Automatic Error Correction for Tree-Mapping Grammars

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

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

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 result is implemented in a grammar development workbench and helps accelerating the grammar writer's work considerably.

1 Introduction

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 (Busemann, 1996). Grammar rules specify which type of (partial) input structure they can interpret. Such grammars may consist of thousands of rules. Debugging is quite tedious and sometimes very time-consuming. During grammar development the generation process often fails at some stage because 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. In developing NLG grammars for the systems TG/2 (Busemann, 1996; Busemann, 2005) or XtraGen (Stenzhorn, 2002) using the workbench eGram (Busemann, 2004), it became obvious that up to 60% of the development time was used to correctly specify the mappings of subtrees.

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 those rules could possibly be corrected. This is achieved by cross-validating grammar rules with the given input structures. The runtime is proportional to the number of grammar rules. The implementation is 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 the available test input structures. We may safely assume that a representative 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 those also using the given input structures.

The remainder of this paper is organized as follows. Section 2 overviews related work on grammar test methods. In Section 3 we introduce some formal background on input structures and grammars. Section 4 describes the detection and correction methods. Some evaluation is provided in Section 5.

2 Related Work

We are not aware of other work on automatic error location in generation grammars. However (Zeller, 2005) describes a dynamic test al-

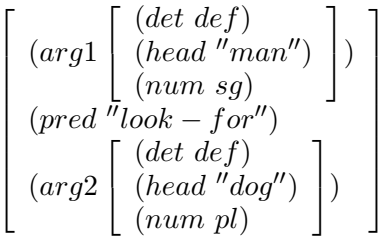

Figure 1: A Sample NLG Input Tree as a Feature Structure.

gorithm for computer programming languages 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. 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 in (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)) that does not rely on executing the underlying NLG system.

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 1, using standard matrix notation.

With a set of context-free grammar rules, in which each non-terminal right-hand side (RHS) category is assigned a substructure of the current input, a derivation tree can be generated. In Figure 2 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.

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

This section provides some formal underpinnings. We first specify the function *psel* to return the part of an input feature structure that is located at the end of the path described by a list of attributes. Let $first$, $last$ and $rest$ be functions over lists that return the first, the last or all elements except the first, respectively. Let further 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:

$$sel([(a_1 w_1)...(a_n w_n)], a_i) = w_i$$

Note that if $a_i \notin \{a_1, ..., a_n\}$ then w_i is the empty feature structure, denoted by [ ]. The function *sel* can be recursively extended to include a list of attribute names, called a path expression, as follows: $psel : F \times A^* \rightarrow F$ with

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

If the specified path expression is empty, the entire feature structure is returned. Instead of writing $psel(s, p)$ we also use the infix notation $p \bullet s$.

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

A path expression can be assigned to a path variable. The usage of path variables bears the advantage of introducing a further abstraction level, which is also useful for error correction. In order to find an appropriate correction, only the small subset of all possible path expressions has to be searched that is assigned to path variables.

Next we turn to the definition of the context-free grammar rules used for tree mapping. Any RHS element is either a terminal symbol (e.g., a string) or a non-terminal category associated with a path variable. This path variable defines the part of the input structure that can be accessed by the rule that is selected by the generation component to further expand the RHS category in the derivation tree.

Consider some node n in a derivation tree with category C. Let $v_1, ..., v_m$ be the path variables assigned to each RHS category in the

course of the derivation from the root node to node n and *value* be a function from path variables onto path expressions. Then a rule applied to category C can access the feature structure s contained in the input structure according to

$$value(v_n) \bullet ... \bullet value(v_1) \bullet s$$

This behavior is illustrated in the sample derivation tree in Figure 2. Its edges are labelled with the path variable names and, following the colon, their values. The nodes are labelled with pairs (C, s) of the category name and the associated part of the input structure.

Furthermore, a grammar rule $R : C \rightarrow A_1[v_1], ..., A_n[v_n]$[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: $\forall i \in \{1, ..., n\} : value(v_i) \bullet s \neq [\,]$.

4 Correction Algorithm

For the automatic correction we will compare the attributes specified by path variables with those that may occur in some input structure. Since path variables are associated to 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 we currently deal only with path expressions of a length ≤ 2. Since longer path expressions do hardly occur in our practice, we decided to leave it to future work to cover such cases as well.

