Logic grammars for diagnosis and repair

Christiansen, Henning; Dahl, Veronica

Published in:
ICTAI'02

DOI:
10.1109/TAI.2002.1180819

Publication date:
2002

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Christiansen, H., & Dahl, V. (2002). Logic grammars for diagnosis and repair. In *ICTAI'02: Proceedings of 14th IEEE International Conference on Tools with Artificial Intelligence, November 4-6, 2002 Washington DC* (pp. 307-314). IEEE. https://doi.org/10.1109/TAI.2002.1180819

Logic Grammars for Diagnosis and Repair

Henning Christiansen
Computer Science Dept.
Roskilde University
P.O.Box 260, DK-4000 Roskilde, Denmark
henning@ruc.dk

Veronica Dahl
Dept. of Computer Science
Simon Fraser University
Burnaby, B.C., Canada
veronica@cs.sfu.ca

Abstract

We propose an abductive model based on Constraint Handling Rule Grammars (CHRGs) for detecting and correcting errors in problem domains that can be described in terms of strings of words accepted by a logic grammar. We provide a proof of concept for the specific problem of detecting and repairing natural language errors, in particular, those concerning feature agreement. Our methodology relies on grammar and string transformation in accordance with a user-defined dictionary of possible repairs. This transformation also serves as top-down guidance for our essentially bottom-up parser. With respect to previous approaches to error detection and repair, including those that also use constraints and/or abduction, our methodology is surprisingly simple while far-reaching and efficient.

1. Introduction

Speech input for computer based applications is becoming a realistic goal with the popularization of reasonably priced and reasonably efficient speech recognition software, such as Naturally Speaking or Microsoft Speech Agent. Even for processing written NL input, the need for more robust language analyzers is increasing, due to the growing masses of internet documents requiring intelligent search and processing. Many of these documents are written in informal or more colloquial language than those prepared for traditional publications, which are screened by referees and editors. In consequence, modern natural language processing systems must include intelligent and efficient ways of parsing and correcting ill-formed input.

The inherent ambiguity of natural language, and the frequency of errors that appear particularly in spoken or colloquial input, suggests bottom-up parsing techniques as the most promising, as we should be able to gather, from an attempt to produce a correct parse, information which does not necessarily conform to strict grammar rules. If possible, we should also be able to interpret that information in ways which allow us to somehow make sense of this "incorrect" input, just as humans do.

Inspired by successes in blending constraints with abduction for diagnosis and repair problems (cf. the next section), we set out to investigate the possibilities of using abduction with a particularly interesting kind of constraint reasoning (that embedded in Constraint Handling Rules or CHRs [18]) with abduction. Our approach uses CHRs in their grammatical incarnation CHR Grammars (CHRGs), which has proved valuable for natural language processing problems as demonstrated in [13]. In [14] it is shown that CHRGs provide a direct realization of abductive language interpretation as bottom-up deductive derivations without any meta-level overhead as is usually associated with abduction. The application to error detection and correction developed in the present paper yields a much simpler mechanism than those proposed earlier.

We find it important not only to produce a parse that represents a feasible interpretation of an input string, but also to be able to return an explanation to the user saying that when certain substrings were changed into certain others, the system was able to interpret the string. Given a grammar G and string S, the problem is to find a modification M to S (i.e., a set of word changes) so that a successful parse of the entire string becomes possible. This is basically an abductive problem as qualified guesses for M, which is a premise for the successful parse, need to be made. Furthermore, the proposed candidates for M should be minimal which can be defined in two ways, by set inclusion or by cardinality; we shall make no preference of the two as our methods can easily be tuned for one or the other. We can demonstrate sound and complete procedures, but our main purpose is to provide a framework for strategies that make qualified guesses (rather than try all possibilities) by means of additional rules that identify errors at higher level syntactic nodes and generate modification requests to the string so that parsing can continue using the normal grammar rules.

The computational process can be characterized as a bottom-up computation with occasional top-down sweeps when errors are detected, and the parse as well as the modification are read out of the final state. No backtracking is involved, all relevant parses due to ambiguity or alternative modifier sets are produced in parallel with no syntactic node produced more than once. Moreover, our approach can also be applied dynamically, as the input string is being typed. In this respect, our approach rely on the flexibility of CHR to combine top-down and bottom-up computations in a uniform way as pointed out by [1, 2].

It is to be noted that, while in this paper we develop one specific application in detail as proof of concept — namely, the automatic correction of feature agreement errors, as in *"the referees was impressed"* :-), our main contribution is a general methodology suitable for correcting errors in problem domains that can be described in terms of strings of words accepted by a logic grammar (i.e., not only natural but also formal languages, molecular biology sequences, etc.).

2. Background

Traditional approaches to syntactic error correction typically resorted to ad-hoc techniques such as constraint relaxation, until it became clear that encompassing and accurate coverage of errors could only be achieved by taking natural language processing techniques into account.

