Parsing Natural Languages with CHR

FRANK MORAWIETZ
Seminar für Sprachwissenschaft, Universität Tübingen
Wilhelmstr. 109
72074 Tübingen, Germany
(*e-mail:* `frank@sfs.uni-tuebingen.de`)

PHILIPPE BLACHE
LPL-CNRS, Université de Provence
29 Avenue Robert Schuman
13621 Aix-en-Provence, France
(*e-mail:* `pb@lpl.univ-aix.fr`)

Abstract

In this paper, parsing as deduction and constraint programming are brought together to outline a procedure for the specification of constraint-based (chart) parsers. Extending the proposal in Shieber (1995) which provides a meta interpreter for several deduction systems, we show how to directly realize the inference rules for chart parsers as Constraint Handling Rules (Frühwirth, 1998) by viewing the items of a conventional chart parser as constraints and the constraint base as a chart. This allows the direct use of the constraint resolution process to parse sentences in diverse natural language formalisms such as minimalist grammars (Stabler, 1997; Stabler, 2001) or property grammars (Blache, 2000; Balfourier *et al.*, 2002).

1 Introduction: Parsing as Deduction

The parsing-as-deduction approach proposed in Pereira & Warren (1983) and extended in Shieber *et al.* (1995) and the parsing schemata defined as special deduction systems in Sikkel (1997) are well established parsing paradigms in computational linguistics. Their main strengths are their flexibility and level of abstraction concerning control information inherent in parsing algorithms. This allows for a uniform presentation and the comparison of a variety of parsing algorithms. Furthermore, they are easily extensible to more complex formalisms, e.g., augmented phrase structure rules or the ID/LP format.

Constraint Programming (CP) has been used in computational linguistics in several areas, for example in (typed) feature-based systems based on Oz (Smolka, 1995), or conditional constraints (Matiasek, 1994), or advanced compilation techniques (Götz & Meurers, 1997) or specialized constraint solvers (Manandhar, 1994). But to our knowledge, none of these approaches uses constraint programming techniques to implement standard (chart) parsing algorithms directly in a constraint system.

In this paper, we will bring these two paradigms together, thereby showing how to implement algorithms from the parsing-as-deduction scheme by viewing the parsing process as constraint satisfaction and propagation.

The core idea is that the items of a conventional chart parser are constraints on labeled links between the words and positions of an input string. Then the inference rules allow for the deduction of new constraints, again labeled and spanning parts of the input string, via constraint propagation. The resulting constraint store represents the chart which can be accessed to determine whether the parse was successful or to reconstruct a parse tree.

While this may seem a trivial observation it allows for a rapid and flexible method of implementation. The goal is not necessarily to build the fastest parser, but rather to build – for an arbitrary algorithm – a parser fast and perspicuously. The advantage of our approach compared to the one proposed in Shieber *et al.* (1995) is that we do not have to design a special deduction engine and we do not have to handle chart and agenda explicitly, i.e., the process can be used in any constraint-based formalism which allows for constraint propagation and therefore can be seamlessly integrated into existing applications.

The paper proceeds by reviewing the parsing-as-deduction approach and a particular way of implementing constraint systems, Constraint Handling Rules (CHR) as presented in Frühwirth (1998).[1] Then it shows how to implement several parsing algorithms very naturally with constraint propagation rules before concluding with an outlook on how to extend the technique to more advanced applications.

The paper summarizes and uses results presented in a series of other papers (Morawietz (1999; 2000a; 2000b; 2000c) and Morawietz and Blache (2000)).

1.1 Related Work

In a series of papers Henning Christiansen (2001; 2002b; 2002a) has described and developed CHRG, a grammar notation based on CHR. Similarly to the integration of DCGs into Prolog, the aim of CHRG is to provide an easy way of writing and parsing grammars within CHR. This aim is slightly different from ours. While we want to show that it is very easy to adapt existing parsing algorithms and implement them in CHR, Christiansen takes the approach of providing a "grammar formalism" in the widest sense. Obviously, this allows for the easy specification of grammars. The advantages of CHRG (partly derived form CHR) are that they are robust concerning partial results, that they are flexible enough to handle all sorts of formalisms and that they are implemented in such a way that the grammars can be parsed efficiently. The disadvantage is that the clean separation between the grammar and the logic of the parsing process we will present in our proposal is blurred.

The idea of using constraint propagation for parsing also appears in Meyer (2000). But instead of implementing the general deduction rules for parsing, the easier route of directly implementing grammar rules as CHR (propagation) rules is proposed. This makes for a more efficient implementation, but lacks the flexibility concerning

[1] CHR are provided as a package for SICStus Prolog and ECLiPSe.

the parsing mode and the parsing direction of our approach. Especially the ensuing problems with ambiguous grammars seem to create large problems for natural language systems.

1.2 Parsing as Deduction

Parsing in general is the process of analyzing the structure of an input by decomposing it into subparts according to some given grammar. If the process can be completed successfully, the input is a valid expression of the language defined by the grammar.

Therefore, to define how to parse a string, one needs the following ingredients:

- a way of initializing the parsing process, i.e., some statements about the acceptability of (parts of) strings which are used to start the parsing process;
- some rules which allow to derive new statements from existing ones, e.g., the initial ones, which actually define the parsing mode;
- a way of controlling the process, i.e., of deciding which statements to try next for possible rule applications;
- and a way of identifying a final statement, i.e., a statement which means that we successfully decomposed the string.

Parsing-as-deduction basically takes formulas of a logic and uses its axioms and deduction rules to drive the parsing process.

Obviously, parsing-as-deduction is an instantiation of the scheme given above. It derives statements about the acceptability of (parts of) strings from other such statements via inference rules, starts the process with axioms and stops when the possibilities for further deductions have been exhausted or a predefined goal has been reached. What does not have to be defined and which actually is seen as an advantage for the specification of parsing algorithms, is the control information. The search procedure is not relevant for the specification of a specific algorithm, but rather for its realization or implementation.

