

# Directly Executable Constraint Based Grammars

Veronica Dahl<sup>1&2</sup> & Philippe Blache<sup>1</sup>

<sup>1</sup> LPL, Université de Provence  
Aix-en-Provence, France  
`pb@lpl.univ-aix.fr`  
<sup>2</sup> Simon Fraser University  
Vancouver, Canada  
`veronica@cs.sfu.ca`

**Abstract.** We propose a direct interpretation for constraint-based linguistic formalisms in which the notions of derivation and hierarchy give way to the more flexible notion of property satisfaction between categories. Such frameworks define sentence acceptability in terms of the properties that must be satisfied by groups of categories (e.g. English noun phrases can be described in Property Grammar terms [Blache01] through a few properties such as precedence (a determiner must precede a noun); uniqueness (there must be only one determiner); exclusion (an adjective phrase must not coexist with a superlative); and so on). Rather than resulting in either a parse tree or failure, such frameworks characterize a sentence through the list of the properties it satisfies and the list of properties it violates. A same structure can bear subparts with different granularities. Properties can be relaxed.

We use a specific constraint programming language, CHR<sub>G</sub> (Constraint Handling Rule Grammars, [Christiansen01]), built on top of CHR [Frühwirth98], for a direct interpretation of Property Grammars which preserves all its theoretical features at the implementation level. Conditions can be defined under which a given property can be relaxed. This allows us to parse incomplete and incorrect input in a very modular and adaptable, while efficient, manner. Our parser uses essentially a single rewrite rule which combines two categories into a third provided they verify the non-relaxable properties involving them. Although parse trees are no longer necessary, we construct them as a side effect of parsing, even in the case of incomplete or incorrect input. We present as well an analysis of property inheritance which allows us to inherit at each stage most of the previous stage's properties, while calculating only the minimally necessary new properties and updating others.

Our system has been successfully tested for a Property Grammar fragment for noun phrases, and is currently being tested for a larger coverage. Together with work on constructivism, the language processing methodologies presented here have inspired a cognitive model of concept formation [Dahl04] which has been used in medical applications [Barranco04].

## 1 Introduction

A present challenge for natural language processing techniques is reusability for different purposes. Concerning parsing, the question is more precisely to develop systems that can adapt to different granularities. For example, some applications such as speech synthesis only need shallow information, whereas other systems (e.g. machine translation) rely on deeper analyses. Even within the same application it is sometimes necessary to consult different levels of analysis (e.g. a semantic component might be consulted during syntactic parsing in order to help correct mistakes in text produced by speech recognition software).

Flexible parsing techniques offer great advantages in this perspective. Among possible solutions, constraint-based approaches have the advantage that the same parsing mechanism, constraint satisfaction, can be applied whether the grammar is incomplete, heterogeneous, etc. A same structure can bear subparts with different granularities. Moreover, provided that no extra mechanisms than satisfaction are used and that all information is represented by means of constraints, constraints can be relaxed in accordance to user-defined criteria.

Several linguistic theories make an intensive use of this notion, in particular HPSG (see [Pollard94], [Sag99]) or the Optimality Theory (see [Prince93]).

Property-based linguistic models [Bès99a,Bès99b,Blache00] view linguistic constraints as properties between sets of categories, rather than in the more traditional terms of properties on hierarchical representations of completely parsed sentences. This view has several advantages, such as allowing for mistakes to be detected and pointed out rather than blocking the analysis altogether, and facilitates dynamic processing of text produced on the fly, as needed for the growing number of applications involving speech.

As argued in [Morawietz and Blache, 2002], CHR rules [Fruhwirth 2000] lend themselves beautifully to the specialized task of describing such models. While they discuss control issues, these are not addressed in their implementation, which is combinatorially explosive.

In this paper, we describe a methodology for *Property Grammars* [Blache00] which relies exclusively on constraints, controls the parse through head-driven analysis, and provides a direct interpretation of this formalism, while preserving all its theoretical properties at the implementation level. We use for this a specific constraint programming language called CHR<sub>G</sub> (Constraint Handling Rule Grammars) described in [Christiansen01] on top of CHR [Frühwirth98]. The methodology presented here has, together with constructivism theory, inspired a cognitive science model of concept formation [Dahl04] which is being used for medical applications [Barranco04]. For completeness and explanatory purposes, we start with a brief summary of this model, which we then specialize to our parsing purposes. Section 2 overlaps partially with [Dahl04].

## 2 Modelling constructive knowledge

### 2.1 The intuitive idea

Constructivism is the idea that we construct our own world rather than it being determined by an outside reality. Because the idea of constructing (solutions, proofs) is at the heart of both traditional computing science and logic programming (through the rough respective methods of computation and inference), constructivism can constitute a natural bridge between cognitive and computing sciences. Constructive theories view learning as the construction of new concepts from previously known concepts. During this process, “the learner selects and transforms information, constructs hypotheses, and makes decisions, relying on a cognitive structure to do so”. The formation of new concepts from known ones is also central to cognitive logic, and also to logic programming and its recent new paradigm, Constraint Handling Rules (CHR). [Dahl04] takes advantage of these natural connections to develop a cognitive model of knowledge construction which can be directly executed through (a specialized system implemented in) CHRs. In this model, information is selected automatically as a side effect of (the system) searching through applicable CHR rules, and automatically transformed (or simply augmented) when a rule triggers; hypotheses can be made in the form of assumptions [Dahl97]; decisions follow from the normal operation of the rules, and cognitive structure is given by properties that the concepts a given rule is trying to relate must satisfy. Moreover some latitude is provided by which rather than rigidly having to satisfy all properties defined as necessary for a concept to form, the user can declare under what circumstances a given property or properties can be relaxed. Concepts formed under relaxed properties result in output which signals not only what concepts were formed, but which of the properties associated with that concept’s construction were satisfied and which were not. This allows us human-like flexibility while maintaining direct executability.

