PBML

The Prague Bulletin of Mathematical Linguistics
NUMBER 111 OCTOBER 2018 97–112

A Probabilistic Approach to Error Detection&Correction for Tree-Mapping Grammars

Tim vor der Brück

School of Information Technology, Lucerne University of Applied Sciences and Arts, Switzerland

Abstract

Rule-based natural language generation denotes the process of converting a semantic input structure into a surface representation by means of a grammar. In the following, we assume that this grammar is handcrafted and not automatically created for instance by a deep neural network. Such a grammar might comprise of a large set of rules. A single error in these rules can already have a large impact on the quality of the generated sentences, potentially causing even a complete failure of the entire generation process. Searching for errors in these rules can be quite tedious and time-consuming due to potentially complex and recursive dependencies. This work proposes a statistical approach to recognizing errors and providing suggestions for correcting certain kinds of errors by cross-checking the grammar with the semantic input structure. The basic assumption is the correctness of the latter, which is usually a valid hypothesis due to the fact that these input structures are often automatically created.

Our evaluation reveals that in many cases an automatic error detection and correction is indeed possible.

1. Introduction

In NLG, one common task is to transform a set of nested feature-value pairs into a constituency tree structure by means of a set of grammar rules. These grammatical rules are often context free production rules enriched by context-sensitive constraints. Figure 1 illustrates the assumed generation model.

While implementing the grammar, a grammar developer will probably commit errors. On the one hand, such an error can be conceptual, i.e., the developer did not take something into account that is crucial for the functioning of the generation process. Such errors can easily require a major redesign of the grammar. On the

 Corresponding author: tim.vorderbrueck@hslu.ch
Cite as: Tim vor der Brück. A Probabilistic Approach to Error Detection&Correction for Tree-Mapping Grammars. The Prague Bulletin of Mathematical Linguistics No. 111, 2018, pp. 97–112. doi: 10.2478/pralin-2018-0009.

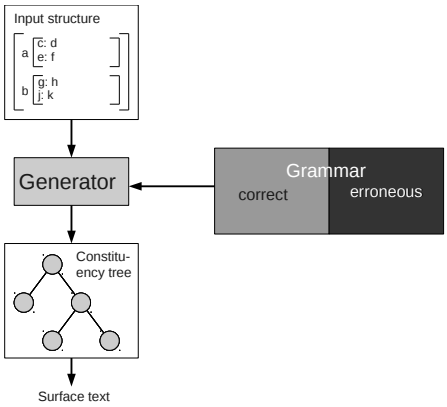

Figure 1: Generation Model.

other hand, an error can be a simple oversight, for instance, the grammar developer misspelled a path expression or a category name. The latter type of errors are the ones we will focus here.

The effect of a grammar error can be manifold. In case the grammar developer accidentally omitted an RHS (right hand side) of a rule, the generated text will probably be incomplete. If he instead switched two RHS, then the output will most likely be scrambled. In both of these cases, the developer will usually get a good hunch regarding the nature and location of the error already by looking at the generated text. However, if the grammar developer accidentally selected the incorrect category or path expression, then in many cases the generation process will fail completely. Such errors are particularly hard to trace. A further source of difficulty for error analysis is the use of recursion, which can result in a deeply nested constituency tree.

In this work, we propose a statistical approach to automatically detecting incorrect path expression and category selections and to providing suggestions for correcting these kinds of errors. It is based on the static grammar verification method introduced by vor der Brück and Busemann (2006) and contains comprehensive and considerable extensions, such as a more detailed evaluation, an in-depth description of a statistical disambiguation method, and discussions of false alarms. The term "*path expression*" is coined to access parts of the input structure. This is quite similar to the term "*relative XPath expression*" in XML. The hypothesis stated in this work is that the semantics contained in the available input structures contains sufficient information to detect those errors.

Our method is intended to be used together with template based text generation systems, in which the grammar rules access an externally defined input structure. Examples of such systems are TG/2 (Busemann, 1996), XtraGen (Stenzhorn, 2003)

and D2S (Theune et al., 2001). It is implemented as plugin for the eGram grammar workbench (Busemann, 2004). eGram supports the comfortable development of text generation grammars for the formalisms TG/2 and XtraGen. Furthermore, eGram is seamlessly integrated with the TG/2/XtraGen generation component, which makes it possible to view the result of the generation process directly inside the environment. The most part of a grammatical rule definition using eGram is acomplished by selecting the appropriate path variables (path variables, see page 104), access functions and categories from several comboboxes.

