A Tool for Automated Revision of Grammars for NLP Systems

Nanda Kambhatla and Wlodek Zadrozny
IBM T.J. Watson Research Center
30 Saw Mill River Road,
Hawthorne, NY, 10532
{*nanda,wlodz*}@us.ibm.com

Abstract

We present an algorithm and a tool for automatically revising grammars for natural language processing (NLP) systems to disallow specifically identified sentences or sets of sentences. We also outline an approach for automatically revising attribute value grammars using counter-examples. Developing grammars for NLP systems that are both general enough to accept most sentences about a domain, but constrained enough to disallow other sentences is very tedious. Our approach of revising grammars automatically using counter-examples greatly simplifies the development and revision of tightly constrained grammars. We have successfully used our tool to constrain over-generalizing grammars of speech understanding systems and obtained higher recognition accuracy.

1 Introduction

Natural language processing systems often constrain the set of "utterances" from a user (spoken, typed in, etc.) to narrow down the possible syntactic and semantic resolutions of the utterance and reduce the number of misrecognitions and/or misunderstandings by the system. Such constraints on the allowed syntax and the inferred semantics are often expressed in the form of a "grammar"[1], a set of rules specifying the set of allowed utterances and possibly also specifying the semantics associated with these utterances. For instance, grammars are commonly used in speech understanding systems to specify both the set of allowed sentences and to specify "tags" to extract semantic entities (e.g. the "amount" of money).

Constraining the number of sentences accepted by a grammar is essential for reducing misinterpretations of user queries by an NLP system. For instance, for speech understanding systems, if the grammar accepts a large number of sentences, then the likelihood of recognizing uttered sentences as random, irrelevant, or undesirable sentences is increased. For transaction processing systems, misrecognized words can lead to unintended transactions being processed. An effective constraining grammar can reduce transactional errors by limiting the number of sentence level errors. The problem of over-generalization of speech grammars and related issues is well discussed by Seneff (1992).

Thus, speech grammars must often balance the conflicting requirements of

- accepting a wide variety of sentences to increase flexibility, and
- accepting a small number of sentences to increase system accuracy and robustness.

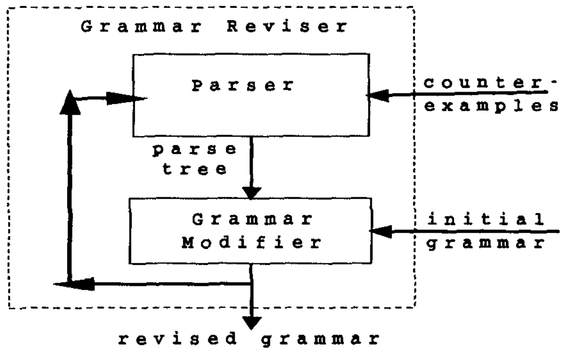
Developing tight grammars which trade-off these conflicting constraints is a tedious and

[1] Throughout this document, by using the word "grammar", we refer to a Context-Free Grammar that consists of a finite set of *non-terminals*, a finite set of *terminals*, a unique non-terminal called the *start symbol*, and a set of *production rules* of the form ***A->a***, where *A* is a non-terminal and ***a*** is a string of terminal or non-terminal symbols. The 'language' accepted by a grammar is the set of all terminal strings that can be generated from the start symbol by successive application of the production rules. The grammar may optionally have semantic interpretation rules associated with each production rule (e.g. see (Allen 95)).

difficult process. Typically, grammars overgeneralize and accept too many sentences that are irrelevant or undesirable for a given application. We call such sentences "counter-examples". The problem is usually handled by revising the grammar manually to disallow such counter-examples. For instance, the sentence "give me my last eighteen transactions" may need to be excluded from a grammar for a speech understanding system, since the words "eighteen" and "ATM" are easily confused by the speech recogniser. However, "five" and "ten" should remain as possible modifiers of "transactions". Counter-examples can also be sets of sentences that need to be excluded from a grammar (specified by allowing the inclusion of non-terminals in counter-examples). For example, for a banking application that disallows money transfers to online accounts, we might wish to exclude the set of sentences "transfer <AMOUNT> dollars to my online account" from the grammar, where <AMOUNT> is a non-terminal in the grammar that maps to all possible ways of specifying amounts.