We will recall some basic definitions for convenience. The notations and the three basic algorithms are directly taken from Shieber *et al.* (1995).

As usual, strings w result from concatenation of symbols from some alphabet set Σ, i.e., $w \in \Sigma^*$. We refer to the decomposition of such a string into its alphabet symbols with indices. We fix this notation using $w = w_1 \dots w_n$ for the given input string. A *grammatical deduction system* or, in Sikkel's terminology a *parsing schema*, is defined as a set of deduction schemes and a set of axioms. These are given with the help of *formula schemata* which contain (syntactic) meta-variables which are instantiated with concrete terms on application of the rules. A deduction scheme R has the general form

$$\frac{A_1 \ \dots \ A_n}{C} \ \langle \text{ side conditions on } A_1 \dots A_n, C \ \rangle$$

where the A_i and C are formula schemata. The A_i are called antecedents and C the consequence. Note that the deduction schemes may refer to the string positions, i.e., the indices of the alphabet symbols of the input string, in their side conditions. We

$$\mathcal{G} = \langle\ N, T, S, P\ \rangle$$
$$NT = \{\text{S, NP, VP, V, PP, P, PN, Det, N1, N}\}$$
$$T = \{\textit{hit, John, dog, stick, with, the}\}$$

and P as given below:

S	$\longrightarrow$	NP VP	N1	$\longrightarrow$	N1 PP
VP	$\longrightarrow$	V NP	V	$\longrightarrow$	*hit*
VP	$\longrightarrow$	V NP PP	PN	$\longrightarrow$	*John*
PP	$\longrightarrow$	P NP	N	$\longrightarrow$	*dog*
NP	$\longrightarrow$	PN	N	$\longrightarrow$	*stick*
NP	$\longrightarrow$	Det N1	P	$\longrightarrow$	*with*
N1	$\longrightarrow$	N	Det	$\longrightarrow$	*the*

Fig. 1. Example Grammar: PP-attachment

say that such a scheme R *applies* if we have as antecedents a sequence of formulas $F_1, \ldots, F_k$, $k \geq n$ such that each F_i, $1 \leq i \leq n$ matches A_i under an appropriate substitution of terms for the meta-variables and if the side conditions are satisfied. For a given deduction system, the derivation of a formula F from assumptions $A_1, \ldots, A_p$ (in symbols: $A_1, \ldots, A_p \vdash F$) is defined to be a sequence of formulas $F_1, \ldots, F_q$ such that $F = F_q$ and each F_i, $1 \leq i \leq q$, is

- either an instance of an axiom,
- one of the A_j, $1 \leq j \leq p$,
- or the result of applying a deduction scheme with $F_i = C$ such that there exist the necessary antecedents $A_{i_1}, \ldots, A_{i_k}$, $i_1, \ldots, i_k \leq i$.

To improve the performance of the implementation of a parser, one also wants to record work which has been done such that recomputations of (partial) results can be avoided. To simply store recognized substrings is not enough since one cannot differentiate between for example two identical occurrences of substrings. Therefore the statements are enriched by indexing the category recognized, i.e., the terminal or nonterminal symbol labeling a node in the parse tree, with the position(s) spanned by that category. These statements are then stored in a well-formed substring table, also called a chart. The chart does not play a direct role for the specification of grammatical deduction systems but since we will present examples below, it is necessary to introduce the complex formulas or statements used there.

Unless specified differently, we assume that we are given a context-free grammar $\mathcal{G} = \langle\ N, T, S, P\ \rangle$ with nonterminals N, terminals T, start symbol S and set of productions P.[2] Each production is of the form $A \longrightarrow \alpha$ with $A \in N$, $\alpha \in (N \cup T)^*$.

As a running example, we will use the simple PP-attachment grammar given in Fig. 1. It is left to the reader to calculate example derivations for the three example algorithms for a sentence such as *John hit the dog with the stick.*

[2] For Earley's algorithm we also assume a new start symbol S' which is not in N.

Table 1. *Parsing algorithms as Grammatical Deduction Systems*

	Bottom-Up	**Top-Down**	**Earley**
Items	$[j, \alpha \bullet]$	$[j, \bullet \beta]$	$[i, j, A, \alpha \bullet \beta]$
Axiom	$[0, \bullet]$	$[0, \bullet S]$	$[0, 0, S', \bullet S]$
Goal	$[n, S \bullet]$	$[n, \bullet]$	$[0, n, S', S \bullet]$
Scan	$\frac{[j, \alpha \bullet]}{[j+1, \alpha w_{j+1} \bullet]}$	$\frac{[j, \bullet w_{j+1}\beta]}{[j+1, \bullet \beta]}$	$\frac{[i, j, A, \alpha \bullet w_{j+1}\beta]}{[i, j+1, A, \alpha w_{j+1} \bullet \beta]}$
Predict		$\frac{[j, \bullet B\beta]}{[j, \bullet \gamma\beta]} \langle B \longrightarrow \gamma \rangle$	$\frac{[i, j, A, \alpha \bullet B\beta]}{[j, j, B, \bullet \gamma]} \langle B \longrightarrow \gamma \rangle$
Complete	$\frac{[j, \alpha\gamma \bullet]}{[j, \alpha B \bullet]} \langle B \longrightarrow \gamma \rangle$		$\frac{[i, k, A, \alpha \bullet B\beta] \quad [k, j, B, \gamma \bullet]}{[i, j, A, \alpha B \bullet \beta]}$

Almost all of the parsing systems used in this paper are then defined by specifying a class of items, a set of axioms, a set of inference rules and a subset of the items, the goals or final items. For better readability, we follow Shieber *et al.* in using the familiar dotted items.

Notational conventions are: n for the length of the string to be parsed; $A, B, C, \ldots$ for arbitrary formulas or nonterminals; $a, b, c, \ldots$ for terminals; ε for the empty string and $\alpha, \beta, \gamma, \ldots$ for strings of terminals and nonterminals. Formulas used in parsing will also be called items or edges.