Relaxation techniques work by relaxing one by one different kinds of constraints from the rules (such as number agreement) in order to enable the recognition of the input string as a sentence and infer its possible correction. Some such systems, e.g. EPISTLE/CRITIQUE [19, 22] or [24] use augmentations of phrase structure grammars, others use chart parsers [6, 20], others [17] use extensions of the PATR-II formalism [23]. These systems are characterized by a great degree of procedurality, and heavily draw on heuristics to decide what the possible and the preferred corrections are. While [17] does allow a declarative specification of possible relaxation of the rules, it shows no clear declarative procedure for generating corrections for hypothesized input errors.

A declarative method that restates the problem in abductive terms, solvable through contradiction removal in a logic program [4, 5] was proposed in [7]. This solution involves a first phase which attempts to fully parse the input string, and a second phase which is triggered if the first one fails, and which sets up the framework for abducing minimal (in the sense of subset ordering) sets of explanations. One problem with this approach is that work needs to be redone because during the second phase, given that the partial parses obtained in the first phase are lost. In [8] this memory loss problem was addressed by using Datalog grammars [15] to obtain a database with all the possible partial parses for the input string. In the second phase, using this database as contextual information, an abductive approach similar to the one proposed in [7] was set up in order to detect if the input string is a correct or a faulty sentence, and propose appropriate changes if the latter. Datalog grammars were further exploited in view of efficiency: by making word boundary information about the input string explicit, they allowed the authors to prune the search space, thus avoiding an explosion in the abductive procedure which resulted in a 15 percent improvement of the response time relative to [7].

The advantages of bottom-up approaches have been demonstrated within other recent declarative paradigms for parsing and sometimes also repairing incorrect input. Blache, for instance, uses constraint graphs, built from constraints over categories such as linear precedence (e.g. "`det` must precede `n`"), exclusion (which restricts co-occurrence between sets of categories), uniqueness (for categories which cannot be repeated in a phrase) [9]. Kakas et al. propose a technique for database repair which could perhaps be adapted to syntactic error repair as well, and which involves interleaving abduction and the resolution steps of a CLP framework [21]. They use abductive criteria to prune the search space as well as to solve the problem at hand.

Relaxation based methods typically might need to transform grammars to less closely related ones, so they must identify, for instance through constraint ordering, which constraints (e.g. gender and number agreement) are more important to relax, which not to try when others have failed, etc. In contrast, the declarative approaches of [7, 8] are simpler in that they do not require the ordering of constraints.

However, they do involve a substantial apparatus: Datalog Grammars, abductive reasoning and model-based fault diagnosis, as well as a two-phase processing scheme.

In this paper we shall argue that we can get similar and maybe even better results than those obtained in the recent declarative work surveyed, through CHR grammars' [13] inherent treatment of ambiguity and abduction with considerably less work from both the implementors and the users:

a) as in [9] or [8], we can build non-connected structures showing all partial successful analyzes (but without having to replace trees by graphs as in the former, or needing extra machinery as in the latter). Moreover, we can repair ill-formed input as well, whereas [9] merely detects it, even though it is capable of parsing sentences containing associated elements such as hesitations, repetitions, etc. (these are largely ignored after being detected).

b) As in [21] and [8], we use abduction in conjunction with bottom-up analysis, but we neither require a special interleaving proof procedure as in [21] (although the effects we obtain from the use of CHRGs are similar to those of interleaving), nor extra machinery for datalog grammars, abduction and model-based diagnosis as in [8].

3. CHR as grammar formalism with facilities for error correction

CHR Grammars, or CHRGs for short, were introduced in [11, 13] as a bottom-up counterpart to Definite Clause Grammars (DCGs) (DCGs; [**?**]) CHR in exactly the same ways as DCGs take their semantics from and are implemented by a direct translation into Prolog. CHRGs are executed as CHR programs that provide robust parsing with an inherent treatment of ambiguity that makes the present approach to error diagnosis and correction feasible. In case no parse is given for the entire string, specialized rules can examine the subphrases generated and activate suitable repair actions. As demonstrated by [13, 14], CHRG is a very powerful system which provides straightforward implementation of assumption grammars [16], abductive language interpretation and a flexible way to handle a variety of linguistic phenomena [12]. For ease of understanding, and to focus on the logical semantics, we write grammar rules in CHR syntax. CHR works on constraint stores with its rules interpreted as rewrite rules over such stores. A string to be analyzed such as *"a boy laughs"* is entered as a sequence of constraints

```
token(0,1,a), token(1,2,boy),
token(2,3,laughs)                    (1)
```

that comprise an initial constraint store. The integer arguments represent word boundaries, and a grammar for this intended language can be expressed in CHR as follows.

```
token(X0,X1,a) ==> det(X0,X1,sing).
token(X0,X1,boy) ==> n(X0,X1,sing).
token(X0,X1,laughs) ==> v(X0,X1,sing).
token(X0,X1,laugh) ==> v(X0,X1,plu).
n(X0,X1,Number), v(X1,X2,Number) ==>
       s(X0,X1,Number).              (2)
```

We can do here basically with the subset of CHR consisting of propagation rules only for which it is easy to specify declarative and procedural semantics.

