the path expression correction algorithm. We dropped one heuristic after the other and evaluated the precision and recall. While a small loss in recall is an expected price to pay for the gain in precision, we are unsure to which extent the strong impact of the heuristics depends on the way the evaluation grammar is designed. Systematic evaluation with many and different grammars will help clarify this.

First investigations of cases in which the algorithm did not work correctly revealed several possible reasons.

⁵In our evaluation there is only one correct solution, namely the inverse operation to our modification, even though there are many grammars that generate the correct output.

⁶First experiments with statistical disambiguation methods let us expect significant improvements of precision values.

Dropped Heuristics	precision	recall
Preference of Frequent Candidates (Section 5.1)	33.0%	77.5 %
Feature-Exists (Section 7)	26.2%	78.5%
Avoiding Double Generation (Section 5.1)	17.5 %	82.0%

Table 6: Effects of Dropping Heuristics-Based Disambiguation on the Correction of Path Expressions.

- Multiple suggestions and missed cases may arise if a transition in the grammar from one category to another can occur in connection with several different path expressions.
- Wrong path expressions at terminal elements may yield multiple suggestions since the related paths cannot be checked using left validation attributes (cf. our guiding example).
- If an attribute has different sets of children in the input structures (*from* and *to* could e.g. also be used for local descriptions), additional spurious suggestions may be generated.
- If a category is just used in very few grammar rules, the usage of a wrong path expression by the grammar developer can result in the determination of incomplete left or right validation attributes. This effect can also happen in the case of interacting errors (cf. Section 5.3). In either case some other, correctly specified path expression might not be verified by those right/left validation attributes and would therefore be presented as a potential error candidate.

The above results are also valid for multiple errors if the errors do not interfere with each other. Interference can occur if the grammar allows for a direct transition from one erroneous category to another one by a single RHS element or by a sequence of rule applications where each RHS element is assigned the empty path expression (cf. Section 5.3).

In-depth evaluation with different grammars and multiple errors is needed to further understand the effects of their mutual interdependencies.

9 Other Applications of the Algorithm

The algorithm described in this work can be applied to any rule-based system using path expressions and categories. Language generation with the formalism used in TG/2 or XtraGen served as our main application area, but a brief look to the non-linguistic area of Extensible Stylesheet Language Transformations (XSLT, see XSLT 2006) reveals a much wider usability.

Let us note first that an interesting parallel can be drawn to Lexical-Functional Grammar (LFG), in which syntactic c-structures are mapped onto surface-semantic f-structures, both being tree-like representations. Path expressions denote what type of f-structure can be constructed from a rule application. The LFG formalism differs from the present formalism used in TG/2 mainly by allowing regular expressions in the paths. This kind of recursion would have to be represented in TG/2 through rules. In addition it is important to observe that f-structures form an ontological level in LFG theory, whereas TG/2 aims at a maximum degree of flexibility to deal with any kind of input structures. Path expressions are thus changed rather opportunistically.

Let us now turn to XSLT. XSLT is an XML-based language to define the transformation of an XML document into some arbitrary format. XSLT can be used to implement a general mapping mechanism between specific formats. An XSLT document consists of several transformation rules in XML syntax. Each rule can contain strings and/or a list of commands which are executed sequentially.

An XSLT rule can access certain parts of an entire XML document by specifying an XPath expression (see XPath 1999). An XPath expression can be specified either absolutely or relative to a context node describing the current position inside the XML source document. At the beginning this node is set to the root node of the entire document. It can be changed during the transformation process. An XSLT rule can call other XSLT rules, thus allowing for recursive application. The call can be accompanied by a path expression, in which case a new context node is set to the node reached by following the path starting from the previous context node.

Similar to the grammars we can extract grammar and validation attributes for the calling and the called XSLT rule. Note however that there exist some differences:

- We associated grammar and validation attributes to categories. Since the concept of categories does not exist in XSLT we have to associate the attributes directly to the rules.
- In our grammars, no absolute path expressions exist. Note that in case an XSLT path is specified absolutely it is only possible to determine grammar and validation attributes of the XSLT rule called, but not of the calling XSLT rule.
- There exist many more different types of path expressions in XSLT than in the grammars, e.g. paths including wildcards. Currently these must be ignored; our algorithm would have to be extended to fit to such cases, too.

Another way to activate a rule in XSLT is to define a trigger accompanied by a path expression. Such a rule is activated if the given path is reachable from the current context node. Rules which are defined in this way might be activated at the very beginning of the transformation process. This makes them comparable to grammar categories that can appear in the derivation

tree directly underneath the start category and should be treated analogously. There are further cases, in which an XSLT rule include path expressions. For instance a command may generate surface text, which is located inside an XML tag. Most of these cases can be handled similar to terminal RHS elements in tree mapping grammars.