### 2.2 Properties and their role in concept formation

Consider the following CHR rule, which could loosely be used by an expert system for fashion design:

$$(1) \quad \text{colour}(A, \text{blue}, P_b), \text{colour}(B, \text{yellow}, P_y) \Rightarrow P_b > P_y, P_g \text{ is } P_b + P_y \mid \text{colour}(A-B, \text{green}, P_g).$$

Such a rule could be used to indicate that if material A is  $P_b$  percent blue and material B is  $P_y$  percent yellow, where the blue dominates (the percentage of blue is greater than that of yellow), we can combine them into the material named A-B that will be  $P_b + P_y$  percent green.

Notice the qualitative difference between the two tests in the guard of rule (1): while the second test is a utility test (i.e. a mere calculation of some value

that needs to be included in the new concept), the first test represents a property that attributes of the two intervening concepts must satisfy in order for the new concept to be derivable from them.

We identify such tests under the name of *properties*. These must be referred to through the three primitive predicates described below. Handling properties through our system primitives rather than as simple Prolog tests allows our system to invisibly keep track of the (degree of) satisfaction of these properties for each newly derived concept, and report it to the user.

**Property definition** We must first name the property, e.g. for rule (1) let's call it greenness. Let P be the Prolog specification of a property named N, let A be the list of attributes intervening in P, and let L be the list whose head is N and whose tail is A. The general form of a property definition is

```
prop(L):- P.
```

For our example, we have:

```
prop([greenness,Pb,Py]):- Pb>Py.
```

**Property Relaxation** Flexibility in deriving concepts is obtained by being able to relax the enforcement of certain properties under user-defined criteria. As a result, a list of satisfied and unsatisfied properties will be produced for concepts where some properties have been relaxed, and in some cases the degree of "incorrectness" will be output as well. In order to relax a property named N (i.e. to allow the derivation of concepts that require it but for which it is not satisfied), we simply write the following:

```
relax(N).
```

For our present example, we can write the following to express that the property named greenness, which is satisfied when the percentage of blue is greater than that of yellow, can be relaxed unconditionally:

```
relax(greenness).
```

Degrees of acceptability can be defined through a binary version of the relaxing primitive, where L is as above and D is a measure of acceptability:

```
relax(L,D).
```

For instance, to keep track of how much we violate the greenness constraint in our fashion design example, we can write

```
relax([greenness,Pb,Py],2):- Py = 2 * Pb, Py<3*Pb.
```

This expresses the degree of violation of our property as two (the amount of yellow is more than twice, but less than three times, the amount of blue). Our system automatically collects the degrees of violation of our properties and reports them to the user.

**Use of properties within rules** The acceptability of a property must be determined within the rule requiring that property to be tested. Let prop stand for that property's name; let L be as above; let D be the degree of acceptability and L1 the concatenation of L and D. All we need to do is to call `acceptable(prop(L1))` inside the rule's guard. As a result, the property in question will be tested, and D will unify with either true, false, or some user-defined degree of acceptability.

For instance, rule (1) must now be expressed as:

```
(2) colour(A,blue,Pb), colour(B,yellow,Py) =>
    acceptable(greenness(Pb,Py,D), Pg is Pb+Py |
    colour(A-B,green,Pg).
```

A list of satisfied and violated properties, together with the degree of violation if appropriate, will be output for each property defined in a given CFC program.

### 2.3 CFG, or Concept Formation Grammars

Concept Formation Grammars are built on top of CHR [Christiansen01], which are to CHR what DCGs are to Prolog. Just as DCG rules compile into Prolog rules, CHR rules compile into CHR rules, e.g. the CHR rule (3) below compiles into the CHR rule (4):

```
(3) determiner, noun, verb ::> sentence.
```

```
(4) determiner(P1,P2), noun(P2,P3), verb(P3,P) => sentence(P1,P).
```

Concept Formation Grammars are CHR plus properties and acceptability defined in the same way as for plain CF rules. Because properties belong in the guard part of a rule, which contains anyway calls to Prolog rather than grammar symbols, no extra apparatus is needed to go from CF programs to CF grammars. For applications in which sentence boundaries need to be manipulated explicitly, CHR includes facilities to make the word boundaries explicit as required. They moreover include left and right context, as well as CHR includes also notation for gaps and for parallel matching, which we neither describe nor use in the present work. We can define the subset of CFG needed for our purposes more formally, as follows.

Definition: a CFG is a finite set of CHR rules of the form

Head => Guard | Body

where Head and Body are conjunctions of atoms and Guard is a test constructed from (Prolog) built-in or system-defined predicates plus the CF specific predicate `property/2`; the variables in Guard and Body occur also in Head; if the Guard is the constant “true” then 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 or grammar is given by conjunction. CFG rules are interpreted as rewrite rules over stores of logic grammar symbols. For instance, the following toy CHR<sup>1</sup>

```
[a] ::> determiner(singular).
[boy] ::> noun(singular).
[boys] ::> noun(plural).
[laughs] ::> verb(singular).
```

```
determiner(Number), noun(Number), v(Number) ::> sentence(Number).
```

can be used to parse correct sentences such as “a boy laughs”.