Definition 1 *A CHR* program *is a finite set of* rules *of the form*

Head `==>` *Guard* `|` *Body* (3)

where Head and Body are conjunctions of atoms and Guard a test constructed from built-in predicates; the variables in Guard and Body occurs also in Head; in case the Guard is the local constant "true", it is omitted together with the vertical bar. Its logical meaning is the formula $\forall(Guard \rightarrow (Head \rightarrow Body))$ *and the meaning of a program is given by conjunction.*

A derivation *starting from an initial state called a* query *of ground constraints is defined by applying rules as long as it adds new constraints to the store. A rule as above* applies *if it has an instance* H`==>`G`|`B *with* G *satisfied and* H *in current store, and it does so by adding* B *to the store.*

In the few cases we go beyond this subset, we give informal explanations; for full introduction to CHR, see [18]. We notice the following properties:

- Any state in a derivation is ground.
- For a derivation for program P and query Q with final state F, we have $A \in F$ iff $P \cup Q \models A$.
- If the application of a rule adds a constraint c to the store which already is there, no additional rules are triggered, e.g., p`==>`p does not loop as it is not applied in a state including p.

Running the program (2) on the initial store given by (1) produces instances of grammar symbols corresponding to subtreenodes, including one covering the whole string, namely `s(0,3,sing)`.

Now suppose that we are given an erroneous string such *"a boy laugh"*. With program (2), the final constraint store contains no s-node, but only nodes corresponding to recognized subtrees. If the token *"laugh"* were changed into *"laughs"*, however, a full parse could be produced. To implement such changes, consider adding the following rule to the program:

```
token(X0,X1,Old), modify(X0,X1,Old,New)
==> token(X0,X1,New)                 (4)
```

If the atom M = `modify(2,3,laugh,laughs)` indicating a change of a word is added to the store *deus ex machina*, a node corresponding to a full sentence will be produced and M can then be viewed as an explanation of how it was reached. Thus the problem of error correction amounts to the abductive problem of finding out which such correction facts should be added to the constraint store in order to provide a full parse.

The methodology of [14] for abductive language interpretation in CHR suggest simply to move the hypothesis to be abduced from the head to the body of the rule and in this way have all necessary abducibles generated as part of the final state. It is not feasible to do it exactly like this in the present setting as any atom would be made subject of a modification.[1] That is why we propose rules for top-down guidance in the following, but it is worth noticing that the only way to produce a new constraint, whether referred to as abducible or not, is to have it appear in the body of a rule.

4. An abductive CHR model for Error Correcting Grammars

We now develop a larger framework for grammars that can correct errors as a side effect of parsing. We first consider only modifications on tokens (i.e., changing one token into another one). Throughout this paper, we shall use agreement features which can be expresses through unification (such as gender, case, number, semantic type) as the main example.

[1] Although not covered by definition 1, [14] can also handle the non-ground abducibles that arise.

Definition 2 *A* grammar *is a set of rules of the form*

$$g_1(X_0,X_1,a_1),\ \ldots,\ g_n(X_{n-1},X_n,a_n)\ \texttt{==>}\ g(X_0,X_n,a) \tag{5}$$

where the predicates g, g_i are called grammar symbols*; a, a_i stand for sequences of terms;arguments referred to by the variables $X_0,\ldots,X_n$ are called boundaries. The special grammar symbol* `token` *of arity 3 occurs only in singletons on the left hand side; such rules are called* lexical*, any other* nonlexical*. For simplicity of the following, we assume a grammatical* start symbol*, typically called s, of some given arity, which does not occur in any lefthand side. No assumption is made on whether a grammar is unambiguous. By a (n* input*)* string*, we refer to a conjunction* `token(0,1,`w_1`)`*, ...,* `token(`$n-1$`,`n`,`w_n`)` *where $w_1,\ldots,w_n$ are constant symbols also referred to as* words*.*

In case of a chain of rules $g_1(X_1,Y_1,A_1)$ `==>` *$g_2(X_2',Y_2',B_2)$, $g_2(X_2,Y_2,A_2)$* `==>` *$g_3(X_3',Y_3',B_3)$, ... $g_{n-1}(X_{n-1},Y_{n-1},A_{n-1})$* `==>` *$g_n(X_n',Y_n',B_N)$ with $g_1=g_n$, there is no B_i of the form $f(A_i)$.*

The restriction on chains of rules ensures that a derivation for a grammar always terminates even if there is a loop among its nonterminals. No assumption is made on whether a grammar is unambiguous; a final state has nodes corresponding all possible phrases recognized inside the string. The possible ways in which a given token can be modified may be derived from a definition of edit distance or expectations of typical grammatical errors, e.g., wrong number. In a realistic system, we would expect this be implemented as a program module referring to an external dictionary that also defines the lexical rules. However, we shall not go into such details here and abstract this aspect away as follows.

Definition 3 *A* dictionary of token modifications *is a set of atoms of the form* `change(`*From*`,`*To*`)` *where From and To are words. A* modification M *with respect to dictionary D is a set of atoms* `change(`$i-1$`,`i`,`*From*`,`*To*`)` *with* `change(`*From*`,`*To*`)` $\in D$ *so that no two atoms in M have the same boundaries. The* modification of string S *by M, denoted S_M, is the string obtained from S by replacing each* `token(`$i-1$`,`i`,`w`)` *by* `token(`$i-1$`,`i`,`w'`)` *whenever* `change(`$i-1$`,`i`,`w`,`w'`)` $\in M$.