In this paper, we are proposing techniques for automatically revising grammars using counter-examples. The grammar developer identifies counter-examples from among sentences (or sets of sentences) mis-recognized by the speech recognizer or from sentences randomly generated by a sentence generator using the original grammar. The grammar reviser modifies

Figure 1



the original grammar to invalidate the counter-examples. The revised grammar can be fed back to the grammar reviser and whole process can be iterated several times until the resulting grammar is deemed satisfactory.

In the next sections, we first describe our algorithm for revising grammars to disallow counter-examples. We also discuss algorithms to make the revised grammar compact using minimum description length (MDL) based grammar compaction techniques and extensions to our basic algorithm to handle grammars with recursion. We then present some results of applying our grammar reviser tool to constrain speech grammars of speech understanding systems. Finally, we present an approach for revising attribute value grammars using our technique and present our conclusions.

2 Automated Grammar Revision by rule modification

In this section, we describe an algorithm (see Figure 1) for revising grammars that directly modifies the rules of the grammar to disallow counter-examples. For each counter-example[2], we generate the parse tree (representation of all the grammar rules needed to generate the sentence or set of sentences) and the grammar modifier modifies the production rules of the grammar to invalidate the counter-example. This process is repeated for each counter-example using the revised grammar from the previous iteration for generating the parse tree for the current counter-example. If a counter-example generates multiple parse trees, the above algorithm is repeated for each parse tree in turn.

2.1 Grammar modification algorithm

We present the grammar modification algorithm below. For, we assume that the parse-tree(s) of the counter-example contain no recursion (i.e. the same production rule does not occur twice in any of the parse trees). In section 2.4, we present an approach for using the algorithm even when the parse-trees contain recursion. Thus, the algorithm is applicable for any context-free grammar. The grammar modification algorithm

[2] Note that a counter-example can be a sentence such as "move to operator" or a set of sentences such as "transfer <AMOUNT> to online account". The latter is specified using non-terminals interspersed with words.

for modifying the rules of a grammar to disallow a counter-example **c** (identified by a grammar developer) using a parse-tree for **c** proceeds as follows :

1. For each non-terminal <N> in the parse tree, except the <<START>> symbol,
   a. Add a rule to define a new non-terminal <N'> such that <N'> generates all phrases that <N> generates except for the phrase in the counter-example that <N> generates.
   b. Add a rule to define a new non-terminal <No> such that <No> generates only the phrase(s) in the counter-example that <N> generates.
2. Modify the rule that contains the <<START>> symbol in the parse tree, such that the <<START>> symbol no longer generates the given counter-example.

Figure 2

(a) Original grammar:

```
<<START>> ::= <V> <N> <PP> |
              <V> <PP> .
    <PP> ::= "to" <N> .
   <V> ::= "move" | "transfer" .
   <N> ::= "checking" | "savings" |
           "money" | "operator" .
```

(b) Parse Tree for "move to operator"

```
<<START>> ::= <V> <PP> .
      <V> ::= "move" .
     <PP> ::= "to" <N> .
      <N> ::= "operator".
```

We illustrate the algorithm with an example. Figure 2(a) shows a simple grammar. Suppose the sentence "move to operator" is a counter-example for an application. Figure 2(b) shows the parse-tree for "move to operator". Since the parse tree contains the rule: <V> ::= "move", new rules are added to define non-terminals <V'> and <Vo>, where <V'> does not generate "move" and <Vo> generates only "move". Similarly, since the parse tree contains the rule: <N>::= "operator", the new rules: <N'>::= "checking" | "savings" | "money"; and <No>::= "operator", are added. For the non-terminal <PP>, the new rules: <PP'>::= "to" <N'>; and <PPo>::= "to" <No>, are added. Note that since <No> only generates the phrase "operator" which is part of the counter-example, <PPo> only generates the phrase "to operator" which is part of the counter-example. Also, <PP'> generates all phrases that <PP> generates except for the phrase "to operator". Finally, the rule: <<START>>::= <V> <PP> is modified using the newly created non-terminals <V'>, <Vo>, <PP'> and <PPo> such that the only sentences which are accepted by the grammar and begin with the phrase "move" do not end with the phrase "to operator", and also, the only sentences which are accepted by the grammar and end with the phrase "to operator" do not begin with the phrase "move". Figure 3 shows the final modified grammar that accepts all the sentences that the grammar in Figure 2(a) accepts except for the sentence "move to operator". In Figure 3, all the grammar rules that are new or modified are shown in bold and italics.