The three example algorithms we will use to illustrate our technique can now be presented as grammatical deduction systems as given in Tab. 1 (taken from Shieber *et al.* (1995)). In the table i, j, k are always between 1 and n.

Although we assume familiarity with these three basic algorithms, we will briefly comment on the intended interpretations of the respective items. In the case of the naive bottom-up algorithm, the items are of the form $[j, \alpha \bullet]$ with the interpretation that we have a stack α of items we recognized reaching up to position j (which implies that we are still looking forward to parsing the substring starting at j). In the case of the top-down algorithm, the items $[j, \bullet \beta]$ mean that we are looking for categories β starting at position j (implying that we already recognized the string up to that position). The items in Earley's algorithm $[i, j, A, \alpha \bullet \beta]$ are to be interpreted that we recognized a substring from i to j with the rule $A \longrightarrow \alpha\beta$ having already found α, but looking for categories β.

1.3 Constraint Handling Rules

Constraint programming is an elegant, declarative way of solving problems with higher-level programming languages. It represents a generalization of logic programming in the sense that it allows the (logical) specification of a problem on an arbitrary domain and tries to solve it with a combination of inference and search.

A complete introduction to constraint programming can be found for example in Marriott & Stuckey (1998).

There are several constraint programming environments available. The most recent and maybe the most flexible is the Constraint Handling Rules (CHR) package included in both SICStus Prolog and ECLiPSe (Intelligent Systems Laboratory, 1995; Frühwirth, 1998). All of these CP systems provide mechanisms for solving constraints. They maintain a constraint base or store which is continually monitored for possible rule applications, i.e., whether there is enough information present to successfully use a rule to simplify constraints or to derive new constraints. Whereas usually one deals with a fixed constraint domain and a specialized solver, CHR is an extension of the Prolog language which allows for the specification of user-defined constraints and arbitrary solvers. The strength of the CHR approach lies in the fact that it allows for multiple (conjunctively interpreted) heads in rules, that it is flexible and that it is tightly and transparently integrated into the Prolog engine.

In CHR constraints are just distinguished sets of (atomic) formulas. CHR allow the definition of rule sets for constraint solving with three types of rules: Firstly simplification rules (symbol <=>) which replace a number of constraints in the store with new constraints; secondly propagation rules (symbol ==>) which add new constraints to the store in case a number of constraints is already present; and thirdly "simpagation" rules (symbol <=> in combination with a \ in the head of the rule) which replace only those constraints with new ones which are to the right of the backslash.

To improve efficiency, rules can have guards and pragma declarations. A guard (separated from the rest of the body by a |) is a condition which has to be met before the rule can be applied.[3] A pragma declaration affects the way the rule is compiled. Our approach will only introduce the *passive* declaration, so we ignore the other possibilities. Declaring a constraint in the head to be passive simply means that this constraint will not trigger the rule on its own.

We cannot go into the details of the formal semantics of CHR here. The interested reader is referred to Frühwirth (1998) and references therein. Since we will refer back to it let us just note that logically, simplification rules are equivalences and propagation rules are implications if their guard is satisfied. Simpagation rules are special cases of simplification rules. Soundness and completeness results for CHR are available (Abdennadher *et al.*, 1996; Abdennadher, 1998).

2 Parsing as Constraint Propagation

The basic observation which turns parsing-as-deduction into constraint propagation is simple: items of a chart parser are just special formulas which are used in an inference process. Since constraints in constraint programming are nothing

[3] A more careful distinction separates *ask* and *tell* guards. Ask guards have to be entailed by the constraint store whereas tell guards only have to be consistent with it (Saraswat, 1993). This is also part of CHR, but not used in our approach.

Table 2. *Parsing systems as CHR programs: Items, Axioms & Goals*

	Bottom-Up	**Top-Down**	**Earley**
Items	`edge(X,N)`	`edge(X,N)`	`edge(A,Alpha,Beta,I,J)`
Axiom	`edge([],0)`	`edge([s],0)`	`edge(sprime,[],[s],0,0)`
Goal	`edge([s],Len)`	`edge([],Len)`	`edge(sprime,[s],[],0,Len)`

but atomic formulas and constraint handling rules nothing but inference rules, the connection is immediate.

In more detail, we will present in this section how to implement the three parsing algorithms given in Tab. 1 in CHR and discuss the advantages and drawbacks of this approach. Since CHR are integrated in SICStus Prolog, we will present constraints and rules in Prolog notation, i.e., strings starting with uppercase letters stand for variables and words with lowercase ones for atoms.

We use the following two types of constraints. The constraints corresponding to the items will be called `edge` constraints, see Tab. 2. They have two arguments in case of the two naive algorithms and five in the case of Earley's algorithm, i.e.,

```
edge(X,N)
```

means in the case of the bottom-up algorithm that we have recognized a list of categories `X` up to position `N`, in the case of the top-down algorithm that we are looking for a list of categories `X` starting at position `N` and in the case of Earley's algorithm

```
edge(A,Alpha,Beta,I,J)
```

stands for the fact that we found a substring from `I` to `J` by recognizing the list of categories `Alpha`, but we are still looking for a list of categories `Beta` to yield category `A`. The second constraint `word/2` is only used for efficiency. Since in grammar rules we do not use the lexical items directly but rather their categories, we do not use edges to represent that we found a word, but rather the `word` constraint, as will become apparent in the scanning rules. This avoids using constraints on the input *words* (compared to their category) in the other inferences needlessly.

The grammars are given as Prolog facts, lexical items as

```
lex(Word,Category)
```

and grammar rules as

```
rule(RHS,LHS)
```

where `RHS` is a list of categories representing the right hand side and `LHS` is a single category representing the left hand side of the rule.