While errors concerning the rule syntax are practically impossible, there are additional types of errors that cannot occur when writing the grammar with a simple text editor. For instance, it is easily possible that the grammar developer clicks on the category or path expression that is displayed below or above the category / path expression that was actually intended. A typical error that can occur when either using eGram or a text editor is that the grammar developer copies a rule, modifies it afterward to build a new one and the modification is incomplete. This type of error is also possible with eGram since this editor provides a similar functionality.

2. Related Work

Gardent and Narayan (2012) as well as vor der Brück and Stenzhorn (2008) describe a dynamic method that identifies errors in generation grammars by running a generator on semantic input structures. A static approach, however, as proposed here is usually much faster. For instance, our system can do several grammar verification iterations in a few seconds. Furthermore, a static approach does not suffer from potential endless recursion preventing a termination of a dynamic error analysis. For a distinction between static and dynamic approaches see Daich et al. (1994).

There is also some prior work conducted on automated error detection for parsing (the opposite operation to generation). Kok et al. (2009) and van Noord (2004) present an approach, where a large corpus is parsed by an analyzer and n-grams leading to a parsing failure are marked as suspicious. These suspicious n-grams can then be used to track down errors in the tokenizer, lexicon and grammatical rules.

Checking linguistic grammars is closely related to program code analysis since a generator is nothing else than a certain type of software. Zeller (2005) proposed a dynamic testing algorithm to determine the causes of program crashes (segmentation faults). This algorithm isolates the error by subsequently executing different parts of the computer program with varying program states, which is called delta debugging.

3. Input Structure and Grammar Formalism

An input structure is a semantic representation of a certain domain. We assume that it can be structured as nested name-value pairs. The primitive value can either be a string or a number. The concatenation of two input structures forms a new input; the

value part of an input structure can again be an input structure. Formally, we define an input structure as follows.

Definition 1. *An input structure* **InStruc** *defines the semantics of a domain, assigns names to components within the domain, and records values of these components, where components at the same granularity shall be named differently.*

$$\begin{aligned} \mathbf{InStruc} &::= \mathtt{string}|\mathtt{number} && (1)\\ \mathbf{InStruc} &::= (\mathtt{name}, \mathbf{InStruc}) && (2)\\ \mathbf{InStruc} &::= \mathbf{InStruc}; \mathbf{InStruc} && (3) \end{aligned}$$

An input structure is a labeled tree structure. Its edges are labeled with names; its nodes are input structures; its leaves can either be strings or numbers. Children of the same node are labeled with different names.

For example, the following input structure describes the semantics of a temporal duration. The corresponding labeled tree structure is illustrated in Formula 4. Each leaf node in the tree is uniquely identified by the sequence of labels starting from the root node. For example, the sequence "*from, hour*" uniquely identifies the leaf node 10. Generally, given a start node, a label sequence defines a travel from this start node. A travel is a selection of nodes among the descendant nodes.

$$\left[\begin{array}{ll} \mathtt{from} & \left[\begin{array}{ll} \mathtt{hour}: & 10\\ \mathtt{min}: & 20\\ \mathtt{sec}: & 30 \end{array}\right]\\ \mathtt{to} & \left[\begin{array}{ll} \mathtt{hour}: & 12\\ \mathtt{min}: & 30\\ \mathtt{sec}: & 10 \end{array}\right] \end{array}\right] \tag{4}$$

Definition 2. *Given an input structure* **InStruc**, *a sequence of strings* $a_1, \ldots, a_n$ *is a* **path expression** *of* **InStruc**, *if and only if, for any* i *there is an input structure* $\mathbf{InStruc}_i$, *such that* $(a_i, \mathbf{InStruc}_i)$ *is within* **InStruc**. *A* **path expression** $a_1, \ldots, a_n$ *is written as* $[a_1/\ldots/a_n]$. *The selection process is defined as follows.*

$$[a_1/\ldots/a_n] \circ \mathbf{InStruc} = \begin{cases} [a_2/\ldots/a_n] \circ \mathbf{InStruc}_{a_1}, & (a_1, \mathbf{InStruc}, \mathbf{InStruc}_{a_1})\\ \emptyset & \text{otherwise} \end{cases} \tag{5}$$

where $(a_1, \mathbf{InStruc}, \mathbf{InStruc}_{a_1})$ *means that there is an edge labeled with* a_1 *between* **InStruc** *and* $\mathbf{InStruc}_{a_1}$.

The uniqueness of the selection process is guaranteed by the structure of the input. For the convenience of computation, we define the empty path expression $[\epsilon]$ as follow.

$$[\epsilon] \circ \mathbf{InStruc} = \mathbf{InStruc} \tag{6}$$

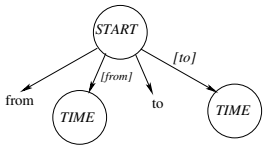

Figure 2: Tree representation of a grammar rule that generates a time interval.

