Specifying Properties of a Language with Regular Expressions

François Trouilleux

To cite this version:

François Trouilleux. Specifying Properties of a Language with Regular Expressions. Recent Advances in Natural Language Processing, 2007, Bulgaria. pp.1. hal-00373340

HAL Id: hal-00373340
https://hal.science/hal-00373340

Submitted on 3 Apr 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Specifying Properties of a Language with Regular Expressions

François Trouilleux
Laboratoire de recherche sur le langage
Université Blaise-Pascal
29, boulevard Gergovia
63037 Clermont-Ferrand cedex
trouilleux@lrl.univ-bpclermont.fr

Abstract

This article presents a translation of the Property formalism of [2] into the XFST regular expression formalism [6]. Besides offering at no cost a platform to use Properties in natural language processing, this operation allows us to clarify the interpretation of the Property formalism, which may be interpreted as strictly limited either to regular languages or to context-free languages, depending on the definition of the objects Properties apply to.

Keywords

Properties, Regular expressions, Phrase structure grammars, Constraints, Xerox Finite-State Tool (XFST)

1 Introduction

In 1999, Gabriel G. Bès proposed a new formalism for the description of natural language syntax, called "Properties" [2], of which Blache afterwards proposed a variant, called "Property Grammar" [3]. The purpose of this paper is to offer a new point of view on this formalism, through its translation into a regular expression formalism, that of the Xerox Finite-State Tool [6]. This translation, developed in section 2, supports both theoretical and practical considerations: section 3 points out the specificity of the property formalism with respect to classical phrase structure grammars, and section 4 clarifies its interpretation in terms of regular or context-free language specification. Practical use of our translation scheme and relevance of the Property formalism are then discussed in section 5.

2 Properties as regular expressions

This section introduces a translation of Properties into regular expressions (2.1), followed by a discussion of some limitations of this translation (2.2).

2.1 A translation scheme

Properties are formulas of the form `pred(id, ...)`, where `pred` is a predicate corresponding to the name of the Property, `id` is the name of the language (or category) the Property applies to, and `...` marks the place of one or several other arguments of the Property predicate `pred` (cf. [2]). These arguments are always symbols which refer to a category.

[2] specifies nine types of properties. Figure 1 presents the definition of six of them: *amod*, *uniq*, *oblig*, *exig*, *exclu* and *precede*, together with their translation into XFST regular expressions (below the dotted line) [1]. Limitations of our translation scheme, including the non-translation of some Properties, are discussed in the next section.

Full definition of the operators used in the regular expressions may be found in [1], as well as on the XRCE web site [2]. In short, `|` denotes union, `&` intersection, `<` precedence, `+` iteration (Kleene plus), `$` containment, `$?` containment of at most one, and `~` complement.

In figure 1, we use letters as arguments of the Property predicates. These letters are to be interpreted as category names, *i.e.* as denoting regular languages. However, to keep the description short, we use expressions such as "an `a`" to mean "a string of category `a`" or "a string from language `a`".

All Properties are presented as applying to a language called `id`. On the regular expression side, this language is defined as the result of the intersection of all the languages denoted by the Properties. Considering the Properties defined in figure 1, one would then have to write as a final definition:

```
define id [AMOD & UNIQ & OBLIG
          & EXIG & EXCLU & PRECEDE];
```

2.2 Limitations

In our translation scheme, we made a number of simplifications on the original definitions of [2], which we briefly justify here.

For space reasons, we do not present the translation of some parts of the formalism, *i.e.* variants for the *oblig*, *exig*, *exclu* and *precede* Property types, the *exigac* Property type, through which one will specify agreement constraints, and the fact that one category

[1] In XFST, the schema of a `define` command is `define` *variable regular-expression* `;` the effect is to "invoke the compiler on the regular expression, create a network, and assign that network to the indicated variable. Once defined in this way, the variable [...] can be used in subsequent regular expressions." [1, p. 85]

[2] `www.xrce.xerox.com/competencies/content-analysis/fsCompiler/home.en.html`

`amod(id, [a, b, ..., z])` specifies that in a string of language `id`, one may only use words of category `a`, `b`, ..., or `z`.

```
define AMOD  [ a | b | ... | z ]+ ;
```