For the moment, we shall ignore exactly how these modifications are created, and focus on how given modifications interact with the grammar rules.[2]

Definition 4 *A* modifying grammar G' *for a grammar G is the CHR program resulting from replacing each lexical rule*

$$\texttt{token(}A\texttt{,}B\texttt{,}w\texttt{)}\ \texttt{==>}\ cat(A,B,a) \tag{6}$$

by the following:

$$\texttt{token(}A\texttt{,}B\texttt{,}w\texttt{),}\ \neg\exists w'\texttt{change(}A\texttt{,}B\texttt{,}w\texttt{,}w'\texttt{)}\ \texttt{==>}\ cat(A,B,a) \tag{7}$$

and adding the following general rule:

```
token(A,B,Word), change(A,B,Word,NewWord)
==> token(A,B,NewWord).                          (8)
```

[2] The negated goal in def. 4 can be understood by means of so-called explicit negation, i.e., with $\nexists w'$`change` viewed as one predicate symbols and a suitable "integrity constraint" that excludes inconsistent states.

The following is obvious from the definitions:

Proposition 5 *Given grammar G and its modifying grammar G', a string S, modification M and instance of grammar symbol A, we have*

$$G\cup S_M\models A \qquad \textit{iff} \qquad G'\cup S\cup M\models A. \tag{9}$$

The problem of error correction is then an abductive problem: If, with grammar G, it is required from string S to derive instances of the start symbol $s(X)$ but $G\cup S\not\models\exists X\colon s(X)$, find a modification M so that

$$G'\cup S\cup M\models s(A) \text{ for some } A. \tag{10}$$

Not any modification is desirable, nor is it computationally feasible to test any possible modification. As already mentioned, it is desirable for a modification to be minimal — in terms of set inclusion or cardinality.

5. Top-down guidance

As we have seen, our CHR based parsing proceeds bottom-up, which is most convenient for the purposes of detecting errors as early as possible. However, a completely bottom-up approach would be caught up in combinatorial explosion. A top-down guidance to complement the bottom-up parsing can be implemented quite naturally by specialized error handling rules that comparing the subtrees already produced. These rules may be compiled directly from the grammar but may also be supplied by a competent grammar writer who can take into account typical errors and expectations of what users a likely to wish to express, and perhaps also develop more sophisticated strategies for detection and correction than the one produced automatically. Here we sketch briefly the idea before going on and adapting the model of section 4 to include this.

For instance, a rule requiring number agreement between the noun phrase and the verb phrase of a sentence:

```
np(X0,X1,N), vp(X1,X2,N) ==> s(X0,X2,N).        (11)
```

can compile into an additional rule such as:

```
np(X0,X1,N1), vp(X1,X2,N2) ==> N1\== N2 |
    modify_np(X0,X1,N1,N2) ∨                     (12)
    modify_vp(X1,X2,N2,N1).
```

The disjunction is treated by CHR by means of backtracking. The intended meaning of a constraint `modify_`*cat*`(`A,B,X,Y`)` is a request to modify the string so that the *cat* phrase with boundaries A,B changes its attribute value from existing X to new Y; this rule, then, needs to be complemented by straightforward rules to propagate the modification request down to the token level. In the example above, `modify_np` is applied to enforce the broken agreement to hold in one way or another.

However, we shall propose below a better approach than using backtracking, which would involve undoing and redoing different choices and also require extra bookkeeping in

order to record the different solutions. Instead, we propose a model that investigates all intended modifications in parallel, integrates dynamically with the normal parsing mode and produces minimal modifications. We shall return to the above example after introducing the necessary machinery.

6. Multiple modifications through local constraint stores

Definition 6 *A* multimodification $\mathbf{M}$ *with respect to a dictionary of token modifications D is a set of atoms* `change(`$i-1$`,`i`,`*From*`,`*To*`)` *with* `change(`*From*`,`*To*`)` $\in D$ *(but not necessarily funct. determ. by boundaries).*

A given multimodification $\mathbf{M}$ is taken as a combined representation of all possible modifications $M \in \mathbf{M}$, i.e., each subset M with at most one modifier per boundary pair.

In order to integrate multimodification in our framework so that the consequences of the embedded modifications are evaluated in parallel bottom-up, we extend each grammar symbol with an extra attribute that serves as a local modification store in the following way.

