Compilation of Constraint-based Contextual Rules for Part-of-Speech Tagging into Finite State Transducers*

Jorge Graña[1], Gloria Andrade[1], and Jesús Vilares[2]

[1] Departamento de Computación, Universidad de La Coruña
Campus de Elviña s/n, 15071 - La Coruña, Spain
{grana,andrade}@dc.fi.udc.es
[2] Departamento de Informática, Universidad de Vigo
Campus de As Lagoas, 32004 - Orense, Spain
jvilares@uvigo.es

Abstract. With the aim of removing the residuary errors made by pure stochastic disambiguation models, we put forward a hybrid system in which linguist users introduce high level contextual rules to be applied in combination with a tagger based on a Hidden Markov Model. The design of these rules is inspired in the Constraint Grammars formalism. In the present work, we review this formalism in order to propose a more intuitive syntax and semantics for rules, and we develop a strategy to compile the rules under the form of Finite State Transducers, thus guaranteeing an efficient execution framework.

1 Introduction

In the context of the use of the tools for *part-of-speech tagging* (POST) developed in the GALENA and CORGA projects[3], our projects for the automatic processing of the Spanish and Galician languages, a repeated request from the linguist partners has been the design of a formalism to introduce high level contextual rules. The purpose of these rules is to remove the residuary errors that pure stochastic taggers systematically make, allowing them to improve the usual performances of 95-97% of success.

The *constraint grammars* formalism (CGs) [3] was a good candidate on the basis of its good results: on concrete languages, such as English [7], performances can reach 99% of precision with a set of about 1 000 contextual rules; and, in general, performances are better than those of the pure stochastic taggers particularly when training and application texts are not from the same style and

* This work has been partially supported by the Spanish Government (under projects TIC2000-0370-C02-01 and HP2001-0044), and by the Galician Government (under project PGIDT01PXI10506PN).

[3] GALENA means *Generation of Natural Language Analyzers* and CORGA means *Reference Corpus of Current Galician*. See http://coleweb.dc.fi.udc.es for more information on both projects.

source. However, comparison is difficult since some ambiguities are not resolved by CGs. That is, CGs return a set of more than one tag in some cases, which makes it necessary to combine this technique with another one.

Furthermore, the syntax and semantics of the rules involved in CGs are not very intuitive, since they try to solve problems other than tagging. In fact, they join tagging and certain steps of parsing, even though both analyses are traditionally treated in separate modules. These aspects definitively lead us to design a new formalism for contextual rules. Furthermore, these rules will be oriented to operate together with our tagger, a second order *hidden Markov model* (HMM) with linear interpolation of uni-, bi-, and trigrams as smoothing technique [2].

The aim of the present work is to describe the new formalism of contextual rules. This formalism is mainly inspired in CGs, but some aspects from other rule-based environments, such as *transformation-based error-driven learning* [1] or *relaxation labelling* [5], have been also considered. After this, we focus the discussion on the efficient execution of the rules, for which we design a strategy that compiles them into *finite state transducers* (FSTs). Finally, we make some reflections about time and space complexity of the FSTs obtained.

2 Contextual Rules based on Constraints

The heterogeneous nature of languages for building contextual rules is a current problem: there is no standard, and there are many differences between languages. Basically, all systems use rules of this kind: if a certain *constraint* or condition is satisfied (for instance, the word following the current word is a verb), then a concrete *action* is executed (for instance, to select the tag substantive). Actions are usually limited to selection or deletion of one of the possible tags. However, constraints involved in the rules can vary greatly in the type of the condition and in the syntax used.

Our formalism omits syntactic components from the rules, represents the same range of conditions as that covered by the rule-based environments cited above, but with a more intuitive and legible syntax, and allows us to express preceding and following contexts on the basis of both tags and words, whether those words are ambiguous or not. The rules can be classified in three different groups: local rules, barrier rules and special rules.