In the remainder of the paper we use the following grammar rules to illustrate the algorithm:[3]

$$\begin{array}{rcl} R_1 : START & \rightarrow & \text{"from"}\ TIME[v_{from} : /from] \\ & & \text{"to"}\ TIME[v_{to} : /to] \\ R_2 : TIME & \rightarrow & toString^4[v_{hour} : /hour] \\ & & toString^4[v_{min} : /min] \end{array}$$

[2]We use C to denote a category symbol and $A_i[v_i]$ to denote a RHS element that has a path variable associated to it. A_i is either a category symbol or a string-valued function over some input structure, giving rise to a terminal element of the derivation tree. We ignore terminal elements (strings) as they do not carry path variables.

[3]To save space, the values of the path variables are included into the rules.

We assume the following input structure is given:

$$\begin{array}{l} [(from\ [(hour\ '12')(min\ '20')] \\ \quad (to\ [(hour\ '12')(min\ '30')])] \end{array}$$

Let us further assume that the grammar developer erroneously specified v_{from} instead of v_{min} in rule R_2 and that 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* 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.

4.1.1 Grammar attributes

Consider all rules with left-hand side (LHS) C that contain one or several RHS elements with path variables. The right attributes of a category C, derived from the grammar, are defined as the set of the first components of the values of these path variables. 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:

$$\begin{array}{rcl} attr_{r,g}(C) & = & \{a | \exists R \in Rules\ : \\ & & R : C \rightarrow A_1[v_1]...A_n[v_n]\ \wedge \\ & & (first(value(v_i)) = a \vee \\ & & value(v_i) = / \wedge \\ & & A_i \in Categories \wedge \end{array}$$

[4]*toString* is a string-valued function adding some input structure, e.g., a string, directly to the output string.

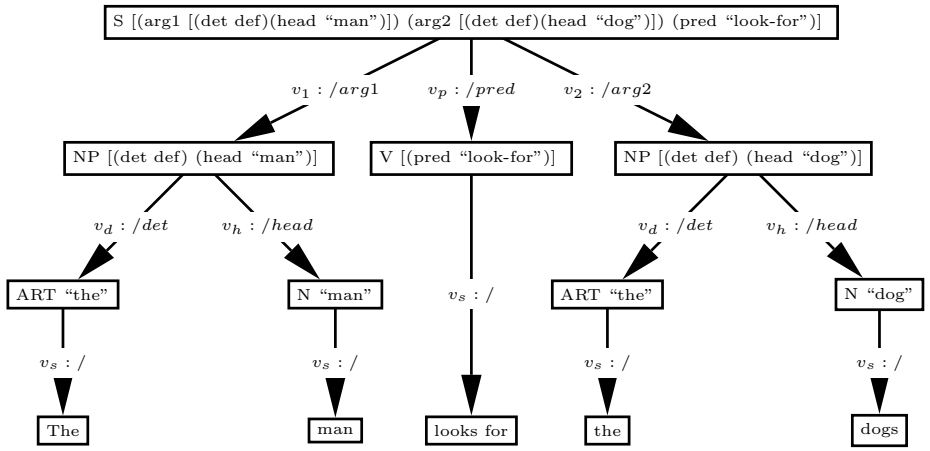

Figure 2: Derivation Tree Generated Using the Input From Figure 1.

$$a \in attr_{r,g}(A_i)) \wedge$$
$$1 \leq i \leq n\}$$

In the derivation tree (see Figure 2) the right grammar attributes of a category contain all first elements of the path expressions attached to the edges that are leaving from that category.

Now consider RHS elements with a category A_i, which are associated with path variables. 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:

$$\begin{aligned} attr_{l,g}(A_i) \quad = \quad & \{a | \exists R \in Rules \ : \\ & R : C \rightarrow A_1[v_1]...A_n[v_n] \ \wedge \\ & (last(value(v_i)) = a \vee \\ & value(v_i) = / \wedge \\ & A_i \in Categories \wedge \\ & a \in attr_{l,g}(C)) \wedge \\ & 1 \leq i \leq n\} \end{aligned}$$

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 following right and left grammar attributes can be determined:

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

4.1.2 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: $children : A \rightarrow 2^A$

$$\begin{aligned} b \in children(a) \quad \Leftrightarrow \quad & \exists s \in Inp, f \in^R s : \\ & sel(f, a) = [...(b, w)...] \end{aligned}$$

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

$$\begin{aligned} children(top) \quad := \quad & \{a | \exists s \in Inp \wedge (a, b) \in^R s \\ & \wedge \ \nexists c : a \in children(c)\} \end{aligned}$$

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. $hour \in children(from)$.