In order to accept also sentences which do not agree in number, while pointing out the error as a side effect of the parse, we can replace its last rule by the following CFG rules:

```
determiner(Ndet), noun(Nn), v(Nv) => acceptable(agreement,Ndet,Nn,Nv,N,_) |
sentence(N).
```

```
prop(agreement, [Ndet,Nn,Nv,N]) :- Ndet=Nn,
Nn=Nv, !, N=Nv.
prop(agreement, [Ndet,Nn,Nv], mismatch).
```

```
relax(agreement).
```

The agreement property will appear in the list of violated properties automatically constructed as a result of the parse. In the case of our example, we don’t need to make use of the boolean value which `acceptable/2` calculates into its second argument, so we leave it anonymous.

So far, we have remained within the traditional framework of rewrite rules which implicitly define a parse tree. We next use CFGs to implement the more flexible property grammar linguistic model.

---

<sup>1</sup> terminal symbols are noted between brackets as in DCGs

### 3 CFG for Property Grammars

#### 3.1 Background: Property Grammars

The basic idea of *Property Grammars* is to represent different kinds of syntactic information separately. In this approach, syntactic structure is not expressed in terms of hierarchy, but only by means of relations between categories. Such relations do not have any topological constraints, they can for example be crossed. Moreover, only relations between objects are used for describing a category. As a consequence, the notion of constituency is no longer crucial for the description process: a category is specified by a set of properties rather than by a set of constituents. In other words, the fact that several categories belong to a network of relations indicates that they characterize an upper-level category. A syntactic category is then described by a set of properties which represent relations between other categories (lexical or syntactic).

In this approach, the goal is to make explicit all the different relations that can exist. We distinguish in this perspective the following types of information:

- linear precedence, which is an order relation between categories,
- subcategorization, which indicates cooccurrence relations between categories or sets of categories,
- the impossibility of cooccurrence between categories,
- the impossibility for a category to be repeated,
- the minimal set of obligatory constituents (usually one single constituent) which is the head,
- semantic relations between categories, in terms of dependency.

These different kinds of information correspond to different properties, respectively: *linearity*, *requirement*, *exclusion*, *unicity*, *obligation*, *dependency*. Such information can always be expressed in terms of relations between categories, as shown in the following examples:

- Linear precedence:  $Det \prec N$  (a determiner precedes the noun)
- Dependency:  $AP \rightsquigarrow N$  (an adjectival phrase depends on the noun)
- Requirement:  $V[inf] \Rightarrow to$  (an infinitive comes with *to*)
- Exclusion:  $seems \not\Leftarrow ThatClause[subj]$  (the verb *seems* cannot have *That* clause subjects)

All syntactic categories are characterized by a set of relations that forms a connected graph. The syntactic description of a language consists of all the different relations that can be expressed between categories. A relation (also called a property) can be conceived as a constraint on the set of categories. A grammar is then a set of constraints and satisfiability becomes the core of the parsing process (see [Blache01]). What is interesting in this approach is that no implicit information, for example under the form of a specific mechanism, is needed. In particular, there is no need to build a structure before being able to verify its properties as it is the case with classical generative approaches.

Moreover, using satisfiability alone has important consequences in the conception of the syntactic structure. Evaluating a constraint system for a given set of categories makes it possible to specify precisely the set of properties that are verified. In the same way, in the case of ill-formed input, such evaluation identifies precisely the set of satisfied and violated constraints. Such a result is then of deep interest in the sense that it identifies precisely all the specificities of an input. In *Property Grammars* this information constitutes the output of a parse, which is but the status of the constraint system after evaluation.

### 3.2 Parsing schema

The basic mechanism in constraint satisfaction problems is to find, for a given set of variables, an assignment that satisfies the constraint system. In the problem addressed here, the variables are taken from the set of categories. An assignment is given from an input (i.e. the sentence to be parsed). Starting from the set of lexical categories corresponding to the words of the sentence, all possible assignments (i.e. subsets of categories) are evaluated. When a syntactic category is characterized, it is added to the set of categories to be evaluated. This approach is basically incremental in the sense that any subset of categories can be evaluated. This means that an assignment  $\mathcal{A}$  can be completed by adding other categories. When syntactic categories are inferred after the first step of the process, it is then possible to complete the first assignments (made with lexical categories) with new syntactic ones.

We have seen that in Property Grammars, a category is described by a set of constraints. But reciprocally, it is possible to identify a category from a given property. This is typically the case with properties expressing relationships between categories, such as linearity, requirement, obligation and dependency. Evaluating such properties makes it possible to infer that the syntactic category to which the property is attached is being characterized. We have referred to these properties under the collective name of *selection constraints*.

The role of selection constraints is central to our approach. The reason such constraints allow us to select the characterized category is that they are local to this category. Moreover, in some cases they have a global scope over the category : their satisfiability value (i.e. satisfied or violated) cannot change for a given category whatever the subset of constituents. As soon as the constraint can be evaluated, this value is permanent. For example, when a linearity or a dependency constraint is satisfied, adding new constituents to the category cannot change this fact. Other kinds of constraints have to be re-evaluated at each stage. For instance, when adding a new category, we need to verify that unicity and exclusion are still satisfied. In all that follows, we call the latter *filtering constraints*. Contrary to *selection constraints*, one cannot infer the materialization of a syntactic category from their evaluation. They play a filtering role in the sense that they rule out some construction. Yet another type of constraint, which we call *recoverable constraints* can succeed by the incorporation of one more category into a given phrase for which, without this added category, the constraint failed.