2.1 Local Rules

The rules that conform this group present the following syntax:

```
<action> (<tag>) ([not] <position> <condition> {<values>}, ...);
```

In `<action>`, we specify the mission of the rule. The possible actions to execute are:

- `select`, to select one of the possible tags of the current word,

- delete, to remove one of those tags,
- and force, to fix the tag for the current word, even though this tag is not present in the set of possible tags for that word.

The field <position> must be replaced by an integer indicating which is the word affected by the condition of the rule: 0 for the current word, -1 for the previous one, 1 for the next one, and so on. The possible values for <condition> and their corresponding semantics are:

- is, which is true when the set of possible tags of the word affected by the condition and the set of tags <values> specified in the rule are equal,
- contains, which is true when the set of possible tags of the word affected by the condition contains the set of tags <values> specified in the rule,
- and belongs, which is true when the word affected by the condition is present in the set of words <values> specified in the rule.

Conditions can be combined with the usual logic connectives:

- for negation, we can use the keyword not in front of the condition,
- for logic-and, we can write several conditions delimited by commas in the same rule,
- and for logic-or, we can write the conditions in separate rules, all these rules involving the same <action> and <tag> fields.

The following example illustrates the aspect presented by the rules included in this group[4]:

```
force (Det) (1 belongs {sobre}, not 1 is {P});
```

This rule fixes determiner as the tag of the current word, when the next one is the word sobre and is not a preposition.

2.2 Barrier Rules

These rules do not refer a concrete position, but navigate leftwards or rightwards from the current word, by replacing the field <direction> by -* or +*, respectively, in the following syntax:

```
<action> (<tag>) (<direction> <condition-1> {<values-1>}
                        barrier <condition-2> {<values-2>}, ...);
```

The general constraint of the rule is established by the condition <condition-1> expressed before the keyword barrier, and the boundary for navigation is established by <condition-2>. This allows us to express situations like the following:

```
select (Adj) (-* contains {S} barrier is {V});
```

where if before the current word there is another word that can be a substantive, and between those two words there is no verb, the current word will be an adjective.

[4] To simplify, in this work, we use Adj for adjective, Det for determiner, P for preposition, Pron for pronoun, S for substantive, and V for verb.

2.3 Special Rules

Finally, we have a group for special rules, that always execute an action. Their syntax is:

```
<action> always (<tag>);
```

In this case, the possible actions are only `select` and `delete`, because the action `force` would produce an output text with all the words tagged with the same tag <tag>. An example could be the following:

```
delete always (ForeignWord);
```

When it is sure that the input text to be processed is written only in the local language, we can use this rule to remove the tag `ForeignWord` from all ambiguous words that can be a foreign word and something else. These rules could be called also *rare rules*, since they are not commonly used.

2.4 A Practical Example

In Table 1, we show how an initial trellis with ambiguous words evolves step by step when a certain set of contextual rules is applied on it. We have marked with boxes the points where individual rules match. The last row contains the final trellis obtained after the application of the whole set of rules.

Rule	Trellis					
`select (S)` `(1 is {V});`	El Pron	sobre P $\boxed{S}$ V	está V	sobre P S V	la Det Pron	mesa S V
`delete (V)` `(-2 contains {P,V});`	El Pron	sobre S	está V	sobre P S V	la Det Pron	mesa S $\boxed{V}$
`select (P)` `(-2 belongs {bajo,sobre}, -1 is {V});`	El Pron	sobre S	está V	sobre $\boxed{P}$ S V	la Det Pron	mesa S
`force (Det)` `(1 belongs {sobre}, not 1 is {P});`	El $\boxed{Pron}$	sobre S	está V	sobre P	la Det Pron	mesa S
	El Det	sobre S	está V	sobre P	la Det Pron	mesa S

Table 1. Evolution of a trellis when a set of contextual rules is applied on it

The correct sense of this sentence is the following: *The envelope is on the table*. Therefore, we can observe in the final trellis that some tagging conflicts have been solved. However, the trellis still contains ambiguities. This behaviour is normal, since the application of contextual rules does not guarantee the removal of all ambiguities; but now we could apply the Viterbi algorithm [6] with a greater possibility of success. It is necessary to remember that we are in the context of a hybrid system for POST, and rules operate together with a HMM-based tagger.