The resulting algorithms are then simple to implement by specifying the inference

Table 3. *Parsing systems as CHR programs: Inference Rules*

	Scan
Bottom-Up	`edge(Stack,N), word(N,Cat-_Word) ==>`
	`N1 is N+1,`
	`edge([Cat|Stack],N1).`
Top-Down	`edge([Cat|T],N), word(N,Cat-_Word) ==>`
	`N1 is N+1,`
	`edge(T,N1).`
Earley	`edge(A,Alpha,[Cat|Beta],I,J), word(J,Cat-_Word) ==>`
	`J1 is J+1,`
	`edge(A,[Cat|Alpha],Beta,I,J1).`
	Predict
Top-Down	`edge([LHS|T],N) ==>`
	`setof(RHS, rule(RHS,LHS), List) |`
	`post_td_edges(List,T,N).`
Earley	`edge(_A,_Alpha,[B|_Beta],_I,J) ==>`
	`setof(Gamma, rule(Gamma,B), List) |`
	`post_ea_edges(List,B,J).`
	Complete
Bottom-Up	`edge(Stack,N) ==>`
	`setof(Rest-LHS, split(Stack,Rest,LHS), List) |`
	`post_bu_edges(List,N).`
Earley	`edge(A,Alpha,[B|Beta],I,K)#Id, edge(B,Gamma,[],K,J) ==>`
	`edge(A,[B|Alpha],Beta,I,J)`
	`pragma passive(Id).`
	Absorb
Bottom-Up	`edge(L,N) \ edge(L,N) <=> true.`
Top-Down	`edge(L,N) \ edge(L,N) <=> true.`
Earley	`edge(A,Alpha,Beta,I,J) \ edge(A,Alpha,Beta,I,J) <=> true.`

rules as constraint propagation rules, the axioms and the goal items. A summarization is presented in Tab. 3.

It is obvious how to link the code for the axioms and goals given to the definitions in Tab. 1. Let us consider Earley's algorithm for a closer look as to what is going on in the CHR propagation rules.

In principle the inference rules are translated into CHR in the following way: The antecedents are transformed into constraints appearing in the head of the propagation rules, the side conditions into the guard and the consequence is posted in the body.[4]

In the scanning step, we can move the head of the list of categories we are looking

[4] Note that calling a CHR constraint as a Prolog goal means to insert it into the constraint store such that it becomes available for constraint resolution steps.

for to those we already recognized in case we have an appropriately matching edge and word constraint in our constraint store. The result is posted as a new edge constraint with the positional index appropriately incremented.

The prediction step is more complex. There is only one head in a rule, namely an edge which is still looking for some category to be found. If one can find rules with a matching LHS, we collect all of them in a list and post the appropriate fresh edge constraints for each element of that list (predicate `post_ea_edges/3` which posts edges of the following kind:

```
edge(LHS,[],RHS,J,J).
```

The collection of all matching rules in a call to `setof/3` is necessary since CHR are a committed choice language. One cannot enumerate all solutions via backtracking. If there are no matching rules, i.e., the list of RHSs we found is empty, the corresponding tests avoid vacuous predictions and therefore nontermination of the predictor.

Lastly, the completion step is a pure propagation rule which translates literally. The two antecedents are in the head and the consequence in the body with appropriate instantiations of the positional variables and with the movement of the category recognized by the passive edge from the categories to be found to those found. Additionally, the rule contains a pragma declaration on the first head which is declared to be *passive*. While we cannot go into detail here, this declaration leads to the fact that this rule is only tried for inference if a constraint matching the nonpassive head is posted. More on pragma declarations can be found in Frühwirth (1998).

In the table there is one more rule, called an absorption rule. It discovers those cases where we posted an edge constraint which is already present in the chart and simply absorbs the newly created one. We come back to this point below.

Note that we do not have to specify how to insert edges into either a chart or into an agenda. The chart and the agenda are in fact represented by the constraint store and therefore built-in. We do not need a specialized deduction engine as was necessary for the implementation described in Shieber *et al.* (1995). In fact, the utilities needed are extremely simple, see Fig. 2.

All we have to do for parsing (predicate `parse/1`) is to post the axiom[5] and on traversal of the input string to post the word constraints according to the lexicon of the given grammar. Then the constraint resolution process with the inference rules will automatically build a complete chart. The call to `report/1` will just determine whether there is an appropriate edge with the correct length in the chart and print that information to the screen.

Coming back to the issues of chart and agenda: the constraint store functions as chart and agenda at the same time since as soon as a constraint is added all rules are tried for applicability. If none apply, the edge will remain dormant until another constraint is added which triggers a rule together with it.[6] So, the parser

[5] `axiom/0` is a predicate which just calls the edge(s) defined in Tab. 3.
[6] Another way to "wake" a constraint is to instantiate any of its variables in which case it will

```
parse(L) :-
         axiom,
         post_constraints(L, 0, Length),
         report(Length).

post_constraints([], Length, Length).
post_constraints([Word|String], InLength, Length) :-
         setof(Cat, lex(Word, Cat), Cats),
         post_words(Cats, InLength, Word),
         NewLength is InLength + 1,
         post_constraints(String, NewLength, Length).

post_words([],_Position,_Word).
post_words([Cat|Cats], Position, Word):-
        word(Position, Cat-Word),
        post_words(Cats, Position, Word).
```

Fig. 2. The utilities for CHR-based deductive parsing

works incrementally. It recursively tries all possible inferences for each constraint added to the store before continuing with the posting of new constraints from the `post_constraints/3` predicate. The way this predicate works at the moment is to traverse the string from left-to-right. It is trivial to alter the predicate such as to post the constraints from right-to-left or any arbitrary order chosen. This can be used to easily test different parsing strategies.

The testing for applicability of new rules also has a connection with the absorption rules. We absorb the newer edge since we can assume that all possible propagations have already been done with the old, identical edge constraint so that we can safely throw the other one away.