Definition 7 *A* multimodifying grammar G' *for a grammar G is the CHR program resulting from adding one to the arity of each grammar symbol, however also keeping the original* `token`$/3$ *symbol, and changing the set of rules as follows. Add the following two rules:*

```
token(A,B,W) ==> token(A,B,W,∅)
token(A,B,W), change(A,B,W,NewW) ==>
token(A,B,NewW,{change(A,B,W,NewW)}).
```
(13)

Replace each grammar rule of G, lexical as well as nonlexical, of form

```
g1(X0,X1,A1), ..., gn(Xn-1,Xn,An)
==> g(X0,Xn,A)
```
(14)

by the following:

```
g1(X0,X1,A1,C1), ..., gn(Xn-1,Xn,An,C1)
==> g(X0,Xn,A,C1 ∪ ··· ∪ Cn)
```
(15)

Intuitively, $g_i(i,j,A,C)$ means that node $g_i(A)$ is recognized as spanning the substring given by i, j, provided the modifications given by C. Conceptually, we prefer to think of "$\cup$" as an interpreted function symbol with its traditional meaning, but CHR will take it as term-builder and anyhow execute the program correctly according to our intentions; it is straightforward to extend the CHR rules so that "$\cup$" in fact is evaluated. It is easy to prove by induction that a set attached to some grammar symbol in a derivation always classify as a modification, i.e., there are no two `change` atoms for the same boundaries. We have obviously:

Proposition 8 *Given grammar G and its multimodifying grammar G', a string S, multimodification $\mathbf{M}$, modification $M \in \mathbf{M}$ and grammar symbol instance A, we have*

$$G \cup S_M \models A \quad \textit{iff} \quad G' \cup S \cup \mathbf{M} \models A_M \tag{16}$$

where A_M is A with M added as extra argument.

Furthermore, if for some $M \in \mathbf{M}$ with $G \cup S_M \models s(A)$ there exists an $M_0 \subset M$ with $G \cup S_{M_0} \models s(A_0)$ for some A_0, we have that $G' \cup S \cup \mathbf{M} \models s(A_0)_M$.

In other words, if a given multimodification $\mathbf{M}$ is added by means of some strategy, the final constraint store will provide a collections of (among others) set-minimal modifications that lead to a correct parse of the entire input string — provided, of course, that such exists as subset of $\mathbf{M}$. So if the strategy applied for generating token modifications suggests to modify more tokens that necessary, the bottom-up evaluation will anyhow also produce minimal ones.

Set-minimality

In most cases, the generated modifications will be minimal or almost minimal in the set inclusion sense, but referring to the second part of proposition 8 there is a straightforward way to obtain the minimal ones. The following so-called simpagation rule (see [18]) sketched as follows will remove all but s-nodes with set minimal modification.

```
s(0,N,A0,M0)\s(0,N,A,M)<=>M0⊊M | true.
```
(17)

The operational semantics states that constraints in the head following the backslash are removed from the constraint store while the others remain.

Minimality in number of word changes

Among the successful modifications in some multimodification $\mathbf{M}$, we can remove all but these with the smallest number of elements by a rules sketched as follows.

```
s(0,N,A0,M0) \ s(0,N,A,M) <=> card(M0)
< card(M) | true.
```
(18)

This may not result in the absolutely smallest modification in case the strategy that created $\mathbf{M}$ has overlooked a modification that is (partly) outside $\mathbf{M}$.

A brute-force method

We have already the bits and pieces to put together a procedure which is sound and complete with respect to either of the two minimality notions: For given grammar G, string S, and dictionary of possible changes D, generate the multimodifying grammar G', and for any `token(`$i-1,i,w_i$ $\in S$ and `change(`w_i,w_i'`)` $\in D$ let `change(`$i-1,i,w_i,w_i'$`)` $\in \mathbf{M}$. Then start a derivation from initial state $S \cup \mathbf{M}$ using program $G' \cup \{\mu\}$ where μ is the preferred one of (17) and (18). From an existential point of view, we can be satisfied with this procedure but it involves a combinatorial explosion of the constraint store and thus impractical for any but small demonstrative examples.

7. A transformation into autocorrrecting grammars – intuitive description

Here we describe a particular strategy for guiding error correction by means of rules that compare attributes of adjacent phrases in case a grammar rule did not apply although its left-hand side presents the same sequence of grammar symbols. These rule, called local autocorrect rule, are produced automatically by a compilation of the original grammar rules together distribution rules that pass change requests down to the token level. In the present section we

introduce the concept in an intuitive way by means of an example, and in the following section 8 we give a general description. Changes that seem to correct errors may at a later point prove to be wrong, as we advance through the string and have more parsing information. For instance in French we do not have two but three relevant attribute values for "number", namely feminine-singular, masculine-singular, and plural. Consider the text fragment "la garçon" which looks like an `np` with an agreement problem. To fix this, we may modify the determiner or the noun leading to two options, say "le garçon" and "la fille". Either of these two makes it possible to recognize an `np` which is either masculine-singular or feminine-singular. But assume now that it is followed by the plural `vp` "sont fatigues". This indicates a mismatch that triggers a number of possible corrections, one of which is to force the `np` into singular. As actual modifications take place at the level of the *original* tokens, a change into "les garçons" is most likely to be suggested. The local autocorrect rule for the sentence level triggers also changes of the `vp` so it can match either of the proposed singular `np`s. Thus we end up with three sentence nodes whose local modification stores each corresponds to one of the sentences "les garçons sont fatigues", "le garçon est fatigue", and "la fille est fatiguée" each with respectively 2, 3, and all 4 tokens changed.