3 Compilation of Contextual Rules into FSTs

Instead of processing the input stream forward and backward repeatedly, looking for the matching of the conditions involved in the rules, it is advisable to perform a previous compilation step that translates the rules into finite state transducers. These structures always treat the input stream linearly forward and allow us to apply contextual rules more efficiently.

3.1 Definition of the Input Alphabet

In order to work with FSTs, we need an alphabet, i.e. the set of symbols that conform the input streams. Since rules will be applied on trellises, in which ambiguous words can appear, a first alternative was to use pairs as elements of the alphabet. In these pairs, the first component would be a word, and the second one would be the set of tags associated to that word in the trellis. However, it would need to perform operations on sets in order to transit adequately between the states of the FSTs, and those operations could involve too high a computational effort for every single transition. Therefore, this approach was rejected.

A better mechanism for the symbolic compilations of the rules is the assignment of integers to words and tags. Furthermore, in order to obtain FSTs which are as compact as possible, we do not consider the whole tagset and the whole dictionary of our system, but only the set of tags and the set of words that appear in the rules (we will call these sets TR and WR, respectively). In this way, the exact correspondence is the one shown in Table 2: 0 is reserved for the empty string ε; -1 and 1 will be the beginning and the end of sentence, respectively; the tags that appear in the rules will be numbered forward from 3 to n, and stored in a structure that we will call *mini-tagset*; the words that appear in the rules will be numbered backward from -3 to $-m$, and stored in a structure that we will call *mini-dictionary*; and finally, -2 and 2 will be reserved as wildcards to represent any other words or tags, respectively, that do not appear in the rules, but can appear in a new sentence to be tagged and in its corresponding trellis.

$w \in WR$	$w \notin WR$	Start	ε	End	$t \notin TR$	$t \in TR$
$-m, \ldots, -4, -3$	-2	-1	0	1	2	$3, 4, \ldots, n$

Table 2. Mapping for words and tags that can appear in trellises

`Mini-tagset`	`Mini-dictionary`
`Det` → `3`	`bajo` → `-3`
`P` → `4`	`sobre` → `-4`
`S` → `5`	
`V` → `6`	

Table 3. A mini-tagset and a mini-dictionary

The set of rules used in the example of Table 1 involves four tags and two words, so we can build the mini-tagset and mini-dictionary shown in Table 3. By using these structures and the mapping explained above, we can transform the initial trellis of that example into the one shown in Table 4.

El	sobre	está	sobre	la	mesa	-2	-4	-2	-4	-2	-2
Pron	P	V	P	Det	S	2	4	6	4	2	5
	S		S	Pron	V		5		5	3	6
	V		V				6		6		

Table 4. A trellis and its corresponding mapping

Now, we collect all the integers of this new trellis by columns and add the marks of beginning and end of sentence in order to obtain the corresponding input stream for a cascade of FSTs implementing the contextual rules:

```
-1 -2  2 -4  4  5  6 -2  6 -4  4  5  6 -2  2  3 -2  5  6  1
```

The FSTs will translate this input stream into an output stream with the same aspect. After this, we apply the inverse procedure to retrieve the final trellis obtained from the application of the contextual rules.

3.2 Building an FST for Every Contextual Rule

The next step consists in the construction of an individual FST for every contextual rule to be applied. Firstly, we describe this process intuitively by using the following rule:

```
select (S) (1 is {V});
```

A possible version of the corresponding FST is shown in Fig. 1. In this case, the action must be represented first (from state 0 to state 2), since the condition (from state 2 to 5) affects a word that appears afterwards. In both cases, action and condition, the word must be represented before its possible tags. We will draw arcs with thick lines for words and arcs with normal lines for tags.