Our task is to transform a given input structure into a linguistic surface structure. In the former example, we need to transform the semantic representation of the temporal duration into the phrase "*from 10:20:30 to 12:30:10*". To this end, we employ grammatical rules.

The grammar considered here follows the TG/2 formalism (Busemann, 1996, 2005), which is also used by the XtraGen generator (Stenzhorn, 2003). TG/2 is basically a context-free grammar formalism consisting of production rules. Each production rule consists of one LHS (left hand side) category and several RHS (right hand side) categories or functions or simple surface text strings. The RHS categories and functions are associated to path expressions that specify the part of the input structure accessible for such categories / functions.

$$\begin{aligned} &\mathtt{R}_1 : \mathit{START} \rightarrow \text{ from } \mathit{TIME}[\mathit{from}] \text{ to } \mathit{TIME}[\mathit{to}] \\ &\mathtt{R}_2 : \mathit{TIME} \rightarrow \mathit{toString}([\mathtt{hour}]) : \mathit{toString}([\mathtt{min}]) : \mathit{toString}([\mathtt{sec}]) \end{aligned} \tag{7}$$

START is the top-level category the generation process kicks off with. "*toString*" is a function that generates the part of the input structure that is referenced by the associated path expression. These rules are used to display a time expression in a formatted way. A tree representation of rule $\mathtt{R}_1$ is given in Figure 2.

After applying rule $\mathtt{R}_1$ on the category *START* with the input structure depicted in formula 4, an incomplete constituency tree is created with two preliminary leaf nodes labeled with *TIME*. Rule $\mathtt{R}_2$ is then applied afterward on the *TIME* nodes and can only access the part of the input structure selected by the path expression [*from*] (or [*to*] respectively). If converted into an absolute path expression, the path expression "*hour*", located on the right side of rule $\mathtt{R}_2$, would evaluate to [*/from/hour*] (or [*/to/hour*] respectively). The generated surface string for the given input structure is: "*from 10:20:30 to 12:30:10*".

4. Error Identification with Left and Right Attributes

We define parent-child relationship for labels.

Definition 3. *A label* $\mathtt{a}$ *is defined to be a parent label of label* $\mathtt{b}$*, if* $\mathtt{a}$ *is the label of an input structure* $\mathbf{InStruc}$*, such that* $\mathtt{b}$ *is a label within* $\mathbf{InStruc}$*.*

Next, we define left and right grammar attributes of categories. If an RHS of a rule with LHS category C is labeled with the path expression $[a_1, \ldots, a_n]$, the path component a_1 is added to the set of right attributes of category C. If a path is empty, then the right grammar attributes of the associated RHS category are inherited by the LHS category C. Similarly, left grammar attributes of a category are defined as the last path components belonging to a rule transition edge leading to this category. If the path is empty, then the left grammar attributes of the LHS category are inherited. The right attributes give a characterization of a category. Consider, for example, the categories *START* and *TIME* and the two rules from the last section:

R_1 : *START* $\rightarrow$ from *TIME*[*from*] to *TIME*[*to*].

R_2 : *TIME* $\rightarrow$ *toString*([*hour*]) : *toString*([*min*]) : *toString*([*sec*])

The right grammar attributes of *START* are "*from*" and "*to*", the right grammar attributes of *TIME* are "*hour*", "*min*", "*sec*".

In addition, we define the right and left validation attributes of categories. Validation attributes build a superset of all allowed grammar attributes and are extracted from both the input structure and the grammar. Consider the case that there exists an RHS labeled with a path "*pe*" leading to category C. Let "*e*" be the last component of path "*pe*". Then all possible child elements in the input structures of "*e*" belong to the right validation attributes of C. This is the case for all RHS of rules leading to category C. If "*pe*" is the empty path expression, then the right validation attributes of the LHS category are added to C. If C is the START (top-level) category, then all top-level input structure attributes are the right validation attributes of C. Formally, the right validation attributes can be specified as follows (vor der Brück and Busemann, 2006):

$$r_C := \begin{cases} \{d | \exists R \in \mathit{Rules}, \\ \mathit{pe} \in \mathit{PE}, D \in \mathit{Cat} : \\ R : D \rightarrow C[\mathit{pe}] \wedge \\ ((\mathtt{parent}(\mathit{pe}[|\mathit{pe}|], d) \vee \\ (\mathit{pe} = \varepsilon \wedge d \in r_D)\}, & C \neq \mathtt{START} \\ \mathtt{top}, & \mathtt{otherwise} \end{cases} \tag{8}$$

where

- *Rules*: set of all rules
- *Cat*: set of all categories
- *PE*: set of all path expressions
- *parent*(a, b): a relation that is fulfilled if and only if a occurs as parent of b in any of the input structures
- *top*: all elements occurring at the top-level of any input structure