Figure 3

```
<<START>> ::= <V> <N> <PP> |
              <V'> <PPo> |
              <Vo> <PP'> |
              <V'> <PP'> .
     <PP> ::= "to" <N> .
    <PP'> ::= "to" <N'> .
    <PPo> ::= "to" <No> .
      <V> ::= "move" | "transfer" .
     <V'> ::= "transfer" .
     <Vo> ::= "move" .
      <N> ::= "checking" | "savings" |
              "money" | "operator" .
     <N'> ::= "checking" | "savings" |
              "money" .
     <No> ::= "operator" .
```

The above algorithm for grammar modification has a time complexity of O(m*2^k) rule creation (or modification) steps for removing a counter-example, where m is the number of production rules in the parse tree of the counter-example and k is the largest number of non-terminals on the right hand side of any of these production rules. Since grammars used for real applications rarely have more than a handful of non-terminals on the right hand side of production rules, this complexity is quite manageable.

2.2 Automated grammar compaction using MDL based grammar induction

As seen in the example described above, the size of the grammar (number of production rules) can increase greatly by applying our algorithm successively for a number of counter-examples. However, we can remedy this by applying grammar induction algorithms based on minimum description length (MDL) (e.g. Grunwald (1996) and Zadrozny (1997)) to combine rules and create a compact grammar that accepts the same language.

The MDL principle (Rissanen (1982)) selects that description (theory) of data, which minimizes the sum of the length, in bits, of the description of the theory, and the length, in bits, of data when encoded using the theory. In our case, the data is the set of possible word combinations and the theory is the grammar that specifies it. We are primarily interested in using the MDL principle to obtain (select) a compact grammar (the theory) from among a set of equivalent grammars. Since the set of possible word combinations (data) is the same for all grammars in consideration, we focus on the description length of the grammars itself, which we approximate by using a set of heuristics described in step 1 below.

We use the following modified version of Zadrozny's (1997) algorithm to generate a more compact grammar from the revised grammar using the MDL principle:

1. Compute the description length of the grammar, i.e. the total number of symbols needed to specify the grammar, where each non-terminal, "::=", and " | " are counted as one symbol.
2. Modify the current grammar by concatenating all possible pairs of non-terminals, and compute the description length of each such resultant grammar. For concatenating <N1> and <N2>, introduce the rule <N3>::= <N1> <N2>, search all other rules for consecutive occurrences of <N1> and <N2>, and replace such occurrences with <N3>. Note that this change results in an equivalent grammar (that accepts the same set of sentences as the original grammar).
3. Modify the current grammar by merging all possible pairs of non-terminals, and compute the description length of each such resultant grammar. For merging <N4> and <N5>, introduce the rule: <N6>::= <N4> | <N5>, search for pairs of rules which differ only in one position such that for one of the rules, <N4> occurs in that position and the other rule, the <N5> occurs in the same position. Replace the pair of rules with a new rule that is exactly the same as either of the pairs of rules, except for the use of <N6> instead of <N3> or <N4>. Note that this change results in an equivalent grammar (that accepts the same set of sentences as the original grammar).
4. Compute a table of description lengths of the grammars obtained by concatenating or merging all possible pairs of non- terminals of the initial grammar, as described above. Select the pair of non-terminals (if any) together with the action (concatenate or merge) that results in the least description length and execute the corresponding action.
5. Iterate steps 2, 3, and 4 until the description length does not decrease. No further modification is performed if the base description length of the grammar is lower than that resulting from merging or concatenating any pair of non- terminals.

In variations of this algorithm, the selection of the pairs of non-terminals to concatenate or merge, can be based on; the syntactic categories of the corresponding terminals, the semantic categories of the corresponding terminals, and the frequency of occurrence of the non-terminals.