`uniq(id, [a, b, ..., z])` specifies that a string of language `id` may contain at most one `a`, at most one `b`, ..., and at most one `z`.

```
define UNIQ [$?a & $?b & ... & $?z] ;
```

`oblig(id, [a, b, ..., z])` specifies that a string of language `id` must contain one `a`, or one `b`, ..., or one `z`.

```
define OBLIG [$a | $b | ... | $z] ;
```

`exig(id, [a, b, c, ..., z])` specifies that in a string of language `id`, the presence of an `a` requires the presence of a `b`, or a `c`, ..., or a `z`.

```
define EXIG [~$a | [$a & $[b | c | ... | z]]] ;
```

`exclu(id, [a, b, c, ..., z])` specifies that in a string of language `id`, the presence of an `a` forbids the presence of a `b`, of a `c`, ..., and of a `z`.

```
define EXCLU [~$a | [$a & ~$[b | c | ... | z]]] ;
```

`precede(id, [a, [b, c, ..., z]])` specifies that, in a string of language `id`, if an `a` occurs with a `b`, or a `c`, ..., or a `z`, it must precede the `b`, `c`, ..., or `z`.

```
define PRECEDE [ a < [b | c | ... | z] ];
```

Fig. 1: *Translation table from Properties to XFST regular expressions.*

may be characterized as the *nucleus* of the defined strings. The absence of a translation for these features does not affect the discussion developed below.

More intersting is the fact that two characteristics of the formalism *had* to be set aside, because they were impossible to express by regular expressions:

1. The original definition of the *amod* Property also specifies that for each category, there exists at least one string which contains a word from that category (*i.e.* all the categories are used at least once). This is a condition on the whole set of strings, not on the strings themselves, and cannot be expressed by regular expressions.

2. The original formalism includes a *fleche* Property type, which specifies relations between the words composing a valid string and cannot be translated by regular expressions. However, it must be noted it has a special status, compared to other Properties, as it allows the expression of statements over the strings defined by the other Properties, but not to modify this set of strings by addition or subtraction.

2.3 An example

The regular expressions in figure 2 illustrate the definition of a language with Properties written as XFST regular expressions. The first nine lines define nine categories. The word forms considered in that example appear between quotes: *is, do, does, sing, sings, singing.* The words appearing next to a `define` command (e.g. `BE`) are category names. The language defined is `VC` (for "verb chunks"); it contains eight strings: {*do sing, do not sing, does sing, does not sing, is singing, is not singing, sing, sings*} [3].

3 Specificity of the Property formalism

Compared with more classical approaches, Properties offer a different perspective. We here compare this formalism with classical phrase structure grammars and with Koskenniemi's Finite-State Intersection Grammar (FSIG) [8].

3.1 Properties vs. phrase grammars

Like a regular expression or any phrase structure grammar, a set of Properties may be viewed as specifying a language. The novelty, when one compares Properties to classical phrase structure grammars, is that Properties systematically make use of *intersection* (as shown by the definitions of `id` at the end of section 2.1, and of `VC` in figure 2), and do not explicitly use *concatenation* (as shown by the absence of an explicit concatenation operator in the regular expressions of figure 1).

[3] With whitespaces added between the terminal symbols for readability.

```
define BE [ "is" ];
define DO [ "do" | "does" ];
define Aux   [ BE | DO ];
define VInf  [ "sing" ];
define V3Pr  [ "sings" ];
define VIng  [ "singing" ];
define VBase [ VInf  | V3Pr ];
define V     [ VBase | VIng ];
define Neg   [ "not" ];

define AMOD [V | Aux | Neg]+;
define OBL  [$V];
define UNIQ [$?V & $?Aux & $?Neg];
define RE1  [~$Neg   | [$Neg & $Aux]];
define RE2  [~$VIng  | [$VIng & $BE]];
define RE3  [~$DO    | [$DO & $VInf]];
define EX1  [~$V3Pr  | [$V3Pr & ~$DO]];
define EX2  [~$VBase | [$VBase & ~$BE]];
define PR1  [Aux < [Neg | V]];
define PR2  [Neg < V];

define VC [AMOD & OBL & UNIQ & RE1 & RE2
         & RE3 & EX1 & EX2 & PR1 & PR2];
```