The left validation attributes of C are defined similarly. Let C be the LHS category of a rule with one RHS labeled with path "*pe*". Let us consider the first element of

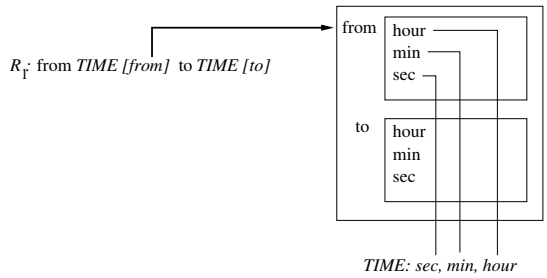

Figure 3: Construction of right validation attributes.

this path called "f". Then all possible parents of "f" in the input structures belong to the left validation attributes of C. If "*pe*" is the empty path, then all left validation attributes of the RHS category are added to the LHS category. This is the case for any such RHS. The process of extracting right and left validation attributes is illustrated in Figures 3, 4, and in Table 2. Formally, the left validation attributes of C are given as:

$$\begin{aligned} \mathrm{l_C} := \{ \mathrm{d} | \exists pe \in PE, \mathrm{R} \in Rules : \\ \mathrm{R} : \mathrm{C} \rightarrow \mathrm{D}[pe] \wedge \\ (\mathtt{parent}(\mathrm{c}, pe[1]) \vee \\ pe = \varepsilon \wedge \mathrm{d} \in \mathrm{l_D}) \end{aligned} \tag{9}$$

where D is either a category, a string-valued function or a string.

In order for a rule to be applicable with the current input structures, the right grammar attributes of its LHS (RHS) category must be contained in the right validation attributes of its LHS (RHS) category. Similarly, the left grammar attributes must be contained in the left validation attributes. To handle multiple errors, right grammar attribute mismatches are identified top-down (beginning with the START category). In this way, the right validation attributes that were introduced due to incorrect RHS paths can be removed. Analogously, left grammar mismatches are identified bottom up.

Now let us consider an example error of the grammar developer. We assume that he wanted to enter rules R_1 and R_2 but made a mistake at rule R_2.

$$\mathrm{R}_{2,\mathtt{error}} : \mathit{TIME} \rightarrow \mathit{toString}([\mathtt{to}]) : \mathit{toString}([\mathtt{min}]) : \mathit{toString}([\mathtt{sec}])$$

The right validation attributes of *TIME* do not change, but the right grammar attributes of *TIME* now contain the attribute "*to*", which is not contained in the right validation attributes of *TIME* and, therefore indicate an error in this RHS. The correct path must begin with one of the right validation attributes of *TIME*, i.e., "*hour*",

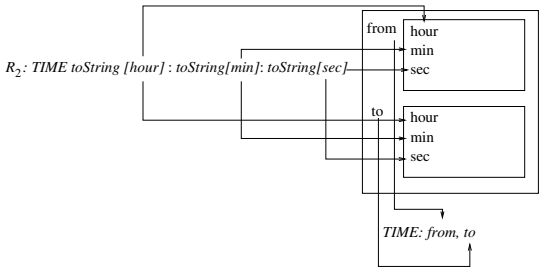

Figure 4: Construction of left validation attributes.

Cat	Left	Right
TIME	from,to	to,min,sec
START	-	from,to

Table 1: Grammar attributes derived by the rules R_1 and $R_{2,\mathtt{error}}$.

Cat	Left	Right
TIME	from,to	hour,min,sec
START	-	from,to

Table 2: Validation attributes derived by the rules R_1 and $R_{2,\mathtt{error}}$.

"*min*" or "*sec*". Table 1 shows the grammar and Table 2 the validation attributes of the categories. Since the selection of *min* or *sec* would lead to the same word generated multiple times (called double generation in short), which is quite unlikely in practice, hour can be correctly selected. Now consider a different error. Let us assume that the grammar developer wrote

$$R_{1,\mathtt{error}} : \mathit{START} \rightarrow \text{from } \mathit{TIME}[\mathit{min}] \text{ to } \mathit{TIME}[\mathit{to}] \tag{10}$$

instead of the correct rule R_1. The right validation attributes of the *START* category are not affected by this error and are the top-level input structure attributes "*to*" and "*from*". Thus, the attribute "*min*" is not contained in the right validation attributes of category *START* and is therefore detected as erroneous. The correct path expression must be either *[to]* or *[from]*. Again, the path *[to]* would lead to a double generation. Therefore, the correct path expression must be *[from]*. Note that in general an arbitrary number of path expressions can be created by combining attributes by path separators. In practice, these expressions are reduced to a finite set by the fact that the grammar editor requires all used path expressions to be associated to path variables. So only all existing path variable values have to be checked. In the example above, we employed a disambiguation heuristic to find a unique solution. In practice, there are many cases where the correct solution cannot so easily be found. Thus, in addition to

the heuristic "*double generation*", a statistical heuristic, which is explained in Section 6 in more detail, was used.