As an example for the resulting chart, the output of an Earley-parse for *John hit the dog with the stick* assuming the grammar given in Fig. 1 is presented in Fig. 3.

As one can see, the entire constraint store is printed to the screen after the constraint resolution process stops producing new edges. And, furthermore, the order of the constraints actually reflects the order of the construction of the edges, i.e., the chart constitutes a trace of the parse at the same time. Naturally the given string was ambiguous but only a single solution is visible in the chart. This is due to the fact that we only did recognition. No explicit parse was built which could have differentiated between the two solutions. It is an easy exercise to either write a predicate to extract all possible parses from the chart or to alter the edges in such a way that an explicit parse tree is built during parsing.

By using a built-in deduction engine, one gives up control of its efficiency. As it turns out, this CHR based approach is slower than the specialized engine developed and provided by Shieber *et al.* (1995) by about a factor of 2, e.g., for a six word

be matched against the rules again. Since all our constraints are ground, this does not play a role here.

```
| ?- parse([john, hit, the, dog, with, the, stick]).

Input recognized.

 word(0,pn-john),                   edge(n1,[n],[],3,4),
 word(1,v-hit),                     edge(np,[n1,det],[],2,4),
 word(2,det-the),                   edge(vp,[np,v],[],1,4),
 word(3,n-dog),                     edge(s,[vp,np],[],0,4),
 word(4,p-with),                    edge(sprime,[s],[],0,4),
 word(5,det-the),                   edge(vp,[np,v],[pp],1,4),
 word(6,n-stick),                   edge(pp,[p],[np],4,5),
 edge(sprime,[],[s],0,0),           edge(np,[],[det,n1],5,5),
 edge(s,[],[np,vp],0,0),            edge(np,[],[pn],5,5),
 edge(np,[],[det,n1],0,0),          edge(np,[det],[n1],5,6),
 edge(np,[],[pn],0,0),              edge(n1,[],[n],6,6),
 edge(np,[pn],[],0,1),              edge(n1,[],[n,pp],6,6),
 edge(s,[np],[vp],0,1),             edge(n1,[n],[pp],6,7),
 edge(vp,[],[v,np],1,1),            edge(pp,[],[p,np],7,7),
 edge(vp,[],[v,np,pp],1,1),         edge(n1,[n],[],6,7),
 edge(vp,[v],[np,pp],1,2),          edge(np,[n1,det],[],5,7),
 edge(np,[],[det,n1],2,2),          edge(pp,[np,p],[],4,7),
 edge(np,[],[pn],2,2),              edge(vp,[pp,np,v],[],1,7),
 edge(vp,[v],[np],1,2),             edge(s,[vp,np],[],0,7),
 edge(np,[det],[n1],2,3),           edge(sprime,[s],[],0,7),
 edge(n1,[],[n],3,3),               edge(n1,[pp,n],[],3,7),
 edge(n1,[],[n,pp],3,3),            edge(np,[n1,det],[],2,7),
 edge(n1,[n],[pp],3,4),             edge(vp,[np,v],[],1,7),
 edge(pp,[],[p,np],4,4),            edge(vp,[np,v],[pp],1,7) ?
```

Fig. 3. Chart for *John hit the dog with the stick*

sentence and a simple grammar the parsing time increased from 0.01 seconds to 0.02 seconds on a LINUX PC running SICStus Prolog. This factor was preserved under 5 and 500 repetitions of the same parse. This may or may not be seen as a serious problem. However, there are two remarks we want to make. Firstly, speed was not the main issue in developing this setup, but rather simplicity and ease of implementation. And secondly it seems that the implementation of the attributed variables package of SICStus Prolog which is the basis for the CHR package, is too general to be maximally efficient. Gerald Penn (Penn, 1999; Penn, 2000) reports two instances of special applications where the efficiency of the implementation was improved by using specific predicates and not the full library. On implementing co-routining in a typed feature language built on top of Prolog, it turned out that a speed-up by a factor of about 50 could be achieved by using `when/2`, and on implementing typed feature structure unification that, by using some undocumented internal predicates directly rather than `get_atts/2` and `put_atts/2`, a speed-up of a little over a factor of 28 could be achieved with the ALE (Carpenter & Penn, 1998) Head-Driven Phrase Structure (HPSG) grammar. If we can apply similar techniques, the implementation may be able to improve on its performance.

The Shieber *et al.* paper contains more advanced parsing algorithms which can be transferred into the CHR setting. To transfer the parsing algorithm for unification grammars we simply include the necessary unifications explicitly into the body of the rules. Below we give the completion rule for unification grammars defined in such a way that the unification for the relevant categories `B` and `B2` computes the result `B1` and an explicit most general unifier `Sigma` which is then applied to the other categories `A`, `Alpha` and `Beta`.

```
edge(A,Alpha,[B|Beta],I,K), edge(B2,Gamma,[],K,J) ==>
          unify(B,B2,B1,Sigma),
          apply(Sigma,A,A1),
          apply(Sigma,Alpha,Alpha1),
          apply(Sigma,Beta,Beta1)       |
          edge(A1,[B1|Alpha1],Beta1,I,J).
```

Even the Shieber *et al.* approach to Combinatory Categorial Grammars can be transferred seamlessly. Consider for example forward application:

$$\frac{[X/Y, i, k] \quad [Y, k, j]}{[X, i, j]}$$

This results in the most trivial completion rule:

```
edge(X/Y,I,K), edge(Y,K,J) ==>
          edge(X,I,J).
```

We think that this short discussion shows convincingly how trivially the Parsing-as-Deduction algorithms can be transferred.