Fig. 2: *Definition of a small example language.*

In contrast, phrase structure grammars favour *union* and *concatenation*. Typically, in a phrase structure grammar, for a given non terminal symbol A, one may have n rules with A on the left-hand side, which will be interpreted as stating that this symbol is to be rewritten as specified by rule 1, *or* rule 2, ... *or* rule n. In other words, the A language is the result of the *union* of the right-hand side specifications, where concatenation is the primary operation. As an example of this preferred use of union and concatenation, one would remark that the language VC of figure 2 would also, in a more classical manner, be defined by the following regular expression, *i.e.* as the union of three languages [4]:

```
define VC [VBase | BE (Neg) VIng
          | DO (Neg) VInf];
```

As this definition is more compact than that of figure 2, one might wonder what could be the advantage of using Properties. The advantage lies in the greater modularity of linguistic descriptions Properties offer. As noted by [10], Properties can be viewed as "a systematization of the decomposition of information initiated by the GPSG ID/LP formalism: the information expressed by the ID rules in GPSG are expressed by the conjunction of the *amod*, *uniq*, *oblig*, *exig* and *exclu* properties". The consequence of this decomposition is that it will be easier to adjust linguistic descriptions to what is seen as variations within a data set (e.g. regional variations of a given language, or spelling errors in a written corpus) [5].

3.2 Properties vs. FSIG

Properties contrast with classical phrase structure grammars in that they favor intersection, but [7] already described a parsing system based on constraints "implemented as finite-state machines" and where "the grammar as a whole is logically an intersection of all constraints". The result of our translation is conceptually identical to that framework, but the Property formalism, however, does differ from its predecessor.

In practice, the preferred rule format in the grammar described in [8] is `EXP => LC _ RC`, which specify that any occurrence of `EXP` must be surrounded by the given contexts `LC` and `RC` (all three parts of the rule being regular expressions). This kind of rules, like phrase grammar rules, in effect favours concatenation and union as the primary operations, as contexts are often specified as a disjunction of admissible strings.

In addition to this type of rules, the rule formalism of [8] gives the linguist the possibility to specify definitions of the form

`name(param`$_1$`, ..., param`$_n$`) = regex;`

which could be used to define not only Properties, in the same manner as we did in figure 1 [6], but also *any* new predicate. The formalism of [8] allows one to use the full power of regular expressions, while Properties, in contrast, form a closed set of predefined constraint schemata.

```
amod(S, [a, b, S])
uniq(S, [a, b, S])
oblig(S, [a])
exig(S, [a, b])
precede(S, [a, [b, S]])
precede(S, [S, [b]])
```

Fig. 3: *Definition of the a^nb^n language.*

4 Expressive power of the Property formalism

If it is possible, as we have shown, to translate Properties into regular expressions, then one must come to the conclusion that the expressive power of Properties is limited to the specification of regular languages. However, [3] gives an example of Properties specifying the context-free language a^nb^n, an example which is reproduced on figure 3. There is here an apparent contradiction, which deserves consideration.

We first examine the interpretation of Properties as specifying CF languages, and then discuss our stricter interpretation, in which they specify regular languages.

4.1 Properties specifying context-free languages

The understanding of Properties as specifying context-free or regular languages lies in the meaning one assigns to the symbols used as the arguments of the Properties. In our interpretation of Properties as

[4] Not counting the use of union in the category definitions, which we assume to be that of the first nine lines of figure 2.

[5] Section 5.2 gives hints at how such adjustments could be implemented.

[6] Indeed, [8] give as an example definition the statement `UNIQUE(FINV)`.

equivalent to regular expressions, the symbols used as arguments of the properties are variables which are defined non recursively. Properties apply to the strings of the language, strings of terminal symbols. In the interpretation of [3], the symbols used as arguments of properties may be either terminal symbols or recursively defined non terminal symbols (e.g. symbol `S` in figure 3). Properties do not apply directly to the strings of the language, but to the strings of immediate constituents of a category. The description presupposes a phrase structure tree, and the Properties apply to levels in this tree.