– **Action**. In this rule, the current word can be any word, hence we use the wildcard -2 and all the words present in the mini-dictionary (`bajo` and `sobre`) in the label of the arc from state 0 to 1. In a trellis, only tags can be modified, not words. This is the reason why these three possible cases for words are translated into themselves. The action of the rule, i.e. to select the tag substantive, is represented by the arc $1 \xrightarrow{\mathtt{S:S}} 2$, which does not modify the tag, and by the loops in states 1 and 2, which remove the rest of possible tags by translating them into the empty string 0. We use two loops instead of only one because we assume that the tags will always appear in a predefined order (usually, the lexicographic order, which should coincide with the order defined in the mapping), and that this order will not be broken by any sequence of iterations with different tags in the same loop. These hypotheses are not critical in practice, and allow us to build more compact FSTs. Be that as it may, the most important aspect is that an arc keeping the tag substantive is mandatory and cannot be included in any optional loop.

– **Condition**. The same reasoning made above is applicable to the word involved in the condition (the arc from state 2 to 3). The rule requires this word to be precisely a verb, hence the presence of only one arc for its tags ($3 \xrightarrow{\texttt{V:V}} 4$, which does not modify the tag since we are implementing a condition, not an action). This is enough for the example under consideration, since verb is the last tag in the mini-tagset. However, it is a better alternative not to allow state 4 to be the final state of this FST, because in general it cannot be guaranteed that verb is the unique tag. Therefore, another word or the end of sentence must appear after the tag (hence the final arc from state 4 to 5).

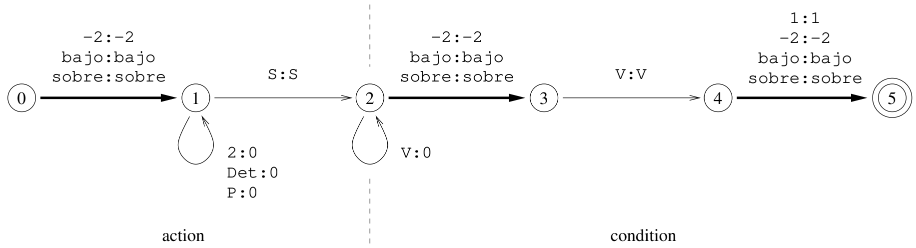

Fig. 1. Fictitious FST for the rule `select (S) (1 is {V});`

The FST presented above is fictitious and has been shown only to facilitate understanding. In practice, words and tags must be replaced by their integers associated in the mapping, producing the corresponding real implementation of the FST shown in Fig. 2.

The designing and building of FSTs becomes more difficult when the condition is complex. For instance, the construction of transducers for barrier rules or for local rules with inclusions or negations is much more complicated than for the simple equality shown above. Nevertheless, this process of symbolic compilation has been implemented for all the rule schemes covered by our formalism. Since it is not possible to describe the whole compilation process here, we include only one more example in order to illustrate how FSTs always prove to be a robust frame to perform any operation involved in this kind of contextual rules. In this

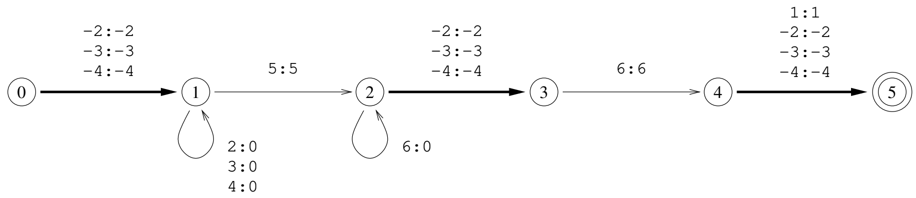

Fig. 2. Real FST for the rule `select (S) (1 is {V});`

case, we consider the generic barrier rule

$$\texttt{select}\ (t_s)\ (+*\ \texttt{is}\ \{t_j, t_k, \ldots, t_l\}\ \texttt{barrier is}\ \{t_x, t_y, \ldots, t_z\})$$

The aspect of the abstract version of the corresponding FST is shown in Fig. 3.