To sum up this section, the advantages of the approach lie in its flexibility and its availability for rapid prototyping of different parsing algorithms since it avoids any explicit handling of chart or agenda. While we used the examples from the Shieber *et al.* article, one can also implement all the different parsing schemata presented as deduction schemes in Sikkel (1997). This also includes advanced schemes such as left-corner or head-corner parsing, the refined Earley-algorithm proposed by Graham *et al.* (1980), or (unification-based) ID/LP parsing as defined in Morawietz (1995) (we just have to include guards to check for precedence, see Fig. 4), or any improved version of any of these. Furthermore, because of the logical semantics of CHR with their soundness and completeness, all correctness and soundness proofs for the algorithms can be directly applied to this constraint propagation proposal. The main disadvantage of the proposed approach certainly lies in its apparent lack of efficiency. One way to overcome this limitation is discussed in the next section.

3 Extensions of the Basic Technique

There are many directions the extensions of the presented technique of CHR parsing might take. Firstly, one might consider parsing of more complicated grammars

```
edge(A,Alpha,[B|Beta],I,K), edge(B,Gamma,[],K,J) ==>
          precedes(Alpha,B),
          precedes(B,Beta)   |
          edge(A,[B|Alpha],Beta,I,J).
```

Fig. 4. The completion step for ID/LP grammars

compared to the CF ones which were assumed so far. Following Shieber *et al.*, one can consider unification-based grammars or tree adjoining grammars. Since we think that the previous sections showed that the Shieber *et al.* approach is transferable in general, the results they present are applicable to our proposal as well. Instead, we want to consider parsing minimalist grammars (Chomsky, 1995) as defined in recent work by Stabler (1997; 2001)[7] and property grammars as defined in, e.g., Blache (2000) or Balfourier *et al.* (2002).

3.1 Minimalist Parsing

We cannot cover the theory behind the derivational minimalism approach presented in Stabler's papers in any detail. Very briefly, lexical items are combined with each other by a binary operation *merge* which is triggered by the availability of an appropriate pair of clashing features, here noted as `cat(C)` for Stabler's categories `c` and `comp(C)` and `spec(C)` for `=c`.[8] Furthermore, there is a unary operation *move* which, again on the availability of a pair of clashing features (e.g., `-case`, `+case`), triggers the extraction of a (possibly trivial) subtree of the tree under consideration and its merging in at the root node. On completion of these operations the clashing feature pairs are removed (in the parlance of minimalist linguistics: *checked*). The lexical items are of the form of linked sequences of trees. Accessibility of features is defined via an order on the nodes in this chain of trees.[9] A parse is acceptable if all features have been checked, apart from one category feature which spans the length of the string. The actual algorithm works naively bottom-up and since the operations are at most binary, the algorithm is CYK-based. This is the most obvious achievement of Stabler's proposal – minimalist parsing can be reduced to CYK parsing.

An initial edge or axiom in this minimalist parsing system cannot simply be assumed to cover the part of the string where it was found since it could have been the result of a movement operation. So the elements of the lexicon which will have to be moved (they contain a phrasal movement trigger `-X`) actually have the positional

[7] The basic code for the implementation underlying the paper was kindly provided by Ed Stabler. Apart from the implementation in CHR, all the rest is his work and his ideas.

[8] The split of the *merge* rule between merging complementizers and specifiers allows for a more perspicuous presentation of the inference rules, as one can see in Tab. 5.

[9] We ignore certain aspects for simplicity. The reader is referred to the original literature for the full details.

Table 4. *A CHR-based minimalist parser: Items, Axioms and Goals*

Items	`edge(I, J, Chain, Chains)`
Axiom	`edge(I, I, Chain, Chains)`
	$\langle$`[Chain|Chains]` $\longrightarrow \varepsilon\rangle$, `I` a variable
	`edge(I, I+1, Chain, Chains)`
	$\langle$`[Chain|Chains]` $\longrightarrow w_{i+1}\rangle$
	and there is no `-X` in `[Chain|Chains]`
	`edge(J, J, Chain, Chains)`
	$\langle$`[Chain|Chains]` $\longrightarrow w_{i+1}\rangle$, `J` a variable
	and there is a `-X` in `[Chain|Chains]`
	and `I` and `I+1` occur in `[Chain|Chains]`
Goal	`edge(0, Length, [cat(C)], [])`

indices instantiated in the last of those features appearing. All other movement triggers and the position it will be base generated are assumed to be traces and therefore empty. Their positional markers are identical variables, i.e., they span no portion of the string and one does not know their value at the moment of the construction of the axioms. They have to be instantiated during the minimalist parse. The same is true for the empty elements appearing explicitly in the lexicon.

As an example consider the set of items as defined by the axioms, see Tab. 3.1. The general form of the items is such that we have the indices first, then we separate the chain of trees into the first one and the remaining ones for better access. The axioms post edges corresponding to the lexical items. As an example for the actual edges and to illustrate the discussion about the possibly variable string positions in the edges, consider the recognition of the string *believe it* with a remnant movement analysis. The lexical item for *it* takes the form:

```
lex(it,I,[(K,K)=[cat(d),-case(I,J)]]) :- J is I+1.
```

where `I=1` if we instantiate it as an axiom. The resulting edge will be `edge(K, K, [cat(d),-case(1,2)]], [])`. We know that *it* has been moved to cover positions `1` to `2`, but we do not know (yet) where it was base generated.

These non-ground edges actually have an effect on the inference rules, see Tab. 5. Apart from this complication, they pretty much reflect a bottom-up approach to merging and moving lexical items. Therefore we will not discuss the details of how the conditions in the guards achieve to test for the correct configurations. The interested reader is referred to Stabler (2001).

We will briefly discuss the Prolog predicates `unif_vars/4`, `test_vars/4` and `test_vars/8`.[10] In the Shieber at al. system, we compared ground positional indices to find matching edges. Since CHR are sensitive to the instantiation of variables, it is not possible to do the same here.[11] The above mentioned predicates basically test

[10] Since we are dealing with the positions, the arguments always are members of a pair stemming from one edge.