Let us examine what the consequences of this are on constraint evaluation. As explained above, the principle consists in completing original assignments with new categories when they are inferred. Insofar as the evaluation of selection constraints (as soon as this evaluation can be performed) is valid through a complete assignment, whatever its constituents, it is not necessary to re-calculate it. In other words, when an assignment  $\mathcal{A}$  is made by completing another assignment  $\mathcal{A}'$ , the set of selection constraints of  $\mathcal{A}'$  is inherited by  $\mathcal{A}$ .

Section 3.7 studies the different types of properties and their consequence for new assignments in more detail, with respect to a specific instance of the general parsing schema we present here.

In the following, we note selection and filtering constraints as  $\mathcal{R}_{select}(\mathcal{C}, XP)$  and  $\mathcal{R}_{filter}(\mathcal{C}, XP)$  in which  $\mathcal{C}$  is the constraint and  $XP$  the syntactic category to which the constraint is associated. For some assignment  $\mathcal{A}$ , a constraint is relevant (or can be evaluated) when the categories of  $\mathcal{A}$  are a subset of the categories involved in  $\mathcal{C}$ . We note the fact that  $\mathcal{A}$  can be evaluated for some constraint as follows:  $\mathcal{A}/\mathcal{R}_{select}(\mathcal{C}, XP)$ . We note the set of filtering and selection constraints for a given category  $XP$  by  $\mathcal{R}(\mathcal{C}, XP) = \mathcal{R}_{select}(\mathcal{C}, XP) \cup \mathcal{R}_{filter}(\mathcal{C}, XP)$ . Finally, we note the state of the constraint system  $\Sigma$  for an assignment  $\mathcal{A}$  after evaluation by  $SAT(\mathcal{A}, \Sigma)$ . Each category is indexed by its boundaries, noted  $c_{(i,j)}$ .

Let  $\mathcal{K}$  be the set of categories, let  $c_i, c_{i+1} \in \mathcal{K}$

1.  $\mathcal{A} \leftarrow \{c_{(i,j)} c_{(j+1,k)}\}$
2. if  $\exists \mathcal{A}/\mathcal{R}_{select}(\mathcal{C}, XP)$
3.     instantiate  $XP_{(i,k)}$
4.      $\Sigma_{XP} = \bigcup \mathcal{R}(\mathcal{C}, XP)$
5.      $Char(\mathcal{A}) \leftarrow SAT(\mathcal{A}, \Sigma)$
6.      $\mathcal{A}' \leftarrow \mathcal{A} \cup \{c_{k+1,l}\}$
7.     while  $SAT(\mathcal{A}', \Sigma)$  is acceptable
8.      $\mathcal{A} \leftarrow \mathcal{A}'; Char(\mathcal{A}) \leftarrow SAT(\mathcal{A}, \Sigma)$
9.      $\mathcal{A}' \leftarrow \mathcal{A} \cup \{c_{(l+1,m)}\}$

In this algorithm schema, the mechanism consists in evaluating the characterization of all sequences of categories. The specificity of selection constraints is used as a control device: when a selection constraint is satisfied, the described category  $XP$  is instantiated and the related set of constraints  $\Sigma$  is activated. It is interesting to notice that a syntactic category can be projected by any selection constraint, independently from any constituency information (especially the head). Each new assignment, built by adding new juxtaposed categories to the initial set, is then evaluated. Such a completion of the initial assignment is possible when the satisfiability of  $\Sigma$  for this new assignment is *acceptable* (cf. line 7). This notion implements the flexibility of the parser. When we need to build only grammatical structures, *acceptability* is reduced to satisfiability. But for more flexible parsing needs (e.g. spoken language), constraints can be relaxed. The set of constraints to be relaxed, their number, etc. is indicated at

this point. Finally, the general process is repeated until no new category can be added.

This parsing schema proposes a general framework in which constraints can be integrated. Each property is implemented by a constraint solver. The mechanism consists in building a characterization for each possible assignment (i.e. any subset of categories). The particularity of selection constraints plays an important role in this schema. In a classical bottom-up technique, the mechanism consists in finding a *handle* which links a set of categories with a non-terminal. Such a relation in our approach is established between a set of properties and a category. In contrast with phrase-structure techniques, the notion of constituency does not play any particular role. As soon as a selection constraint is evaluated, the corresponding syntactic category is added to the set of categories and all the constraints participating in its description are activated. Concretely, all the selection constraints can be evaluated with no need to know the upper-level category, in contrast to filtering constraints which have to be activated.

Different strategies can be applied according to the needs of the parse. A restricted application stipulates that all constraints have to be satisfied. In this case, only grammatical characterizations are built, all ill-formed structures are ruled out. For more flexible applications, typically in the case of parsing spoken language material, constraints have to be relaxed. In this case, characterizations can contain violated constraints. We next describe how CFGs are used in our approach to achieve direct interpretation in these kinds of flexible applications.

### 3.3 Direct interpretation

Our methodology for Property Grammar parsing has been designed to provide direct interpretation of Property Grammar rules. This is an interesting contribution with respect to Property Grammars themselves, but also a novel and important proof-of-concept that can lead the way for any constraint-based parsing formalism which relates categories contextually through their properties. To the best of our knowledge, our methodology is the first one that permits an expression of such parsing constraints which is directly and efficiently executable. Thus, we can say that our approach represents for grammars based on contextual properties what DCGs represent for context-free grammars, in the sense that they are as directly executable descriptive formalisms as DCGs<sup>2</sup>.

In this section we describe the different components of our methodology : the notion of extended categories, which includes not only traditional information such as category names and features, but also the category's characterization in terms of satisfied and unsatisfied constraints; the modular notation through which a user defines the properties of a given grammar, the single rule through which parsing proceeds, and the analysis of property inheritance which is used in our system.