Using the algorithm described above in conjunction with the algorithm in section 2.1, we can obtain a compact grammar that is guaranteed to disallow the counter-examples.

2.3 Results for grammar revision for speech understanding systems

We have built a graphical tool for revising grammars for NLP systems based on the algorithm described in sections 2.1 and 2.2 above. The tool takes as input an existing grammar and can randomly generate sentences accepted by the grammar including non-terminal strings and strings containing terminals and non-terminals (e.g. both "move to operator" and "transfer <AMOUNT> to online account" would be generated if they were accepted by the grammar). A grammar developer (a human) interacts with the tool and either inputs counter-examples selected from speech recognition error logs or selects counter-examples like the ones listed above. The grammar developer can then revise the grammar to disallow the counter-examples by pressing a button and then reduce the size of the resulting grammar using the algorithm in section 2.2 by pressing another button to obtain a compact grammar that does not accept any of the identified counter-examples. Typically, the grammar developer repeats the above cycle several times to obtain a tightly constrained grammar.

We have successfully used the tool described above to greatly constrain overgeneralizing grammars for speech understanding systems that we built for telephony banking, stock trading and directory assistance (Zadrozny *et al*, 1998). The speech recognition grammars for these systems accepted around fifty million sentences each. We successfully used the reviser tool to constrain these grammars by eliminating thousands of sentences and obtained around 20-30% improvement in sentence recognition accuracy. We conducted two user studies of our telephony banking system at different stages of development. The user studies were conducted eight months apart. During these eight months, we used a multi-pronged strategy of constraining grammars using the grammar revision algorithms described in this paper, improving the pronunciation models of some words and redesigning the prompts of the system to enable fast and easy error recovery by users. The combination of all these techniques resulted in improving the 'successful transaction in first try'[3] from 43% to 71%, an improvement of 65%. The average number of wrong tries (turns of conversation) to get a successful answer was reduced from 2.1 to 0.5 tries. We did not conduct experiments to isolate the contribution of each factor towards this improvement in system performance.

It is important to note here that we would probably have obtained this improvement in recognition accuracy even with a manual revision of the grammars. However, the main advantage in using our tool is the tremendous simplification of the whole process of revision for a grammar developer who now selects counter-examples with an interactive tool instead of manually revising the grammars.

2.4 Handling recursion in grammars

We now describe an extension of the algorithm in section 2.1 that can modify grammars with recursion to disallow a finite set of counter-examples. The example grammars shown above are regular grammars (i.e. equivalent finite state automatons exist). For regular grammars (and only for regular grammars), an alternative approach for eliminating counter-examples using standard automata theory is:

- Compute the finite state automaton (FSA) G corresponding to the original grammar.
- Compute the FSA C corresponding to the set of counter-examples.
- Compute C', the complement of C with respect to the given alphabet.
- Compute G', the intersection of G and C'. The FSA G' is equivalent to a revised grammar which disallows the counter-examples.

[3] We measured the number of times the user's transactional intent (e.g. checking balance, last five transactions etc.) was recognized and acted upon correctly by the system in the first try, even when the actual utterance may not have been recognized correctly word for word.

The time complexity of the algorithm is O(n*m), where n and m are the number of states in the finite state automatons G and C respectively. This is comparable to the quadratic time complexity of our grammar revision algorithm presented in Section 3.1.

However, the above algorithm for eliminating counter-examples only works for regular grammars. This is because context-free grammars are not closed under complementation and intersection. However we can use our algorithm for grammar modification (section 2.1) to handle any context-free grammar as follows:

1) As before, generate parse tree p for counter-example c for an initial grammar G.
2) If p contains a recursion (two or more repetitions of any production rule in the same parse tree), rewrite the initial grammar G as the equivalent grammar G', where the recursion is "unrolled" sufficiently many times (at least one more time than the number of repetitions of the recursive production rule in the parse tree). We explain the unrolling of recursion in greater detail below. If p does not contain any recursion, go to step 4.
3) Generate parse tree p' for the counter-example c for the rewritten grammar G'. Note that p' will no longer contain a recursive application of any production rules, though G' itself will still have recursion.
4) Use the algorithm described in section 2.1 to modify the grammar G' to eliminate the counter-example c using the parse tree p'.