We now describe the attributes associated to 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 R' a rule that expands A_i (cf. Figure 3). Using the last element a_m of the (non-empty) path expression v_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 v_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:

$$
\begin{aligned}
attr_{r,v}(A_i) \quad = \quad & \{a | R \in Rules \ : \\
& R : C \rightarrow A_1[v_1]...A_n[v_n] \wedge \\
& (a \in children(last(value(v_i))) \vee \\
& value(v_i) = / \wedge \\
& a \in attr_{r,v}(C)) \wedge \\
& 1 \leq i \leq n\}
\end{aligned}
$$

Note that the right validation attributes of the start category[5] are just the attributes without parents: $attr_{r,v}(START) = children(top)$.

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 2), as well as the underlying rule R with LHS category C. Note that the top level attributes of that input structure should always be subset of the right validation attributes of C. The right grammar attributes of C derived 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.

We now define left validation attributes in a similar way. Consider a rule $R : C \rightarrow A_1[v_1]...A_n[v_n]$ with $value(v_i) = /a_1/.../a_m$ (cf. Figure 4), where v_i is not assigned an empty path expression. The top-level attributes of the input structures rule R operates on are a subset of all parents a of a_1 ($a_1 \in children(a)$). For a given category C and for all rules with C as

[5]The start category is the top-most category in a derivation tree.

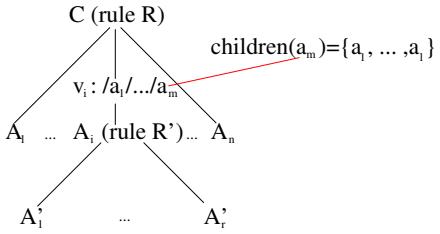

Figure 3: Right Validation Attributes: retrieving the children of a_m.

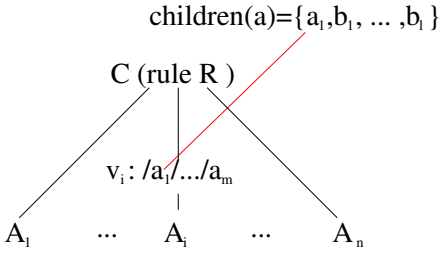

Figure 4: Left Validation Attributes: retrieving the parents of a_1.

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:

$$
\begin{aligned}
attr_{l,v}(C) \quad = \quad & \{a | \exists R \in Rules \ with \\
& R : C \rightarrow A_1[v_1]...A_n[v_n] \wedge \\
& (first(value(v_i)) \in children(a) \vee \\
& value(v_i) = / \wedge \\
& a \in attr_{l,v}(A_i)) \wedge \\
& 1 \leq i \leq n\}
\end{aligned}
$$

Note that there is no left validation attribute of the start category: $attr_{l,v}(START) = \emptyset$.

In our sample grammar the following right and left validation attributes can be determined as follows:

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

4.2 Identifying incorrect path variable occurrences

Basically a path variable 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:

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

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:

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

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 resolve such ambiguities.

4.3 Correcting invalid path variables

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

- incorrect rule;
- incorrect RHS element of that rule;
- wrong path variable appearing in that element;
- possible correct path variables.

A grammar error is due to the grammar writer either selecting the wrong path variable or using a wrong definition of the correct path variable. In the first case the correct path variable is already defined in the grammar and just has to be retrieved. In the second case no automatic correction can be made as the correct definition is unavailable. In this section, we concentrate on the first case.

A correct path variable must fulfill the following conditions:

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

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

$$\begin{aligned} V_c(A_i) &= \\ \{v \in V &: first(value(v)) \in attr_{r,v}(lhs(A_i))\} \\ &\cap \\ \{v \in V &: last(value(v)) \in attr_{l,v}(A_i)\} \end{aligned}$$

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

$$\begin{aligned} V_c(A_i) &= \\ \{v \in V &: first(value(v)) \in attr_{r,v}(lhs(A_i))\} \end{aligned}$$

The special path variable v_{self} containing the empty path expression is predicted as a possible correction as well if the right/left attributes of A_i and $lhs(A_i)$ seem to be identical:

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

V_c may contain multiple elements as a unique solution cannot always be found. In this case several heuristics may be applied to rule out some of the candidates. For instance, one

heuristic we use exploits the fact that usually the same path variable does not occur twice in connection with the same category in a single rule. Such variables are discharged in favor of less frequent ones.