---

<sup>2</sup> Of course, DCGs do permit context sensitive parsing as well, but the context sensitivity cannot be directly expressed through symbol contiguity, it has to be indirectly expressed in extra arguments or through other extra devices such as linear implication.

### 3.4 Extended Categories

Extended categories are of the form: *cat*(*Name*, *Features*, *Graph*, *Sat*, *Unsat*) where *Name* is the category name, *Features* a list of features associated with the category (which may be used to check some of the properties between categories), *Graph* is a parse tree which is obtained as a side effect of parsing (which is built even in those cases of incorrect input), and *Sat* and *Unsat* are respectively, the list of satisfied and unsatisfied properties that the immediate daughters of *Name* inside the *Graph* verify between them. In the case of single word categories, the *Sat* and *Unsat* lists will be empty. These categories are created automatically from user's lexical definitions, which are done in terms of CHR<sub>G</sub>. For instance, a user's entry:

```
[1e] ::> cat(det,[singular,masculin]).
```

compiles into:

```
[1e] ::> cat(det,[singular,masculin],det(1e),[],[]).
```

Because these are CHR<sub>G</sub> rules (i.e., grammar rules, as opposed to plain CHR rules), word boundaries are carried invisibly. If needed, we can retrieve them in a grammar rule by adding `:(Start,End)` after the category, which will unify `Start` to the starting point of the category, and `End` to its end point, or we can write a plain CHR rule that looks at `cat/5` not as a grammar rule, but as the CHR constraint it compiles into, in which case `Start` and `End` can be retrieved as the two first arguments of the corresponding constraint, `cat/7`.

### 3.5 User defined properties

Our system allows the user to enter the specific linear precedence, dependency, requirement, exclusion, constituency and unicity properties that apply to the grammar being defined, through simple primitive predicates which are respectively `prec/3`, `dep/3`, `req/3`, `exclude/3`, `cons/2`, and `one/2`. Figure 3.5 exemplifies for a simplified noun phrase.

It is to be noted that while the user's definition of constituency is not rigorously needed (since, as we have seen, when selection properties are verified, the determination of constituency follows as a side effect), having it explicitly defined results in improved efficiency. Likewise, phrase definitions can be inferred from any of the other properties, but defining them explicitly makes the system easier and more readable. The user's definitions of properties will be called from system predicates which verify each of these properties on a given set of categories, as we shall see next.

### 3.6 Inferring new categories: one rule fits all

**Contiguous Constituents** Conceptually, little more than a single rule is enough for a string's complete bottom-up parse from contiguous constituents.

|                           |                     |                          |                 |
|---------------------------|---------------------|--------------------------|-----------------|
| <i>Linear precedence:</i> | <i>Dependence:</i>  | <i>Constituency:</i>     |                 |
| prec(det,n,sn).           | dep(det,n,sn).      | cons(sn,[det,adj,sa,n]). |                 |
| prec(det,sa,sn).          | dep(n,sa,sn).       | cons(sa,adj).            |                 |
| prec(n,sa,sn).            |                     |                          |                 |
| <i>Unicity:</i>           | <i>Exclusion:</i>   | <i>Requirement:</i>      | <i>Phrases:</i> |
| one(det,sn).              | exclude(sa,sup,sn). | req(n,det,sn).           | xp(sn).         |
|                           |                     |                          | xp(sa).         |

**Fig. 1.** User defined properties

This rule combines two consecutive categories (one of which is of type XP or obligatory) into a third, by testing each of the properties on the pair and creating the new property lists through property inheritance (cf. next section). Its form is described in Fig. 2.

```

cat(Cat,Features1,Graph1,Sat1,Unsat1):(Start1,End1),
cat(Cat2,Features2,Graph2,Sat2,Unsat2):(End1,End2) :>
  xp_or_obli(Cat2,XP), ok_in(XP,Cat),
  acceptable(precedence(XP,Start1,End1,End2,Cat,Cat2,Sat1,Unsat1,SP,UP,BP),
  acceptable(dependency(XP,Start1,End1,End2,Cat,Features1,Cat2,Features2,SP,UP,SD,UD,BD),
  build_tree(XP,Graph1,Graph2,Graph,ImmDaughters),
  acceptable(unicity(Start,End2,Cat,XP,ImmDaughters,SD,UD,SU,UU,BU),
  acceptable(requirement(Start,End2,Cat,XP,ImmDaughters,SU,UU,SR,UR,BR),
  acceptable(exclusion(Start,End2,Cat,XP,ImmDaughters,SR,UR,Sat,Unsat,BE)
  | cat(XP,Features2,Graph,Sat,Unsat).

```

**Fig. 2.** New Category Inference

This rule first tests that one of the two categories is of type XP (a phrase category) or obligatory (i.e., the head of an XP), and that the other category is an allowable constituent for that XP. It then successively tests each of the PG properties among those categories, incrementally building as it goes along the lists of satisfied and unsatisfied properties. Finally, it infers a new category of type XP spanning both these categories, with the finally obtained **Sat** and **Unsat** lists as its characterization.

In practice, this rule unfolds into two symmetric parts, to accommodate the situation in which the XP category appears before the category **Cat** which is to be incorporated into it.

**Obligation** The obligation property, which states which categories (in general, the head) are obligatory in a phrase, is not explicitly represented as a call, as is

the case with all other properties. Obligation is ensured by rules which project a phrase's kernel (i.e., its obligatory category) into the phrasal level all by itself. Since contextual rules incorporate more categories into phrasal ones already found, and since the smallest such phrasal categories include the phrase's kernel, there is no way for an obligatory category to go missing.