We illustrate the above algorithm with an example. Figure 4(a) shows a context free grammar which accepts all strings of the form $a^n b^n$, for any n greater than 0. Note that this is not a regular language. Suppose we wish to eliminate the counter-example ***aaabbb*** from the initial grammar. The parse tree ***p*** for the counter-example ***aaabbb*** is shown in Figure 4(b). The grammar in 4(a) can be rewritten as the equivalent grammar 4(c), where the recursion of (***S->aSb***) is unrolled three times. The parse tree ***p'*** for the counter-example aaabbb with respect to grammar in 4(c) is shown in Figure 4(d). Note that ***p'*** does not contain any recursion, though the rewritten grammar does. We revised the

FIGURE 4

(a) ORIGINAL GRAMMAR G
```
<S> ::= "a" <S> "b" | "a" "b" .
```
(b) PARSE TREE p
```
<S> ::= "a" <S> "b" .
<S> ::= "a" <S> "b" .
<S> ::= "a" "b" .
```
(c) REWRITTEN GRAMMAR G'
```
<S>  ::= "a" <S1> "b" | "a" "b" .
<S1> ::= "a" <S2> "b" | "a" "b" .
<S2> ::= "a" <S3> "b" | "a" "b" .
<S3> ::= "a" <S3> "b" | "a" "b" .
```
(d) PARSE TREE p'
```
<S>  ::= "a" <S1> "b" .
<S1> ::= "a" <S2> "b" .
<S2>  ::= "a" "b" .
```
(e) REVISED GRAMMAR Gr
```
<S>  ::= "a" <S1> "b" | "a" "b" .
<S1> ::= "a" <S2> "b" | "a" "b" .
<S2> ::= "a" <S3> "b" .
<S3> ::= "a" <S3> "b" | "a" "b" .
```

grammar in 4(c) to eliminate the counter-example ***aaabbb*** using the parse tree in Figure 4(d). The revised grammar is shown in Figure 4(e). Note that here we are assuming that a mechanism exists for rewriting the rules of a grammar with recursion to unroll the recursion (if it exists) a finite number of times. Such an unrolling is readily accomplished by introducing a set of new non-terminal's, one for each iteration of unrolling as shown in Figure 4(c).

3 Automated revision of attribute-value grammars

In this section, we delineate an approach for automatically modifying attribute value grammars using counter-examples. We first convert an attribute value grammar into an equivalent non-attributed grammar by creating new non-terminals and encoding the attributes in the names of the new non-terminals (see Manaster Ramer and Zadrozny (1990) and Pollard and Sag (1994)).

For example, suppose the grammar in Figure 2(a) is an attribute value grammar with an

Figure 5

```
<<START>> ::= <V> <N> <PP> |
              <V> <PP> .
     <PP> ::= "to" <N> .
      <V> ::= "move" | "transfer" .
      <N> ::= <N_account_checking> |
              <N_account_savings> |
              <N_account_unspecified> .

  <N_account_checking> ::= "checking" .
   <N_account_savings> ::= "savings".
<N_account_unspecified> ::= "money" |
                            "operator" .
```

attribute 'account', which encodes information about the type of account specified, e.g. 'account' might have the values, SAVINGS, CHECKING and UNSPECIFIED. Figure 5 shows an equivalent non-attributed grammar, where the value of the attribute 'account' has been encoded in the names of the non-terminals. Note that such an encoding can potentially create a very large number of non-terminals. Also, the specific coding used needs to be such

Figure 6

```
               <<START>> ::= <V> <N> <PP> | <V'> <PPo> |
                             <Vo> <PP'> | <V'> <PP'> .
                    <PP> ::= "to" <N> .
                   <PP'> ::= "to" <N'> .
                   <PPo> ::= "to" <No> .
                     <V> ::= "move" | "transfer" .
                    <V'> ::= "transfer" .
                    <Vo> ::= "move" .
                     <N> ::= <N_account_checking> |
                             <N_account_savings> |
                             <N_account_unspecified> .
                    <N'> ::= <N_account_checking> |
                             <N_account_savings> |
                             <N'_account_unspecified> .
                    <No> ::= <No_account_unspecified> .
    <N_account_checking> ::= "checking" .
     <N_account_savings> ::= "savings".
 <N_account_unspecified> ::= "money" |
                             "operator" .
<N'_account_unspecified> ::= "money" .
<No_account_unspecified> ::= "operator" .
```

that the attributes can be easily recovered from the non-terminal names later on.