After discussing how grammar and validation attributes could be extracted from XSLT rules, let us look at what corresponds to the input structures, namely, the XML documents to be transformed. Instead of attribute value pairs, we handle XML tags, which are always ordered in a tree-like representation. Hence we can easily extract the parent-child relationships as required by our algorithm. On the other hand, if a document type description or an XML schema definition is available for the XML input documents this information can be extracted directly from these descriptions.

10 Conclusion and Further Work

An algorithm for the detection and correction of path expressions and categories for tree-mapping grammars with a context-free backbone has been developed and implemented. The evaluation results showed that this work can be a valuable support for grammar developers. Practical tests in the context of NLG grammar development will probably cut down the development time considerably.

Still various improvements of our algorithm are desirable. For instance, it is currently not possible to decide automatically if a category or a path expression should be corrected. Therefore the user has to explicitly select which kind of correction should be done.

Further research includes the extension of the algorithm to longer path expressions, a systematic evaluation of mutually dependent errors, and the treatment of constraint errors. Constraints are a formal element of eGram grammar rules that allows for the percolation of e.g. agreement features across the derivation tree (Busemann, 1996). The detection and correction of missing equations and inconsistent value assignments will be of interest.

References

- Stephan Busemann. eGram—a grammar development environment and its usage for language generation. In *Proc. 4th Conference on Linguistic Resources and Evaluation (LREC)*, Lisbon, Portugal, 2004.
- Stephan Busemann. Ten years after: An update on TG/2 (and friends). In *Proc. 10th European Workshop on Natural Language Generation*, Aberdeen, Scotland, 2005.

- Stephan Busemann. Best-first surface realization. In *Proc. 8th International Workshop on Natural Language Generation*, Herstmonceux, Univ. of Brighton, England, 1996. Donia Scott.
- Stephan Busemann. Generierung natürlichsprachlicher Texte. In Günther Görz, C.-R. Rollinger, and J. Schneeberger, editors, *Handbuch der Künstlichen Intelligenz*, pages 783–814. Oldenbourg Wissenschaftsverlag, 2000.
- Ann Copestake and Dan Flickinger. An open source grammar development environment and broad-coverage english grammar using hpsg. In *Proc. 2nd Conference on Linguistic Resources and Evaluation (LREC)*, Athens, Greece, 2000.
- Richard Crouch, Tracy Holloway King, John T. Maxwell III, Stefan Riezler, and Annie Zaenen. Exploiting F-structure input for sentence condensation. In Miriam Butt and Tracy Holloway King, editors, *The Proceedings of LFG04 Conference*, Canterbury (UK), 2004.
- Gregory T. Daich, Gordon Price, Bryce Raglund, and Mark Dawood. Software test technologies report, 1994.
- Hans-Ulrich Krieger and Ulrich Schäfer. TDL – a type description language for constraint-based grammars. In *Proc. 15th International Conference on Computational Linguistics (COLING)*, Kyoto, Japan, 1994.
- Sabine Lehmann, Stephan Oepen, Sylvie Regnier-Prost, Klaus Netter, and al. TSNLP – Test suites for natural language processing. In *Proc. 16th International Conference on Computational Linguistics (COLING)*, Copenhagen, Denmark, 1996.
- Jens Palsberg and Michael L. Schwartzbach. Object-oriented type inference. In *OOPSLA'91 ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Language and Applications*, pages 146–161, Phoenix, Arizona, 1991.
- Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest*. Dpunkt Verlag, 2003.
- Holger Stenzhorn. XtraGen. A natural language generation system using Java and XML technologies. In *Proc. 2nd Workshop on NLP and XML*, Taipeh, Taiwan, 2002.
- Michael White. Efficient realization of coordinate structures in combinatory categorial grammar. *Research on Language and Computation*, 4(1):39–75, 2006.
- XPath 1999. XML path language (XPath) version 1.0 W3C recommendation, 1999. URL <http://www.w3.org/TR/xpath>.

XSLT 2006. XSL Transformations (XSLT) version 2.0, W3C proposed recommendation 21 November 2006, 2006. URL <http://www.w3.org/TR/2006/PR-xslt20-20061121>.

Andreas Zeller. Locating causes of program failures. In *Proc. 27th International Conference on Software Engineering (ICSE)*, Saint Louis, Missouri, USA, 2005.