In our example grammar the set of possibly correct path attributes is evaluated to $attr_{r,v}(TIME) = \{hour, min\}$. Therefore the path variable v_{from} occurring in rule R_2 has to be replaced by either v_{hour} or v_{min}. Applying the above heuristic yields the unique solution v_{min}, which is actually correct.

4.4 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 start category are always 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 right validation attributes of the start category's daughter categories, thus maintaining a correct set of attributes for further processing. However, some right grammar attributes of a daughter category may no longer by 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.

For the same reason left attributes are processed in reverse order.[6]

5 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 plugin offers menu items for displaying the set differences between validation and grammar attributes as well as the suggested corrections. The right and left grammar and validation attributes together with the RHS elements they are derived from can be displayed as well. Errors must be manually corrected within eGram.

The algorithm was evaluated on two grammars, the larger one (gr. 2 in the following table) having 270 rules and 111 input structures. Both grammars were verified to be correct. First we evaluated how many of the RHS of both grammars' rules, which we assumed to be correct, were indeed classified as correct by our algorithm ("Recognised correctness"). Second we evaluated the recall of errors found after inserting an erroneous path variable randomly into the grammar. In 200 trials it was counted how often the grammar modification was recognised by our algorithm.

Criterion	gr. 1	gr. 2
Recognised correctness	100%	98%
Total correct detections	88%	64%
Correct corrections 2	85%	49%
Correct corrections 1	58%	45%

"Total correct detections" specifies how often the incorrect RHS element and associated path variable could be detected correctly. "Correct corrections 1" ("Correct corrections2") specifies how often one (at most two) path variables were suggested for correction, and one of them was correct indeed.

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

- Multiple suggestions and overlooks may arise if a transition in the grammar from one category to another can occur in connection with several different path variables.
- Wrong path variables at terminal elements may yield multiple suggestions since the related paths cannot be checked using left validation attributes (cf. our guiding example).

[6] Actually the usage of this algorithm for left validation attributes needs a heuristic, which is beyond the scope of this paper.

- 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 variable 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 4.4). In either case some other, correctly specified path variable might not be verified by those right/left validation attributes and would therefore be presented as a potential error canditate.

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 error catgory to another one by a single RHS element or by a sequence of calls where each RHS element is assigned the empty path expression (cf. Section 4.4).

Further evaluation with different grammars and multiple errors is needed to better understand the effects of their mutual interdependencies.

6 Conclusion and Further Work

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

Sometimes the algorithm specified so far indicates a grammar error although the grammar developer specified the correct path variable, but used a wrong category. This algorithm has been successfully extended to also correct wrong LHS categories. Consider a rule R with a wrong LHS side category C. For a correct category C' we require that the right grammar attributes of C that are derived from R be a subset of the right validation attributes of C': $attr_{r,g,R}(C) \subset attr_{r,v}(C')$ (and analogously for the left attributes).

Future 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.

Acknowledgement

We 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. 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).

References

Stephan Busemann. 1996. Best-first surface realization. In Donia Scott, editor, *Proc. 8th INLG Workshop*, Herstmonceux, Univ. of Brighton, England.

Stephan Busemann. 2004. eGram – a grammar development environment and its usage for language generation. In *Proc. 4th LREC*, Lisbon, Portugal.

Stephan Busemann. 2005. Ten years after: An update on TG/2 (and friends). In *Proc. 10th ENLG Workshop*, Aberdeen, Scotland.

Gregory T. Daich, Gordon Price, Bryce Raglund, and Mark Dawood. 1994. Software test technologies report.

Hans-Ulrich Krieger and Ulrich Schäfer. 1994. *TDL* – a type description language for constraint-based grammars. In *Proc. 15th COLING*, Kyoto, Japan.

Sabine Lehmann, Stephan Oepen, Sylvie Regnier-Prost, Klaus Netter, and al. 1996. TSNLP – Test suites for natural language processing. In *Proc. 16th COLING*, Copenhagen, Denmark.

Andreas Spillner and Tilo Linz. 2003. *Basiswissen Softwaretest*. Dpunkt Verlag.

Holger Stenzhorn. 2002. XtraGen. A natural language generation system using Java and XML technologies. In *Proc. 2nd Workshop on NLP and XML*, Taipeh, Taiwan.

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