**Discontiguous Constituents** Constituents with discontinuities must also be allowed, for completeness. In this case we consider two categories where the End node of the first does not coincide with the start node of the second. Our current research does not yet include discontiguous constituents, for which further linguistic constraints for avoiding combinatorial explosion need to be incorporated.

### 3.7 Property Inheritance Analysis

As mentioned earlier, not all PG properties are inherited from a phrase into which another category is being incorporated. In this section we analyse the properties of linear precedence, dependence, requirement, exclusion and unicity in the light of determining how the properties that have been found to hold (fail) in a given phrase are inherited or not by the new phrase comprising that phrase plus a category being incorporated into it. Let  $XP$  be a phrase into which we are considering incorporating a category  $Cat$ . Let  $P$  be one of the properties above mentioned. Let:

- $Fp(XP)$  = set of  $P$  properties that fail to hold within  $XP$
- $Sp(XP)$  = set of  $P$  properties that hold within  $XP$
- $fp(Cat, XP)$  = set of  $P$  properties that fail to hold between  $Cat$  and  $XP$
- $sp(Cat, XP)$  = set of  $P$  properties that hold between  $Cat$  and  $XP$

Let us designate by  $XP+Cat$  the new constituent (of type  $XP$ ) formed by incorporating  $Cat$  into  $XP$ .

**Definition** : We say that a property  $P$  is success-monotonic (failure-monotonic) if all  $P$  properties that hold (fail) in  $XP$  also hold (fail) in  $XP+Cat$ . We shall consider three types of properties: selection, filtering and recoverable.

**Selection properties** Selection properties are both success monotonic and failure monotonic. In GPs, they are represented by linear precedence and dependency: constituents that precede or depend on others in a smaller phrase continue to precede or depend on them in the extended phrase, and those that are ill-ordered or do not satisfy dependency in a smaller phrase continue to fail in those ways when the phrase is extended.

**Filtering properties** Filtering properties are failure monotonic but not success monotonic. Examples are unicity and exclusion, which if failing for a smaller constituent, continue to fail in a bigger one, but their satisfaction in a smaller constituent does not guarantee satisfaction in a bigger one containing it- adding one more element may introduce duplication or lack of exclusion.

**Recoverable properties** We call properties such as requirement recoverable because if they fail inside a given XP, its extension by a category **Cat** may recover them from this failure by incorporating precisely the element that was required inside XP and was absent. Recoverable properties are success monotonic (e.g. requirements that were satisfied in a given XP continue to be satisfied after its extension by one more category), but not failure monotonic.

Figure 3.7 shows equations for calculating the set of failed and succeeding properties of an XP that incorporates one more category **Cat**.

|                     |   |
|---------------------|---|
| Filtering :         |   |
| Unicity :           | $FU(XP+Cat) = FU(XP) \cup \{fu(Cat,XP)\}$<br>$SU(XP+Cat) = SU(XP) \cup \{su(Cat,XP)\} - \{fu(Cat,XP)\}$ |
| Exclusion :         | $FE(XP+Cat) = FE(XP) \cup \{fe(Cat,XP)\}$<br>$SE(XP+Cat) = SE(XP) \cup \{se(Cat,XP)\} - \{fe(Cat,XP)\}$ |
| Selection :         |   |
| Linear Precedence : | $FP(SK+Cat) = FP(XP) \cup \{fp(Cat,SK)\}$<br>$SP(XP+Cat) = SP(XP) \cup \{sp(Cat,XP)\}$                  |
| Dependence :        | $FD(XP+Cat) = FD(XP) \cup \{fd(Cat,XP)\}$<br>$SD(XP+Cat) = SD(XP) \cup \{sd(Cat,XP)\}$                  |
| Recoverable :       |   |
| Requirement :       | $FR(XP+Cat) = FR(XP) \cup \{fr(Cat,XP)\}$<br>$SR(XP+Cat) = SR(XP) \cup \{sr(Cat,XP)\} - \{fr(Cat,XP)\}$ |

**Fig. 3.** Property inheritance rules

## 4 Refining the notion of characterization

The aim of flexibility was paramount in our design of the present approach, since we are interested in such applications as systems for aid to the handicapped, voice commanded systems, and the like, in which the hesitations and errors the input is expected to exhibit is usually higher than for more traditional language processing applications. Naturally, as we increase the level of tolerance for errors, we also decrease parsing effectiveness because more options are created, among which the best final choice may have become unclear. In order to allow for maximum flexibility, we provide a facility to declare which properties must be strictly observed, and which could be relaxed. Thus for instance, in a Spanish language learning system used by anglosaxon speakers, we might want to relax dependency, so as to be tolerant of gender agreement mistakes. This feature can be exploited in view of error correction : allow the ill-formed input to be generated, then examine possible repairs of this input, once the error has been localized as a failed dependency property.

The level of acceptability expected is in the present version of our system indicated by the user's definition :

`tolerate_violations_of(Prec,Dep,Uni,Req,Excl)`.

where a property required (not required) by the user to be satisfied is indicated by **yes** (**no**) in the corresponding argument. This approach to flexibility allows for easy experimentation. All we have to do in order to test different levels of error tolerance is to change the **yes** and **no** values of the `tolerate_violations_of/1` fact, and see what happens. From an implementation point of view, all that is needed to ensure the user's acceptability requirements is to block the generation of a given category when it is associated to a failed property that has been required to succeed. A characterization of a category to be inferred, then, now becomes a list of required properties that are satisfied, together with a list of failed properties that are not required.

Of course, it is possible to refine the characterization even further by partitioning the set of instances of each property into a set that can be relaxed and another that should be enforced, or even into several sets with an associated "degree of acceptability" value- we have already seen the CFG primitives that allow us to do this.