We now expand a multimodifying grammar in the format of definition 7, so that it can figure out how to correct mistakes. Consider again grammar rule (11). Whenever it cannot apply to an `np` and an adjacent `vp` node, i.e., if we have a mismatch of attribute values as in

```
np(16,18,sing), vp(19,20,plu)                    (19)
```

we will need what we call a *local autocorrect rule* (generated from (11) at compile time):

```
np(X0,X1,N1,_),vp(X1,X2,N2,_)==>N1\==N2 |
modify_np(X0,X1,N1,N2),                          (20)
modify_vp(X1,X2,N2,N1).
```

Notice that with respect to the initial formulation (12) that

- we have an extra argument at the end: the local modification store,
- we now use "and" rather than "or", because we have lifted the unicity requirement, so now we have multimodifying rules, i.e., both ways of modifying the input string will be explored in parallel.

Grammar rules where mismatch is not possible (e.g., `iv(X1,X2,N) ==> vp(X1,X2,N)`) do not generate autocorrect rules. They do, however, generate another kind of rule (top-down distribution rules), as we shall see next.

The compiler also generates, for each grammar rule, a *top-down distribution rule*, in charge of percolating modification requests all the way down to the token level where actual changes can take place. The sample rule

```
det(X0,X1,N),n(X1,X2,N)==>np(X0,X2,N)            (21)
```

generates the top-down distribution rule:

```
det(X0,X1,N,_), n(X1,X2,N,_), np(X0,X2,N,_),
modify_np(X0,X2,N,NewN) ==> N \== NewN |         (22)
    modify_det(X0,X1,N,NewN),
    modify_n(X1,X2,N,NewN).
```

which will apply to all possible `det`-`n` sequences that span the same boundary interval as the `np`[3], whenever a constraint `modify_np` has been issued and provided that the `np` at hand has been constructed from a `det`-`n` sequence.

For instance, if a request to modify a singular np into plural has been issued, rule (22) locates the relevant `det`-`n` sequences and modifies their number accordingly.

Top-down distribution rules are generated for all grammar rules, including lexical ones, with the exception that there is no `modify_token` constraint but we refer to the `change` constraint used in earlier sections.

It is easy to understand the result of this process as the consequence of a set of implication formulas. When executed as CHR rules, they will perform as overlapping waves shifting between bottom-up and top-down execution. Nodes start to grow up from the bottom, and when a mismatch is found, a wave is sent downwards and reflected by from the bottom in terms of new bottom-up reductions. Maybe this gives rise to a parsing of the entire string or another mismatch is identified at a higher level and the story continues. However, termination is guaranteed if the set of possible modifications is finite.

8. A general characterization of autocorrecting grammars

We describe the two transformations referring to the following general pattern for grammar rules.

$$g_1(X_0, X_1, A_{1,1}, \ldots, A_{1,k_1}),\ \ldots,\ g_n(X_{n-1}, X_n, A_{n,1}, \ldots, A_{n,k_n})\ \texttt{==>}\ g(X_0, X_n, A_1, \ldots, A_k). \quad (23)$$

The X_i variables stand for word boundaries and each attribute expressions $A_{i,j}$ is either a constant or a variable. For each grammar symbol g of arity $n+2$ in the original grammar a new constraint symbol `modify_`g of arity $2n+2$ (for $g = \texttt{token}$ coincident with the previously defined `change` constraint). The intuitive meaning of

$$\texttt{modify\_}g(A, B, \mathit{From}_1, \mathit{To}_1, \ldots, \mathit{From}_n, \mathit{To}_n) \quad (24)$$

is a request for modification of some tokens so that $g(A, B, \mathit{From}_1, \ldots, \mathit{From}_n)$ can transform into $g(A, B, \mathit{To}_1, \ldots, \mathit{To}_n)$. If the dictionary of possible changes is capable of suggesting sufficient token changes, this g node (with suitable local modification store attached) will eventually be produced by the grammar rules.

In the general case, it is a dynamic property which modification requests should be generated, so we need to have

[3] In the case of a highly ambiguous grammar, this may trigger a few redundant modification sequences. However, this is not a problem since nonminimal modifications sets are anyhow removed, as explained above.

control expressions in the body of a local autocorrect rule. The autocorrect rule compiled for a grammar rule (23) makes a comparison of the actual attributes values with those expected by the grammar rule. Based on that, a set of perhaps new possible attribute values $V_{i,j}$ is suggested for each position i, j. The general shape of the autocorrect is as follows; the $V_{i,j}$ sets are characterized below and each $Z_{i,j}$ is a new variable.

$$\begin{array}{l} g_1(X_0, X_1, Z_{1,i}, \ldots, Z_{1,k_1}, \_),\ \ldots, \\ \quad g_n(X_{n-1}, X_n, Z_{n,1}, \ldots, Z_{n,k_n}, \_)\ \texttt{==>} \\ \langle Z_{1,1}, \ldots, Z_{1,k_1}, \ldots, Z_{n,1}, \ldots, Z_{n,k_n} \rangle \text{ not an instance} \\ \text{of } \langle A_{1,1}, \ldots, A_{1,k_1}, \ldots, A_{n,1}, \ldots, A_{n,k_n} \rangle \ \texttt{|} \\ \text{for each } i \text{ and each } v_{i,j} \in V_{i,j}, \text{ call constraint} \\ \quad \texttt{modify\_}g_i(X_{i-1}, X_i, Z_{i,1}, v_{i,1}, \ldots, Z_{i,i_n}, v_{i,i_n}) \end{array} \tag{25}$$

As we have said earlier, autocorrect rules are only generated from rules where there is opportunity for mismatch. This is the case when either:

- A variable $A_{i,j} = A$ in (23) stands in a number > 1 of positions at the left-hand where the current constraints display a collection of different values that we call V_A; for any such (i, j) occupied by A let $V_{i,j} = V_A$.
- An attribute position $A_{i,j}$ in (23) is occupied by constant c and the current constraint displays a different constant c'; in that case, let $V_{i,j} = \{c\}$.