The $a^n b^n$ language example illustrates adequately this orientation: it is possible to say that the string *aabb* satisfies all the properties of figure 3 because the description implies the tree in figure 4, and the properties are about the immediate constituents of each S constituant, not about the string *aabb* itself (in which case, quite trivially, the property `uniq(a, b, S)` would *not* be satisfied, since the string contains several as and several bs).

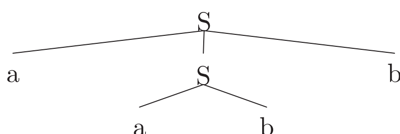

Fig. 4: *Analysis tree for the string aabb.*

[5] state that, in “Property Grammar”, “parse trees are no longer necessary”. [4] claims that “Property Grammar is a non-generative theory in the sense that no structure has to be build, only constraints are used both to represent linguistic information and to describe inputs.” As our analysis of the $a^n b^n$ example shows, if Properties are to be interpreted as possibly applying to recursively defined non terminal symbols (as indeed they are in the cited articles), these statements are wrong.

4.2 Properties vs. regular expressions

Coming back to the interpretation of Properties of figure 1, one may question whether the Property formalism has the same expressive power as regular expressions. The answer is no.

Assuming categories defined using only the union operator (as in figure 1), the expressive power of Properties (understood as in figure 1) is strictly smaller than that of XFST regular expressions. For instance, it is impossible to define with Properties the language denoted by the regular expression `[a (a)]`, *i.e.* the set $\{a, aa\}$. As a rule, one cannot precisely control with Properties how many words of the same category are allowed in a string (e.g. say “one or two `a`s”). One can only state that such words may or must appear (with the *amod* and *oblig* Properties, and that there may only be one such word (with the *uniq* Property). The *exig* Property cannot help in that matter as a Property such as `exig(id, [a, a])` would be trivially satisfied.

4.3 (Dis)advantages of the two interpretations

At this point, we are left with two interpretations of one formalism. Choosing one rather than the other would presumably depend on one's objectives and on the (dis)advantages of regular expressions vs. context-free grammars. As is well known, natural language has been shown to be non regular, but it has also been shown that some aspects of natural language syntax could be described by finite-state methods (e.g. chunks).

Appropriateness of Properties to such and such objectives (e.g. chunking vs. deep parsing) will be demonstrated by the development of effective linguistic descriptions. However, we would venture that the use of an underlying tree representation might suffer from two drawbacks: (1) interpretation of the Properties would sometimes be counter-intuitive (as when one reads that the string *aabb* satisfies the Property that there is only one a), and (2) it might make the Property formalism less relevant, as the tree tends to reduce the length of the strings Properties apply to. For instance, if one would work with binary branching trees, one might question whether Properties are not too sophisticated a system to describe two word strings.

5 Practical application of the translation scheme

Our translation of Properties into XFST regular expressions helped us to clarify the understanding of the Property formalim, but it also offers at no cost a platform to actually put Properties in practice, as one can use all the functionalities of the Xerox software. We here evoke some possible uses, which will be presented with the following definition as a reference:

```
define L [ P1 & P2 & ... & Pn ] ;
```

We will say that this formula defines the language `L` as well as the automaton `L`.

5.1 Analysis and generation

The most straightforward use of XFST will be to compile the automaton `L` and use it either to test whether a string belongs to the `L` language, or to generate strings of `L`, possibly *all* the strings of `L` if it is finite. In this latter case, XFST provides the “pattern generator” of [2, §5] and this shows that Properties may indeed be used for generation.

5.2 Multiple automata from a single set of Properties

[3] claims that Properties challenge generativity in that rather than parsing only grammatical sentences, one can take any input sentence and produce the lists of Properties it satisfies or not. This may be implemented within XFST by defining one automaton for each Property and analysing strings with each of these automata in turn. More generally, given a set P of

Properties defining $L(P)$, any subset P' of P can be used to define a language $L(P')$ of which $L(P)$ will be a subset. $L(P')$ can be viewed as a language resulting from the relaxation of some constraints on $L(P)$ (*i.e.* the subtraction of some Properties), a language which in effect contains sentences which would be judged ill-formed with respect to $L(P)$.

Properties offer modularity and an easy way to define from a single base set multiple languages included in each other. This quality, however, does not question the fact that the Property formalism in itself fully belongs to the generative grammar paradigm.