## 5 Discussion, Related Work

In our approach, as we have seen, syntactic categories are inferred from the evaluation of properties, without any need of constituency information.

This aspect has important consequences on the role of constraints in the parsing process. One of the problems with constraint-based approaches is that constraints are usually expressed over high-level objects or structures. This is the case for example in HPSG, in which complex feature-structures must first be built before constraints can be evaluated. Similarly, Optimality Theory also generates a set of structures (or candidate structures) and then uses constraints to filter this set. In our approach, any constraint can be evaluated at any time for any set of categories. Such evaluation, as explained above, dynamically adds new information: the satisfaction of a *selection* constraint instantiates the syntactic category it describes. But this instantiation is conceived almost as a side effect of evaluation: satisfying constraints does not rely on the knowledge of the upper-level category. In other words, the hierarchical information is no longer preponderant in the parsing process. This means that one can evaluate subsets of constraints, for example in the case of applications that only need NP recognition. In this approach, the conception of the relationship between grammar and language becomes very different from that of the generative paradigm. In the latter, a language is conceived as being generated by a grammar. In Property Grammars, a grammar is only used as a characterization device of the language properties.

As a consequence, instead of restricting the role of parsing to the evaluation of the input's grammaticality, we can propose a more flexible vision, in which a

parser’s output is the description of all the properties of the input. Concretely, such a description consists in the state of the constraint system after evaluation—in other words, the set of satisfied and violated constraints. We call such state a *characterization* of the input. In some cases, a characterization only contains satisfied constraints, but it can also be the case that some constraints can be violated, especially when parsing real life corpora. In most cases, such violations do not have consequences on the acceptability of the input.

One other formalism that shares the aims and some of the features of Property Grammars are Dependency Grammars (cf. [Tesnière59] and on this point [Mel’čuk88]), a purely equational system in which the notion of generation, or derivation between an abstract structure and a given string, is also absent. However, whereas in Dependency Grammars, as their name indicates, the property of dependence plays a fundamental role, in the framework we are considering it is but one of the many properties contributing to a category’s characterization. Perhaps the work that most relates to ours is Morawietz’s [Morawietz00], which implements deductive parsing [Shieber95] in CHR, and proposes different types of parsing strategies (including one for Property Grammars) as specializations of a general bottom-up parser. Efficiency however is not addressed beyond a general discussion of possible improvements, so while theoretically interesting, this methodology is in practice unusable due to combinatorial explosion. Moreover, it produces all properties that apply for each pair of categories without keeping track of how these categories are formed in terms of their subcategories, so there is no easy way to make sense of the output in terms of a complete analysis of a given input string.

The idea of throwing away the traditional, hierarchical parsing scheme in favour of a view of parsing which involves properties on categories rather than rewriting schemes first materialized in the 5P formalism (cf. [Bès99a], [Bès99b]). Preliminary work re. the advisability of a direct implementation of such an approach had yielded pessimistic results : [Blache95] showed that the mechanism of verification of a constraint system for syntactic analysis could be very expensive, given that the satisfiability of the system had to be verified in each stage. In the present work, however, we have moved beyond that obstacle by our analysis of property inheritance, which removes the need to recalculate all properties at each stage, allowing us to inherit at each stage most of the previous stage’s properties, while calculating only the minimally necessary new properties and updating the previous properties along the lines of our property inheritance analysis. Thus, our work has validated the model of property-centered parsing with respect to efficiency, while preserving the level of generality of this theory. In addition, a direct interpretation guarantees a better evolution of the initial system: it can better adjust to changes in the theory and to experimental stages.

We hope to have convincingly argued that direct renditions of flexible, constraint based parsing formalisms can be made to run efficiently while preserving a one to one correspondence between the conceptual and the representational levels, including for such non traditional formalisms as Property Grammars, in

which category inference does not depend on hierarchical or even constituency notions.

The representations allowed by our methodology, while extremely close to the computer-independent, conceptual representations of these formalisms, are directly executable, and moreover non-deterministic. This is satisfying with respect to logic programming's original aims of declarativeness and higher level expressiveness. Together with the advantages of this approach, we are able to even produce hierarchical depictions of the parse history of any category, including "incorrect" or incomplete ones. This is not important in itself, but is provided as an easy side effect, in the interest of historic comfort : we are all used to thinking in terms of parse trees or graphs, so showing a parse record in graph form may prove convenient to some users.

We have provided as well a simple mechanism by which a user can refine the notion of a category's characterization, by requiring some properties to be inflexibly checked, while relaxing others, for experimentation or suitability-to-measure purposes. We hope that these encouraging results can stimulate further research along these lines.

## Appendix – Some sample runs

N.B. Here we use total acceptability, i.e. we tolerate failure of any property, but show which are satisfied and which fail (see cat's two last arguments). Complete output is only shown in the first example, due to lack of space.

```
| ?- s3.  
  <0> le <1> livre <2> jaune <3>  
  
all(0,3),  
begin(-1,0),  
end(3,4),  
token(0,1,le),  
token(1,2,livre),  
token(2,3,jaune),  
cat(0,1,det,[sing,masc],det(le),[],[]),  
cat(1,2,n,[sing,masc],n(livre),[],[]),  
cat(1,2,sn,[sing,masc],sn(n(livre)),[],[]),  
cat(0,2,sn,[sing,masc],sn(det(le),n(livre)),  
  [prec(0,det,1,n,2),dep(0,det,1,n,2),  
  unicity(det,0,2),exige(n,det,0,2)],[]),  
cat(2,3,adj,[sing,masc],adj(jaune),[],[]),  
cat(2,3,sa,[sing,masc],sa(adj(jaune)),[],[]),  
cat(1,3,sn,[sing,masc],sn(n(livre),sa(adj(jaune))),  
  [prec(1,n,2,sa,3),dep(1,n,2,sa,3),unicity(n,1,3)],  
  [exige(n,det,1,3)]),  
cat(0,3,sn,[sing,masc],sn(det(le),n(livre),sa(adj(jaune))),
```

```

[prec(1,n,2,sa,3),dep(1,n,2,sa,3),unicity(n,1,3),
 prec(0,det,1,n,3),dep(0,det,1,n,3),unicity(det,0,3),
 exige(n,det,0,3)],[])?
yes