[11] CHR have a compile time option which allows the disabling of this sensitivity. Unfortunately,

Table 5. *A CHR-based minimalist parser: Inference Rules*

Move

```
edge(I, J, [+(X)|RestHead], Chains0) ==>
      test_vars(I, J, P, Q),
      Link =.. [X,K,P],
      select([-Link|RestLinks], Chains0, Chains1),
      add(RestLinks, Chains1, Chains),
      linkStart(RestHead, NewHead, K, Q, L, M) |
   edge(L, M, NewHead, Chains).
```

Merge$_{Comp}$

```
edge(I, J, [comp(X)|RestHead], Chains0),
           edge(K, L, [cat(X)|RestComp], Chains1) ==>
      test_vars(I, J, K, L, P, Q, R, S),
      check(RestHead, NewHead, P, S, A, B,
            RestComp, Chains0, Chains1, Chains) |
   edge(A, B, NewHead, Chains).
```

Merge$_{Spec}$

```
edge(I, J, [cat(X)|RestSpec], Chains0),
           edge(K, L, [spec(X)|RestHead], Chains1) ==>
      test_vars(I, J, K, L, P, Q, R, S),
      check(RestHead, NewHead, P, S, A, B,
            RestSpec, Chains0, Chains1, Chains) |
   edge(A, B, NewHead, Chains).
```

Absorption

```
edge(I, J, L, Ls) \ edge(P, Q, L, Ls) <=>
      unif_vars(I, J, P, Q) |
      true.
```

for their arguments to be variables. If they are, new ones are returned such that the relationship (identity) between the members of pairs are preserved. If they are ground, they are unified. It is obvious that the price we pay is a decrease in efficiency since the CHR system actually has to enter the guards to determine applicability of the rules.

We cannot go into any detail how the actual parser work. Nevertheless, the implementation shown in Tab. 5 demonstrates how easily one can implement even complicated parsers without the need for extra utilities to deal with chart and agenda. Furthermore, it offers the possibility to implement feature cancellation as real constraint solving which opens up an interesting new perspective for processing.

doing so does not help since CHR are a committed choice language and therefore the unification cannot be undone via backtracking.

3.2 Property Grammar Parsing

In order to motivate the use of a new formalism, we highlight some limitations in the use of constraints in the generative paradigm. In particular, one of the main problems comes from the fact that linguistic information is expressed by means of rules (or rule schemata). In this case linguistic constraints are expressed over structures instead of linguistic objects. This limits the parsing process to a generate-and-test approach.

Constraints in linguistic theories generally play only a local role:[12] they are used to control the instantiation of linguistic objects (the categories) and represent relations between these objects (or their components). Then, a clear distinction between the objects and their properties becomes possible, which favors declarativity. For example, in a theory like GPSG (Gazdar *et al.*, 1985), one can find constraints at different levels of the description: linear precedence, feature cooccurrence restriction or universal instantiation principles. But not all the information is represented with constraints and during a parse, one has to select a local tree first and then verify that this tree satisfies the different constraints. This is typically a passive use of constraints.

The problem is different in HPSG in which most of the information is actually represented by means of constraints (cf. Sag (1999) and especially for an implementation (Blache & Paquelin, 1996; Meurers & Minnen, 1998)). However, the hierarchical information remains predominant in the sense that verification of constraints can only be applied to an implicit local tree. The verification of the principles for example requires the construction of a hierarchical structure containing a mother and its daughters. This does not mean that such a structure has to be fully specified. Some delaying mechanisms can obviously be applied and active constraints can be used locally. But in this case, we can globally characterize the use of constraints for the parsing process as passive and parsing does not amount to pure constraint resolution.

One of the reasons for these problems is the generative interpretation of the relation between grammars and languages. In this case, the notion of derivation is central and parsing an input consists in finding a derivation generating it. Such a conception is revealed in the approaches cited above with the role played by the local trees (which usually amounts to one derivation step). Parsing is then conceived as a derivation mechanism controlled by constraints (often reduced to unification).

This certainly stems from the fact that licensing of constraints is implemented as improved versions of the generate-and-test paradigm to ensure some sort of efficiency (and sometimes even termination).

[12] Constraints in this case are not active all along the parsing process, but only for specific structures built during the parse. Such constraints are local because their scope is reduced according to the knowledge of the constrained structure.

3.2.1 Some Desiderata for Constraint-Based NLP

An optimal use of constraints, both from the linguistic and the computational point of view, should meet some requirements. We present in this section some basic desiderata for a genuine constraint-based approach.

One important property concerns knowledge representation: as indicated before, the problem with classical approaches partly comes from the fact that information is represented my means of rules. Defining the parsing process in terms of constraint resolution is coupled with the representation of all linguistic information with constraints. In this case, a grammar can be conceived as an actual constraint system and constraints are stipulated over objects, not over structures. In our proposal, all constraints are expressed over categories (which do not contain any hierarchical information which is a difference to HPSG). This aspect has an important consequence on the declarativity of the approach. Indeed, it allows a clear distinction between the representation of information (in particular the form of the objects) and the mechanisms for calculating with them.

Another important point is the fact that, insofar as the set of constraints constitutes a system, all constraints are at the same level. Obviously, some heuristics can rely on a certain hierarchization (for example by means of weights) or on the control of constraint relaxation. But this does not alter the fact that the constraint system has to be taken as a whole. This characteristic is thus in opposition with the way of using constraints in Optimality Theory (see Prince & Smolensky (1993)). In this theory, ranking constraints is an essential element of the grammar. No information can be computed without such a hierarchy (which then implicitly contains linguistic information).

Constraint homogeneity also constitutes an important property for constraint-based theories: a constraint must represent homogeneous information. One constraint encodes one type of syntactic information and only one. In this way, we systematize the idea initiated with GPSG (with the distinction between ID and LP information) which consists in representing the different pieces of information of the grammar separately. We need to distinguish different types of constraints, each one representing a specific part of linguistic information. As a side effect, this eliminates implicit information. Moreover, each type of constraint becomes linguistically motivated.