5. False Alarms

The errors detected by the method described here are only guaranteed to be actual errors (under the precondition that the input structures are always correct) if the empty path expression is never used in a grammar rule. It is actually possible that a false alarm is produced if a category can be reached from two different parent categories and one of the transitions is connected to the empty path expression. Consider for example the following grammar and input structure, which might not follow good design principles, but still leads to a successful generation of "Text2 Text1".

$$
\begin{aligned}
Q_1 &: \mathit{START} \rightarrow B[\epsilon] \\
Q_2 &: \mathit{START} \rightarrow C[\epsilon] \\
Q_3 &: B \rightarrow D[\epsilon] \\
Q_4 &: C \rightarrow D[a] \\
Q_4 &: D \rightarrow \mathit{toString}([c]) \\
Q_5 &: D \rightarrow \mathit{toString}([d]) \\
\text{Input} &: \begin{bmatrix} a & [d\ \text{“}\mathit{Text1}\text{“}] \\ c & \text{“}\mathit{Text2}\text{“} \end{bmatrix}
\end{aligned}
$$

B inherits among others the right grammar attribute d from category D, which is not contained in B's right validation attributes (a and c). One solution for this problem is to add all right validation attributes of C to category B or more generally: If a category C_1 is connected by the empty path expression with its child category C_2, all right validation attributes of all other categories leading to C_2 are added to C_1[1]. A similar approach can be employed for left attributes. The possible rule applications are visualized in Figure 5.

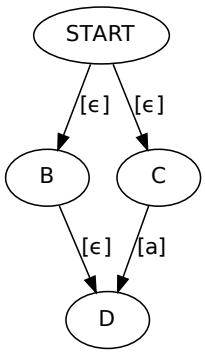

Figure 5: Graph that visualizes the rules where a false alarm is produced by the error-recognition algorithm.

[1]Note that this feature is not yet implemented in our current system.

6. Statistical approach to automatic error correction

Suppose that the error could be successfully spotted, but several path expressions are possible for correcting the erroneous RHS and the disambiguation methods mentioned above still leave several potential candidates. Now the statistical disambiguation method comes into play.

This method estimates, how probably a certain rule candidate is given all existing rules (the input structures are disregarded) and suggests the most probable one(s). Actually, the path expression '*pe*' is chosen that maximizes the posterior probability that the path expression '*pe*' occurs, given a transition from category LHS C_1 to RHS C_2, which can be formalized as follows:

- $pa : E \rightarrow P$: a function that assigns an edge $e \in E$ to a path expression $pe \in P$ (for the definition of a rule edge see Figure 2)
- $s : E \rightarrow Cat$: source category of an edge
- $d : E \rightarrow Cat$: destination category of an edge
- *Cat*: set of categories, *E*: set of edges

$$
\begin{aligned}
pe' &= \operatorname*{argmax}_{pe} P' \text{ with} \\
P' &= P(pa(e) = pe | s(e) = C_1 \wedge d(e) = C_2) \\
&\quad \text{(Theorem of Bayes)} \\
&= P(pa(e) = pe) \cdot \\
&\quad \frac{P(s(e) = C_1 \wedge d(e) = C_2 | pa(e) = pe)}{P(s(e) = C_1 \wedge d(e) = C_2)} \\
&\quad (P(s(e) = C_1 \wedge d(e) = C_2) \text{ is independent of } pe) \\
pe' &= \arg\max_{pe} P'' \\
P'' &= P(pa(e) = pe) \cdot \\
&\quad P(s(e) = C_1 \wedge d(e) = C_2 | pa(e) = pe) \\
&\quad \text{(Now we make the assumption that} \\
&\quad s(e) = C_1 \text{ and } d(e) = C_2 \\
&\quad \text{are approximately conditionally independent} \\
&\quad \text{given } pa(e) = pe. \\
&\quad \text{This assumption is made} \\
&\quad \text{to handle the sparse data problem} \\
&\quad \text{that usually shows up in} \\
&\quad \text{hand-written grammars.)}
\end{aligned}
$$