```

```

| ?- s4.
<0> le <1> le <2> livre <3>

```

```

cat(0,3,sn,[sing,masc],sn(det(le),det(le),n(livre)),
 [prec(1,det,2,n,3),dep(1,det,2,n,3),unicity(det,1,3),
 exige(n,det,1,3),prec(0,det,1,n,3),dep(0,det,1,n,3),
 exige(n,det,0,3)],[unicity(det,0,3)]) ?
yes

```

```

| ?- s5.
<0> livre <1> le <2>

```

```

cat(0,2,sn,[sing,masc],sn(n(livre),det(le)),
 [dep(0,n,1,det,2),unicity(det,0,2),exige(n,det,0,2)],
 [prec(0,det,1,n,2)]) ?

```

yes

## References

- [Abdennadher98] Abdennadher S. and Schütz H. (1998) “CHR: A flexible query language”, in proceedings of *Int. Conference on Flexible Query Answering Systems*, volume 1495 of LNCS, Springer-Verlag.
- [Barranco04] Barranco-Mendoza, A., Persaoud, D.R. and Dahl, V. (2004) “A property-based model for lung cancer diagnosis”, in proceedings of *8th Annual Int. Conf. on Computational Molecular Biology, RECOMB 2004*, San Diego, California (accepted poster).
- [Bès99a] Bès G & P. Blache (1999) “Propri/’et/’es et analyse d’un langage, in proceedings of *TALN’99*.
- [Bès99b] Bès G., P. Blache & C. Hagège (1999) “*The 5P Paradigm*”, rapport de recherche, GRIL/LPL.
- [Blache95] Blache P. & N. Hathout (1995) ”Constraint Logic Programming for Natural Language Processing”, in proceedings of *NLULP’95*.
- [Blache00] Blache P. (2000) “Constraints, Linguistic Theories and Natural Language Processing”, in *Natural Language Processing*, D. Christodoulakis (ed.), Lecture Notes in Artificial Intelligence, Springer.
- [Blache01] Blache P. & J.-M. Balfourier (2001) “Property Grammars: a Flexible Constraint-Based Approach to Parsing”, in proceedings of *IWPT-2001*.
- [Christiansen01] Christiansen, H. (2001) “CHR as grammar formalism, a first report”, Sixth Annual Workshop of the ERCIM Working Group on Constraints.
- [Christiansen02] Christiansen, H. (2002) *CHR Grammar web site*, <http://www.ruc.dk/~henning/chr>

- [Christiansen02] Christiansen H. & V. Dahl (2002), “Logic grammars for diagnosis and repair”, in proceedings of *ICTAI'02*.
- [Dahl04] Dahl V. and Voll K. (2004) “Concept Formation Rules: an executable cognitive model of knowledge construction”, in proceedings of *First International Workshop on Natural Language Understanding and Cognitive Sciences*, INSTICC Press.
- [Dahl97] Dahl, V., Tarau, P. and Li, R. (1997) “Assumption Grammars for Natural Language Processing”. in *Lee Naish (ed.) Proc. Fourteenth International Conference on Logic Programming*, pages 256-270, MIT Press, 1997.
- [Frühwirth98] Frühwirth T. (1998) “Theory and Practice of Constraint Handling Rules”, in *Journal of Logic Programming*, 37:1-3.
- [Gazdar85] Gazdar G., E. Klein, G. Pullum, I. Sag (1985), *Generalized Phrase Structure Grammar*, Blackwell.
- [Hecksher02] Hecksher T., S. Nielsen & A. Pigeon (2002) *A CHR model of the ancient Egyptian grammar*, Project report, Roskilde University, Denmark.
- [Mel'čuk88] Igor Mel'čuk (1988) “*Dependency Syntax*”, SUNY Press.
- [Morawietz00] (Morawietz 00) Morawietz F. (2000) “Chart parsing as constraint propagation”, in proceedings of *COLING-2000*.
- [Pollard94] Pollard C. & I. Sag (1994), *Head-driven Phrase Structure Grammars*, CSLI, Chicago University Press.
- [Prince93] Prince A. & Smolensky P. (1993), *Optimality Theory: Constraint Interaction in Generative Grammars*, Technical Report RUCSS TR-2, Rutgers Center for Cognitive Science.
- [Sag99] Sag I. & T. Wasow (1999), *Syntactic Theory. A Formal Introduction*, CSLI.
- [Shieber95] Shieber S., Y. Schabes & F. Pereira (1995) “Principles and implementation of deductive parsing”, in *Journal of Logic Programming*, 24(1-2):3-36, 1995.
- [SICSTUS03] Swedish Institute of Computer Science (2003) *SICStus Prolog user's manual, Version 3.10*, Most recent version available at <http://www.sics.se/isl>, 2003.
- [Tessière59] Tessière L. (1959) *El'ements de syntaxe structurale*, Klincksieck.