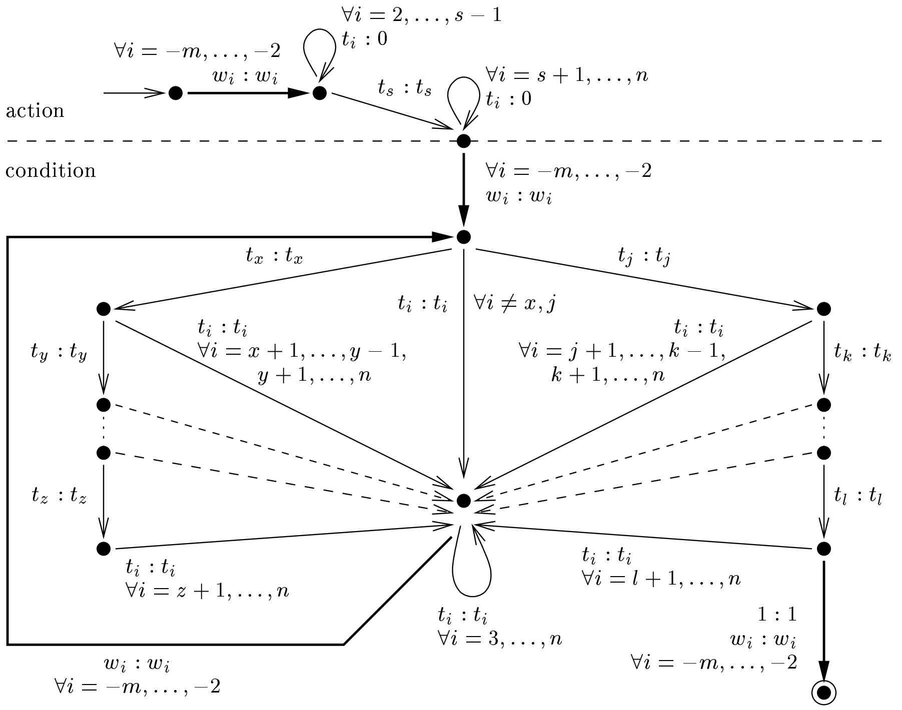

Fig. 3. Generic FST for the barrier rule $\texttt{select}\ (t_s)\ (+*\ \texttt{is}\ \{t_j, t_k, \ldots, t_l\}\ \texttt{barrier is}\ \{t_x, t_y, \ldots, t_z\})$

3.3 Making the FSTs Operative

Let us consider, for instance, a simple transducer which transforms `3` into `4` when `1` appears before the `3`. This FST will produce the output stream `1 4` only if the input stream is exactly `1 3`. However, we want this FST to produce the same effect for any other sequences of symbols containing `1 3` as substring. In order to do this, we need to apply a normalization process on the FST. That is, once we have built a transducer for every contextual rule, the next phase is to normalize all these FSTs. The normalization process requires the following steps:

- Compilation and closure of the alphabet. Once we have defined the input alphabet, which we will call Σ, it is necessary to create a FST to transform every symbol into itself. Then, we obtain Σ^*, where $*$ is the closure operation, in order to be able to process input strings with more than one symbol.
- For every transducer T, we perform the following operations:
  - $a = p_1(T)$, where p_1 is the first project operation, i.e. every arc $q \xrightarrow{\mathtt{x:y}} r$ is replaced by $q \xrightarrow{\mathtt{x}} r$ or by $q \xrightarrow{\mathtt{x:x}} r$. Therefore, a can be seen as an automaton or as a transducer that does not change its input.
  - $\Gamma = \Sigma^* - (\Sigma^* \cdot a \cdot \Sigma^*)$, where $-$ is the subtraction operation and $\cdot$ is the concatenation operation. In this way, we obtain in Γ a transducer that accepts the complement of the language accepted by a (and by T).
  - Finally, $N = \Gamma \cdot (T \cdot \Gamma)^*$, in order to obtain in N the normalized version of the transducer T.