For any attribute position not already assigned a $V_{i,j}$ set, let $V_{i,j}$ be a the set consisting of the constant observed in the current constraint. By attribute values in current constraints, we referred to the actual values of the $Z_{i,j}$ variables in an application of the rule.

The general shape of a rule for distribution of change requests eventually to hit the token level is given as follows for each rule of the grammar. Notice that the guard stops requests for modifying a node into itself; this seems to be a simpler approach than actually avoiding the creation of such trivial requests.

$$\begin{array}{l} \texttt{modify\_}g(X_0, X_1, F_1, T_1, \ldots, F_k, T_k), \\ g(X_0, X_1, F_1, \ldots, F_k, \_), \\ \quad g_1(X_0, X_1, F_{1,i}, \ldots, F_{1,k_1}, \_),\ \ldots, \\ \quad g_n(X_{n-1}, X_n, F_{n,1}, \ldots, F_{n,k_n}, \_) \\ \texttt{==>}\ (F_1, \ldots, F_k)\ \texttt{\textbackslash==}\ (T_1, \ldots, T_k)\ \texttt{|} \\ \text{unify } (A_1, \ldots, A_k) \text{ and } (T_1, \ldots, T_k) \text{ and for any} \\ \quad \text{variable } A_{i,j} \text{ that received a value, let } N_{i,j} \text{ be this value,} \\ \text{for any other } i, j, \text{ let } N_{i,j} \text{ be } F_{i,j}, \\ \texttt{modify\_}g_1(X_0, X_1, N_{1,1}, \ldots, N_{1,k_1}),\ \ldots, \\ \texttt{modify\_}g_n(X_{n-1}, X_n, N_{n,1}, \ldots, N_{n,k_n}). \end{array} \tag{26}$$

In most cases, this very general format can be reduced to simpler formulations, such as (22). The distribution rule can be made more precise by a more complicated matching so that it applies to only those particular instances of g_1, ..., g_n to which rule (23) were applied at an earlier stage in order to produce this instance of g. However, this will probably slow down the overall performance in a way that will outbalance what is gained in all but very rare examples. And, as noted already, it does not make any damage to the overall process to explore a few redundant modification.

For lack of space, we shall refrain from a formal characterization and proof of correctness of an autocorrecting grammar G' compiled from some grammar G. However, the overall properties are as follows.

- G' is sound in the sense that if it produces a given node, G can produce the same node from a modified version of the input string.
- G' is complete within a set of parse tree structures that preserve the topology of those produced by G for the original string (and G' adds new nodes on top of those). In fact G' may, so to speak by accident, produce other parses corresponding to a completely totally phrase structure in case of an ambiguous grammar.

9. Discussion

We have presented a general framework which makes it possible to mechanically transform a grammar G into an auto-correcting extension G' given a dictionary of possible modifications defined by the user, and given a CHRG processor such as the one defined on top of CHRs in a standard way. With respect to previous work on both abductive and non-abductive error detection and correction schemes, our approach is the most economical in terms of the machinery needed, while also providing a great degree of efficiency and flexibility. We are not aware of benchmark tests to which we can compare, but our expectations of efficiency is grounded on the absence of meta-level overhead in our approach (which is quite heavy in other abduction-based and transformation-based approaches) and the effective pruning of the search space provided by top-down guidance. It is also farther reaching in many aspects: for instance, Blache's approach requires the input to contain disambiguated POS tags, whereas we can let ambiguous tags coexist, since disambiguation largely follows from which constraints in our constraint store will be satisfied. We can also correct rather than just detect errors, as we have seen. Finally, the original grammar is expressed just like any DCG, except that we reverse the order of body and head to facilitate bottom-up parsing. This is a much simpler and standard way than the constraint graph approach, which necessitates recasting the grammar in terms of six fairly unknown constraints (relationships) between linguistic objects, as well as the representation and manipulation of the resulting constraint graph. However, the benefits we obtain are similar although requiring much less apparatus: we also can avoid the need to build a complete structure before we can verify its properties. Associated elements (such as hesitations, interjections, etc.) which are part of the input without having exact relation to the rest of the sentence, can be "skipped over" by

repairing their input and output word boundaries into the same. Our abductive reasoning requires no special mechanisms either, because it follows from the interactions between our grammar and string transformations and the standard CHR store workings. Finally, let us point out that the auto-correct rules we introduce also serve as top-down guidance without any extra layers of theorem proving, thus improving efficiency in and by themselves. In this sense we might view our approach as comparable to what magic sets achieved for data base theory. Mixing bottom-up and top-down directions does of course have antecedents in parsing theory itself: left-corner parsing, for instance, also uses astute rule transformation which can result from compiling in order to achieve top-down guidance of essentially bottom up-parsing. However our local autocorrect rules serve a double purpose: while designed for correcting errors, they double up as top down guides, — an agreeable side effect.