$$
\begin{aligned}
&\approx \quad P(pa(e) = pe)P(s(e) = C_1|pa(e) = pe) \cdot \\
&\qquad P(d(e) = C_2|pa(e) = pe) \\
&\qquad \text{(applying Theorem of Bayes again)} \\
&= \quad P(pa(e) = pe) \cdot \\
&\qquad P(pa(e) = pe|s(e) = C_1) \cdot \\
&\qquad P(pa(e) = pe|d(e) = C_2) \cdot \\
&\qquad \frac{P(s(e) = C_1)P(d(e) = C_2)}{P^2(pa(e) = pe)} \\
&= \quad P(pa(e) = pe|s(e) = C_1) \cdot \\
&\qquad P(pa(e) = pe|d(e) = C_2) \cdot \\
&\qquad \frac{P(s(e) = C_1)P(d(e) = C_2)}{P(pa(e) = pe)} \\
&\qquad (P(s(e) = C_1) \text{ and } P(d(e) = C_2 \\
&\qquad \text{are independent of } pe) \\
pe' &\approx \quad \arg\max_{pe} P''' \\
P''' &= \quad P(pa(e) = pe|s(e) = C_1) \cdot \\
&\qquad \frac{P(pa(e) = pe|d(e) = C_2)}{P(pa(e) = pe)}
\end{aligned}
$$

If there exists no RHS category due to the fact that the RHS is a function, then we instead determine the path expression pe with

$$\underset{pe}{\text{argmax}} P(pa(e) = pe|s(e) = C_1). \tag{11}$$

The most probably path expression(s) as obtained above are then suggested as correction. As usual, the probabilities are estimated by relative frequencies.

Let us now consider an example for this procedure. We extend the original grammar (rules R_1 and R_2, see page 102) by the following rules:

$$
\begin{aligned}
&R_3 : \mathit{TIME} \rightarrow \mathit{toString}([\mathit{hour}]) : \mathit{toString}([\mathit{min}]) \\
&R_4 : \mathit{TIME} \rightarrow \mathit{toString}([\mathit{hour}]) \\
&R_5 : \mathit{START} \rightarrow \text{from } \mathit{toString}([\mathit{from/loc}]) \text{ to} \\
&\qquad \mathit{toString}([\mathit{from/loc}])
\end{aligned} \tag{12}
$$

and add a second input structure:

$$
\begin{bmatrix} \mathit{from}: & [\mathit{loc} : \mathit{Bonn}] \\ \mathit{to}: & [\mathit{loc} : \mathit{Cologne}] \end{bmatrix} \tag{13}
$$

Let us now consider the case that rule R_2 was entered erroneously in the following way. Instead of the correct rule

$$R_2 : \ TIME \rightarrow toString[hour] : \\ toString[min] : toString[sec] \tag{14}$$

the grammar developer accidentally entered the incorrect rule:

$$R_{2,err} : TIME \rightarrow toString[from] : \\ toString[min] : toString[sec] \tag{15}$$

The analysis with validation and grammar attributes results in the fact that the incorrect path expression [*from*] has to be replaced by either [loc], [hour], [min], or [sec]. [min] and [sec] can be ruled out by the double generation rule so the alternatives [*from*] and [*loc*] remain possible. Now the probabilistic rule comes into play. The probability for [*loc*] is given by:

$$P(pa(e) = [loc] | \, \mathrm{source}(e) = TIME) = 0.0 \tag{16}$$

and the probability for [*hour*] is given by:

$$P(pa(e) = [hour] | \, \mathrm{source}(e) = TIME) = 1/3 \tag{17}$$

since there are 6 rule RHS with LHS *TIME* and 2 of them are associated to the path expression [*hour*]. Thus, the path expression [*hour*] is selected. Another heuristic is name comparison. Often, the paths and associated RHS categories have similar names. Therefore, we prefer paths that contain common substrings with the LHS/RHS categories. Finally, we use a heuristic that the empty path should not be suggested for LHS category, for which the set of right validation attributes is not empty, since presumably such a category matches a non-leaf node in the input structure.

Note that in some cases the generation fails because the path might be correct but the LHS (or RHS) category of some rule might be incorrectly chosen. In this case, instead of looking for the correct path we have to look for the category that best fulfills the validation attribute constraints of the erroneous rule.

7. Evaluation

For the evaluation, we used a grammar generating natural language descriptions of houses comprising of 267 rules, 96 categories and 104 different path variables. This grammar is assumed to be correct (errors displayed already for the unmodified grammar are ignored in the evaluation).