5.3 Testing the relevance of Properties

Another application of XFST is that it makes it possible to verify that in a set of Properties defining a language `L`, each Property is *relevant* to the definition of `L`. Given the definition of `L` above, the following XFST command sequence tests the relevance of any Property `Pi`:

`regex L ;`	Put the `L` network on the stack.
`regex L - Pi ;`	Put the `L - Pi` network on the stack.
`test equivalent`	Test whether the top two networks on the stack are equivalent.

A Property `Pi` is relevant to the definition of a language `L` iff `L - Pi` is not equivalent to `L`.

Note that this relevance testing procedure makes use of the subtraction operator. Unlike context-free languages, regular languages are closed under subtraction, as well as under intersection. We may then view this procedure as another advantage of our XFST interpretation of Properties.

5.4 Exploring the relevance of the Property formalism

Ultimately, the relevance of Properties will be demonstrated (or not) by effective descriptions of natural languages. Looking at our translation of Properties into regular expressions, one might wonder what would be the point of using Properties rather than regular expressions?

The weaker expressive power of Properties (cf. section 4.2) actually suggests a nice experimentation program: how far can we go into the description of natural languages with Properties understood as in figure 1? The objective would be to determine what, within finite-state expressivity, is needed or not to describe such and such aspect of a language, to find the appropriate position between a system using the full power of regular expressions and a system strictly limited to a specific set of constraint schemata.

Properties put constraints on the linguist's expression, but it might be to their benefit. [9] introduced a translation system from natural language to XFST regular expressions. This author, pointing out that the same thing could be said in a messy way as well as in a structured way, concluded his demonstration by advocating the importance of structured programming. We believe Properties are a good way to structure linguistic descriptions. Especially if, rather than the sometimes cumbersome notation of regular expression Properties in figure 1, one considers the possibility of an interface to this notation.

6 Conclusion

We presented a translation of the Properties of [2] into regular expressions, a translation which we consider an *indirect* implementation of this formalism. To us, this indirectness is an advantage, because

- it helped to clarify the interpretation of the Property formalism,
- it provides at no cost a tool to actually analyse and generate strings defined by a set of Properties, as well as a tool to test the relevance of each Property,
- as it integrates Properties into a system with greater expressive power, it opens space to test the limits of the Property formalism on linguistic data.

With respect to the interpretation of Properties, our comparison shows to what extent they depart from classical phrase structure grammars, favouring the definition of a language by the intersection of sets rather than union, but also to what extent they do belong to this paradigm. In particular, as any set of Properties may indeed be translated into an equivalent regular or context-free grammar, they can be assigned the same interpretation as such grammars, *i.e.* they are expressions which denote languages.

References

[1] K. R. Beesley and L. Karttunen. *Finite Stale Morphology.* CSLI Studies in Computational Linguistics, 2003.

[2] G. G. Bès. La phrase verbale noyau en français. *Recherches sur le français parlé*, 15:273–358, 1999.

[3] P. Blache. *Les Grammaires de propriétés.* Hermès Science Publications, 2001.

[4] P. Blache. Property grammars: A fully constraint-based theory. In *Constraint Solving and Language Processing.* 2005.

[5] V. Dahl and P. Blache. Directly executable constraint based grammars. In *Proc. Journées Francophones de Programmation en Logique avec Contraintes*, Angers, France, June 2004.

[6] L. Karttunen, T. Gaál, and A. Kempe. *Xerox Finite-State Tool.* The Document Company - Xerox, 1997.

[7] K. Koskenniemi. Finite-state parsing and disambiguation. In H. Karlgren, editor, *Proceedings of COLING-90*, Helsinki, 1990.

[8] K. Koskenniemi, P. Tapanainen, and A. Voutilainen. Compiling and using finite-state syntactic rules. In *Proceedings of COLING-92*, Nantes, 1992.

[9] A. Ranta. A multilingual natural-language interface to regular expressions. In L. Karttunen and K. Oflazer, editors, *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, pages 79–90, Bilkent University, Ankara, 1998.

[10] F. Trouilleux. Note de lecture sur Philippe Blache, *Les Grammaires de propriétés*, Hermès Science Publications. *TAL*, 44(2):256–259, 2003.