Acknowledgment: This research is supported in part by the Danish Natural-Science Research Council, the IT-University of Copenhagen, and Canada's NSERC Grants 31-611024 and 31613183.

References

[1] Abdennadher, S., Christiansen, H., An Experimental CLP Platform for Integrity Constraints and Abduction.Proc. FQAS2000, pp. 141–152, *Advances in Soft Computing series*, Physica-Verlag (Springer), 2000.

[2] Abdennadher, S., Schütz, H. CHR$^{\vee}$: A flexible query language. Proc. FQAS'98, *Lecture Notes in Artificial Intelligence* 1495, pp. 1–14, Springer, 1998.

[3] Aho, A.V., Sethi, R., Ullman, J.D., Compilers. Principles, techniques, and tools. Addison-Wesley, 1986.

[4] Alferes, J.J., Semantics of Logic Programs with Explicit Negation. *PhD Thesis, Universidade Nova de Lishoa*, 1993.

[5] Alferes, J.J., and Pereira, L. M., Reasoning with Logic Programming. *LNAI 1111*, Springer Verlag, 1996

[6] Baiao, A. Chart parsing: Repairing morpho-syntactic errors on written portuguese. *Master's thesis*, Universidade Nova de Lisboa, 1994.

[7] Balsa, J., Abductive Reasoning for Repairing and Explaining Errors in Portuguese. *Master's Thesis, Unicersidade Nova de Lisboa*. 1994.

[8] Balsa, J., Dahl, V. and Pereira Lopes, J.G., Datalog Grammars for Abductive Syntactic Error Diagnosis and Repair. *Proc. Natural Language Understanding and Logic Programming Workshop*, Lisbon, 1995.

[9] Blache, P. and Azulay, D. O., Parsing Ill-formed Inputs with Constraints Graphs. *A. Gelbukh (ed), Intelligent Text Processing and Computational Linguistics, LNCS*, Springer, 2002.

[10] Brill, E., Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part-of-Speech Tagging. *Computational Linguistics* 21(4), p. 543–565, 1995.

[11] Christiansen, H., CHR as grammar formalism, a first report. *Proc. Sixth Annual Workshop of the ERCIM Working group on Constraints*, Prague, Czech Republic, June 18-20, 2001. Avaiable eloctronically at CoRR: http://arXiv.org/abs/cs.PL/0106059.

[12] Christiansen, H., *CHR Grammar web site*, http://www.dat.ruc.dk/~henning/chrg/

[13] Christiansen, H., Logical grammars based on constraint handling rules, (Poster abstract). *Proc. Int'l Conference on Logic Programming*, Lecture Notes in Computer Science 2401, p. 481, Springer 2002.

[14] Christiansen, H., Abductive language interpretation as bottom-up deduction. Proc. NLULP 2002, *Datalogiske Skrifter* 92, pp. 33–48, Roskilde University, 2002.

[15] Dahl, V., Tarau, P. and Huang, Y.N. Datalog Grammars, *Proc. GULP-PRODE'94*, 1994.

[16] Dahl, V., Tarau, P., Li, R., Assumption grammars for processing natural language. *Proc. Fourteenth International Conference on Logic Programming.* pp. 256–270, MIT Press, 1997.

[17] Douglas, S., and Dale, R. Towards Robust PATR. *Proc. COLING'92*, pp. 468–474, 1992.

[18] Frühwirth, T.W., Theory and Practice of Constraint Handling Rules, *Journal of Logic Programming*, Vol. 37(1–3), pp. 95–138, 1998.

[19] Heidorn, G.E., Jensen, K., Miller, L.A., Byrd, R.J., and Chodorow, M.S., The EPISTLE text-critiquing system. *IBM Systems Journal*, Vol. 21, no. 3, pp. 305–326, 1982.

[20] Mellish, C. S., Some chart-based techniques for parsing ill-formed input. *Proc. 27th Annual Meeting of the ACL*, pp. 102–109, 1989.

[21] Kakas, A.C., Michael, A., and Mourlas, C., ACLP: Abductive Constraint Logic Programming. *The Journal of Logic Programming*, Vol. 44, pp. 129–177, 2000.

[22] Richardson, S. and Braden-Harder, L., The experience of developing a large-scale natural language text processing system: CRITIQUE. *Proc. 2nd Conference on Applied Natural Language Processing and Industrial Applications*, pp. 195–202, 1988.

[23] Shieber, S., Uzkoreit, H., Pereira, F. , Robinson, J., and Tyson, M. The formalism and implementation of PATR-II, *SRI International*, pp. 39–79, 1983.

[24] Vosse, T., Detecting and correcting morpho-syntactic errors in real texts. *Proc. 3rd Conference on Applied Natural Language Processing*, pp. 111–118, 1992.