Once we have normalized all transducers, we can form a cascade by applying them in sequence, taking the output of an FST as input for the following one. Another option is to build only one FST able to simulate the simultaneous application of all contextual rules by only one pass over the input stream. This FST can be obtained by composing all individual transducers.

The operations of closure, project, subtraction, concatenation, composition, etc., are well-defined in the theory of FSTs, and there are tools available which implement all these procedures. In our case, we have used the free version of *FSM Library: General-purpose finite-state machine software tools*, available in `http://www.research.att.com/sw/tools/fsm/`, implemented by Mohri, Pereira and Riley from the *AT&T* laboratories [4]. This library has allowed us to build the final operative version of our FSTs.

3.4 Space and Time Complexities

The FSTs obtained have always operated correctly and rapidly, which shows that the general compilation procedure is robust. For instance, the spaces and times consumed by the FSTs corresponding to the rules of Table 1 appear in Table 5, where T_i is the FST of each individual rule, and T_{1234} is the composition of the four preceding FSTs. It is important to note that, in theory, the time complexity of the FST obtained by composition is linear respect to the length of the input, and does not depend on the number of rules. But in practice, even though in a hybrid system for POST a small set of contextual rules (between 50 and 100) is expected to be sufficient, the computational cost of the composition operation is very high, and it is more advisable to perform a sequential application of the rules by executing the cascade of the corresponding individual FSTs: for time, 0.247 vs. 1.717 seconds, in a Pentium III 450 MHz. under Linux operating system; and for space, 20 160 vs. 2 931 156 bytes, in all cases with only 4 contextual rules.

FSTs	Initial FST			Normalized FST				
	number of states	number of arcs	size in bytes	number of states	number of arcs	size in bytes	compilation in seconds	execution in seconds
T_1	6	16	348	19	142	2 520	0.245	0.051
T_2	11	32	664	58	520	9 036	0.878	0.072
T_3	8	26	532	39	346	6 024	0.586	0.070
T_4	7	23	472	20	145	2 580	0.251	0.054
Total	32	97	2 016	136	1 153	20 160	1.960	0.247
T_{1234}	-	-	-	30 244	160 513	2 931 156	11.773	1.717

Table 5. Spaces and times for a set of FSTs and for their corresponding composition

4 Conclusion and Future Work

We have presented a strategy to compile constraint-based contextual rules for part-of-speech tagging under the form of finite state transducers. The purpose of these contextual rules is to remove the residuary errors that pure stochastic taggers make. These contextual rules must be introduced by linguists, and therefore we also provide a new formalism with intuitive syntax and semantics. Transducers have proved to be a very formal and robust execution framework for contextual rules, but there are still some aspects that should be investigated further. In the context of our hybrid system for POST, where contextual rules operate in combination with a HMM-based tagger, it is necessary to perform experiments with the purpose of selecting the best order in which to apply these two disambiguation techniques. Another aspect of future work, in this case for situations where it is possible to detect that residuary errors are systematic, is the automatic generation of specific contextual rules for these errors.

References

1. Brill, E. (1994). Some advances in rule-based part of speech tagging. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*.
2. Graña, J.; Chappelier, J.-C.; Vilares, M. (2001). Integrating external dictionaries into part-of-speech taggers. In *Proc. of the Euroconference on Recent Advances in Natural Language Processing (RANLP-2001)*, pp. 122-128.
3. Karlsson, F.; Voutilainen, A.; Heikkilä, J.; Anttila, A. (1995). Constraint grammar: a language-independent system for parsing unrestricted text. *Mouton de Gruyer*, Berlin.
4. Mohri, M. (1997). Finite-state transducers in language and speech processing. *Computational Linguistics*, vol. 23(2), pp. 269-311.
5. Padró, L. (1996). POS tagging using relaxation labelling. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*.
6. Viterbi, A.J. (1967). Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Trans. Information Theory*, vol. IT-13 (April).
7. Voutilainen, A.; Heikkilä, J. (1994). An English constraint grammar (EngCG): a surface-syntactic parser of English. In Fries, Tottie and Schneider (eds.), *Creating and using English language corpora*, Rodopi.