We can now use our modification algorithms (Section 2.1 and 2.2) to eliminate counter-examples from the non-attributed grammar. For instance, suppose we wish to eliminate 'move to operator' from the attributed grammar based on Figure 2(a), as discussed above. We apply our algorithm (Section 2.1) to the grammar in Figure 5 and obtain the grammar shown in Figure 6. Note that we name any new non-terminals created during the grammar modification in such a way as to leave the encoding of the attribute values in the non-terminal names intact.

After applying the grammar revision algorithm, we can extract the attribute values from the encoding in the non-terminal names. For instance, in the example outlined above, we might systematically check for suffixes of a certain type and recover the attributes and their values. Also, as described earlier, we can use the algorithm described in section 2.2 to make the resulting grammar compact again by using MDL based grammar induction algorithms.

4 Conclusions

We have presented a set of algorithms and an interactive tool for automatically revising grammars of NLP systems to disallow identified counter-examples (sentences or sets of sentences accepted by the current grammar but deemed to be irrelevant for a given application). We have successfully used the tool to constrain overgeneralizing grammars of speech understanding systems and obtained 20-30% higher recognition accuracy. However, we believe the primary benefit of using our tool is the tremendously reduced effort for the grammar developer. Our technique relieves the grammar developer from the burden of going through the tedious and time consuming task of revising grammars by manually modifying production rules one at a time. Instead, the grammar developer simply identifies counter-examples to an interactive tool that revises the grammar to invalidate the identified sentences.

We also discussed an MDL based algorithm for grammar compaction to reduce the size of the revised grammar. Thus, using a combination of the algorithms presented in this paper, one can obtain a compact grammar that is guaranteed to disallow the counter-examples.

Although our discussion here was focussed on speech understanding applications, the algorithms and the tool described here are applicable for any domain where grammars are used. We are currently implementing an extension of the grammar modifier to handle attribute-value grammars. We outlined an approach for automated modification of attribute-value grammars in Section 3.

We conclude that algorithms for automatically constraining grammars based on counter-examples can be highly effective in reducing the burden on grammar developers to develop constrained, domain specific grammars. Moreover, these algorithms can be used in any applications, which deal with grammars.

Acknowledgements

We thank all of our colleagues in the conversation machines group at IBM T.J. Watson Research Center for several helpful comments and suggestions through the course of this work.

References

Zadrozny W., Wolf C., Kambhatla N., and Ye Y. (1998). *Conversation machines for transaction processing*. In proceedings of AAAI'98/IAAI'98, AAAI Press/MIT Press, pp 1160-1166.

Allen J. (1995). *Natural Language Understanding.* The Benjamin/Cummings Publishing Company, Redwood City, CA 94065.

Grunwald P. (1996). *A minimum description length approach to grammar inference.* In S. Wemter et al., editors, Symbolic, Connectionist and Statistical Approach to Learning for Natural Language Processing, Springer, Berlin, p 203-216.

Manaster-Ramer and Zadrozny W. (1990), *Expressive Power of Grammatical Formalisms,* Proceedings of Coling-90. Universitas Helsingiensis. Helsinki, Finland", pp.195-200.

Pollard, C. and Sag I A. (1994). *Head-Driven Phrase Structure Grammar.* The U. of Chicago Press.

Rissanen J. (1982). *A universal prior for integers and estimation by minimum description length.* Annals of Statistics, 11:416-431.

Seneff S. (1992). TINA: *A natural language system for spoken language applications*, Computational Linguistics, 18:p61-86.

Zadrozny W. (1997). *Minimum description length and compositionality.* Proceedings of Second International Workshop for Computational Semantics, Tilburg. Recently re-published as a book chapter in: H.Bunt and R.Muskens (eds.) *Computing Meaning.* Kluwer Academic Publishers, Dordrecht/Boston, 1999.