In general, there are two possibilities how to conduct the evaluation. The intrinsic approach is to look directly at the grammar and determine how many of the incorrect categories or path expression can be corrected. In contrast, we could evaluate

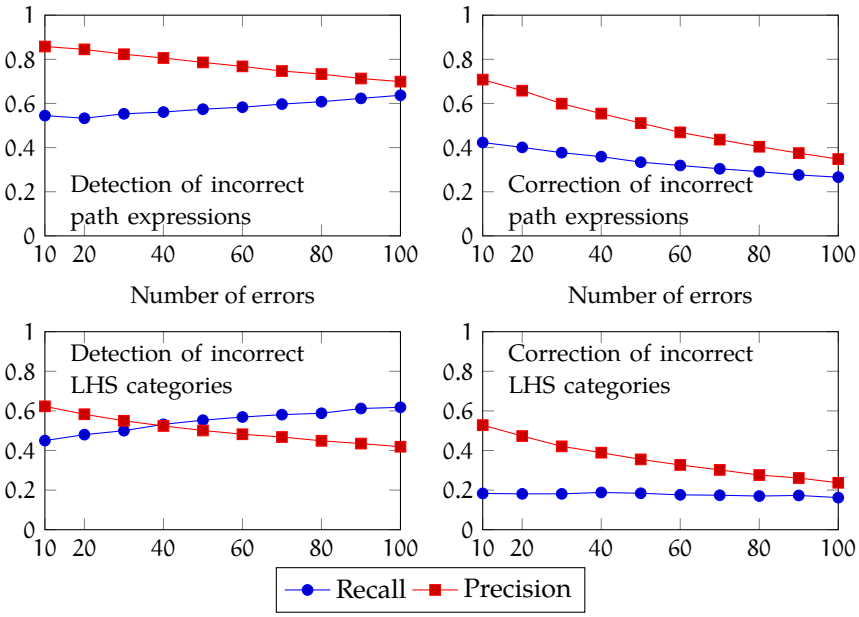

Figure 6: Recall and precision of detection (left) and correction (right) of incorrect path expressions (top) and LHS categories (bottom).

	Detection		Correction	
Errors	Prec.	Recall	Prec.	Recall
10	0.545	0.858	0.423	0.708
20	0.533	0.845	0.401	0.658
30	0.553	0.823	0.377	0.599
40	0.561	0.806	0.359	0.554
50	0.574	0.786	0.334	0.511
60	0.583	0.768	0.319	0.469
70	0.597	0.747	0.304	0.436
80	0.608	0.733	0.291	0.404
90	0.623	0.713	0.276	0.375
100	0.637	0.699	0.266	0.348

Table 3: Precision and recall for the correction of invalid path expressions.

	Detection		Correction	
Errors	Prec.	Recall	Prec.	Recall
10	0.450	0.623	0.183	0.528
20	0.480	0.583	0.181	0.473
30	0.500	0.550	0.181	0.421
40	0.532	0.524	0.188	0.389
50	0.553	0.501	0.184	0.355
60	0.569	0.482	0.176	0.327
70	0.581	0.468	0.174	0.302
80	0.588	0.449	0.170	0.276
90	0.612	0.435	0.173	0.261
100	0.618	0.419	0.162	0.237

Table 4: Precision and recall for the correction of invalid LHS categories.

our approach also extrinsically by comparing the actually generated text with the expected output and calculate some typical performance score like BLUE/METEOR or ROGUE. However, the extrinsic evaluation is not really adequate in this scenario, since most of the errors covered here would result not in a scrambled output but in no output at all. Thus, by employing an extrinsic evaluation, it would usually not be possible to discern for example, whether one or two errors were resolved. Likewise, it could normally not be decided, whether our method failed completely or was at least able to spot the error but then suggested the wrong correction. Hence, we opted to evaluate our approach intrinsically only. In particular, we insert errors randomly assuming a uniform error distribution by either choosing an RHS and modifying the path expression (path errors) or by selecting an LHS and changing its category (LHS category errors). In practice, a uniform distribution of the errors is rather unlikely. For instance, we would expect path expressions that are located nearby the correct one in the GUI list box to be chosen more often than path expressions that are far away. Unfortunately, it is very hard to obtain real error data. The number of inserted errors are varied from 10 to 100 in steps of 10. For each number of errors, this procedure is repeated one thousand times. See Figure 6 and Tables 3 and 4, for recall and precision of detection and correction of incorrect path expressions and LHS categories.

The precision of the error detection approach is defined by the quotient of the number of correctly determined errors and the number of total errors displayed. The percentual number of errors, which were detected correctly is called the recall of the error detection. An error is considered as detected if the incorrect rule and RHS (LHS in case of category errors) are recognized correctly.

The precision of the error correction is defined by the quotient of the number of the correct error-correction suggestions divided by the number of all suggestions. The percentual number of cases where the correct suggestions were found is called the recall of the error correction. For the evaluation of the error correction, we only regard the cases where the error was correctly detected. Note that there usually exist a lot of possible modifications that would make the grammar correct. However, for practical reasons, we only considered such a modification correct if it is exactly the inverse operation of the conducted modification. The evaluation showed that the erroneous RHS could be identified with an average precision of 54.5% (for 10 randomly inserted errors), which is far above the random baseline of $<1/267$ (analogously for the correction of rules), which means that our hypothesis that the input structures contain enough information to detect errors automatically cannot be rejected (significance level: 1%). The most difficult to detect are errors involving the empty path expressions, which can introduce a lot of ambiguities. Moreover, the recall degrades with an increasing number of errors, which is caused by the fact that left and right validation attributes become less reliable for error detection if the grammar contains a lot of errors. In contrast, the precision of our error detection approach is increasing with the number of errors since the proportion of erroneous rules become larger and therefore the a priori probability that a selected rule is erroneous increases.

For comparison, we converted the grammar into the XtraGen format and applied the method of (vor der Brück and Stenzhorn, 2008). Unfortunately, we did not get any result after several hours, which might be caused by an endless recursion in the generator, and aborted the run. A comparison with the approach of Gardent and Narayan (Gardent and Narayan, 2012) is not directly possible, since the authors focus especially on inputs in form of dependency trees and employ a generator based on tree adjoining grammars. However, in our method we assume a context free generation process, while tree adjoining grammars are actually context sensitive. Additionally, we do not make any assumption about the nature of our input despite being hierarchically ordered. In particular, the input structure of the grammar used for this evaluation constitutes no dependency tree.

8. Conclusion

A generation grammar might contain errors that result in an empty output for a given input structure. An empty output gives, in contrast to an ill-formed output, almost no clues about the reason for the generation failure. We presented and evaluated a method to detect and propose possible corrections for such errors automatically. The evaluation showed that in many cases a detection and correction was indeed possible.

For future work, we plan to investigate how to decide if a path expression or a category is actually incorrect. Also, we plan to recognize errors in RHS categories as well.

Currently, our approach is only used for checking text generation grammars. However, it could also be used to verify arbitrary XSLT stylesheets containing XPath expressions. Instead for categories we would then extract right and left attributes for stylesheet rules. The contents of the match attribute of an XML template would contribute both to the left grammar attributes as well as to the right validation attributes.

A completely self-correcting XSLT stylesheet or text generation grammar is still out of reach, but nevertheless, some ideas and concepts are shown how this goal can become a reality.

Acknowledgments

Hereby I thank the DFKI for its support of this work, especially for granting me free access to the eGram grammar workbench. Special thanks go to the associate head of their natural language processing group, Prof. Dr. Stephan Busemann.

Bibliography

Busemann, S. Best-first surface realization. In *Eight International Natural Language Generation Workshop*, pages 101–110, Brighton, England, 1996.

Busemann, S. eGram - A Grammar Development Environment and Its Usage for Language Generation. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC)*, Lisbon, Portugal, 2004. URL `http://www.dfki.de/dfkibib/publications/docs/busemann-LREC04.pdf`.

Busemann, S. Ten Years After: An Update on TG/2 (and Friends). In Wilcock, Graham, Kristiina Jokinen, Chris Mellish, and Ehud Reiter, editors, *Proceedings of the Tenth European Natural Language Generation Workshop (ENLG 2005)*, pages 32–39, Aberdeen, UK, 2005.

Daich, G., G. Price, B. Raglund, and M. Dawood. Software Test Technologies Report, 1994. URL `http://citeseer.ist.psu.edu/daich94software.html`.

Gardent, Claire and Shashi Narayan. Error Mining in Dependency Trees. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*, Jeju Island, South Korea, 2012.

Kok, Daniël, Jianqiang Ma, and Gertjan van Noord. A generalized method for iterative error mining in parsing results. In *Proceedings of the 2009 Workshop on Grammar Engineering Across Frameworks (GEAF)*, Suntec, Singapore, 2009.

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

Theune, M., Esther Klabbers, Jan Odijk, J.R. De Pijper, and Emiel Krahmer. From Data to Speech: A General Approach. *Natural Language Engineering*, 7(1), 2001.

van Noord, Gertjan. Error Mining for Wide-Coverage Grammar Engineering. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics (ACL)*, Barcelona, Spain, 2004.

vor der Brück, Tim and Stephan Busemann. Automatic Error Correction for Tree-Mapping Grammars. In *Proceedings of KONVENS 2006*, pages 1–8, Konstanz, Germany, 2006. ISBN 3-89318-050-8.

vor der Brück, Tim and Holger Stenzhorn. A Dynamic Approach for Automatic Error Detection in Generation Grammars. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI)*, Patras, Greece, 2008.

Zeller, A. Locating Causes of Program Failures. In *27th International Conference on Software Engineering (ICSE)*, Saint Louis, Missouri, USA, 2005.

Address for correspondence:
Tim vor der Brück
`tim.vorderbrueck@hslu.ch`
Suurstoffi 41b, CH 6343 Rotkreuz