The last requirement concerns the notion of grammaticality. One interesting property of constraint solving is the possibility of building an approximate answer when no exact solution can be found. In a constraint-base approach, such an approximation is nothing but the state of the constraint system after verifying its satisfiability for a given input. Let's see what does this mean for parsing. In classical approaches, finding a solution (i.e., parsing) consists in associating a syntactic structure with a given input. For generative methods, this involves finding a derivation from a distinguished symbol toward this input. Doing this, we answer the question of grammaticality: an input is grammatical iff a solution (a derivation) can be found. However, this seems to us a very restrictive concept of analysis. Indeed, a more relevant question concerning parsing (especially unrestricted texts or more generally real natural language input) should not verify grammaticality of an input, but rather provide as much information about it as possible. In this view, any kind of input can be parsed, whatever its form: it is an intrinsically robust way of defining the parsing problem. Therefore we propose to replace the notion of grammaticality with that of *characterization* which is more general: a characterization is the state of the constraint system for a given input, consisting of satisfied and violated constraints. The only difference between the characterization of ill-formed and well-formed inputs is that the former contains at least one violated constraint whereas the later only contains satisfied ones. Characterization subsumes grammaticality in the sense that verifying grammaticality consists in building a characterization in which no constraint violation is allowed.

3.2.2 Presentation of Property Grammars

The *Property Grammars* formalism, hereafter PG (cf. Blache (2001)), implements these requirements: declarativity, information homogeneity, satisfiability. Compared to other linguistic formalism, it does not rely on a two-stage process consisting in building first a structure and then applying constraints to it. PG proposes the representation of all information by means of constraints (also called properties). This conception relies on the observation that what linguists usually call a constraint corresponds to an explicit and homogeneous piece of information. It is the case with linear precedence (even if GPSG uses the term of *statement*). PG simply systematizes this move initiated with GPSG which consisted in representing all information separately. In the end, a PG grammar forms a constraint system which contains all necessary information and which does not employ any other mechanism than constraint resolution.We insist on this point which is of deep importance: no generative function, be it explicit as in the Optimality Theory (the GEN function) or in Constraint Dependency Grammars or implicit as in HPSG is used here.

One of the most important aspect in the specification of a constraint-based approach, lies in the definition of the objects to be constrained. It is possible to stipulate constraints over objects of any level. Usually, constraints in linguistics are defined over high-level objects such as local trees in DCG or linguistic signs in HPSG. However, the consequence is that we need first to build these objects before being able to verify a constraint. The interest in stipulating linguistic information in terms of constraints over low-level objects is that they can be verified at the very beginning of the process. An affectation in this case can be chosen even before any other treatment. In PG, constraints are expressed over categories. As usual in constraint-based approaches, a category is a set of attribute-value features pairs.

PG is not a lexicalized theory in the sense that the lexicon contains little syntactic information. A lexical item is formed by three main features which are LABEL, SYNT and SEM. In the same way as typed feature structure approaches do, the form of the structure may vary according to the category, some features being appropriate to all categories, some other being more specific. The following example illustrates a possible shape for a verbal structure:

$$\begin{bmatrix} \text{LABEL} & \\ \text{SYNT} & \begin{bmatrix} \text{AGR} \\ \text{VALENCE} \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{REL} \\ \text{AGENT} \\ \text{OBJECT} \end{bmatrix} \end{bmatrix}$$

In PG, all properties involve such categories. No hierarchical information is contained in these objects and a category can be specified at the lexical or the phrasal level. We will not present the category specifications in more detail here. However, we insist on the fact that categories are the only elements to be used by properties, at the exclusion of any other structured object such as local trees for example.

We propose to represent different kind of linguistic relations in PG by means of different constraints. We illustrate this aspect here with syntactic information, but the hypothesis is that other linguistic domains such as prosody, phonology, semantics or pragmatics can also be represented by means of such properties. As for syntax, the kind of information that we have to represent concerns different aspects such as linear order, selectional restrictions, sub-categorization, etc. What we propose is to represent this information by means of relations between categories. One consequence is that no relation between two objects needs to be propagated through a structure. In other words, in the manner of dependency grammars, relations are directly specified between the objects and not inherited in one way or another. However, this does not mean that PG represents syntax in a flat way. A hierarchized representation of syntactic structure is still relevant and PG specifies information over lexical items as well as phrases.

The considered syntactic relations are represented in terms of relations between sets of categories. This gives the formalism expressive power and makes it possible to represent – if necessary – contextual information. A category is simply specified by a set of such relations between those other categories which play a role in its structure. Lets see more precisely what these relation are before describing how a grammar is organized.

We use 6 different types of properties in order to represent syntactic information.[13] They are defined as follows:

Obligation (*oblig*): Set of compulsory, unique categories (they correspond to the heads of the phrase). This is a relation between a set of categories (the possible heads) and one category (the projection, usually the phrase).

Uniqueness (*uniq*): Set of categories which cannot be repeated in a phrase.

[Requirement ($\Rightarrow$):] Cooccurrence between sets of categories. This relation expresses generally selection between objects. It can be expressed between any categories, not necessarily an obligatory one. The possibility of using sets of categories allows the expression of contextual selections.

Exclusion ($\not\Leftrightarrow$): Restriction of cooccurrence between sets of categories.

[13] This set of properties could be extended if necessary in order to integrate other levels of linguistic analysis such as prosody or semantics.

Linearity ($\prec$): Linear precedence constraints.

Dependency ($\leadsto$): Dependency relations between categories. This relation concerns more precisely syntax/semantic interface, it is used to build the semantic representation of the category. We do not develop this point in this paper.

All these constraints, as it is usually the case, play a role as a filter but can also be used in order to instantiate new information. Let us illustrate these different properties with some examples taken from a grammar for French NPs:

- *Obligation*: *oblig(NP)* $=\{N, AP, Pro\}$
- *Uniqueness*: *uniq(NP)* $= \{Det, N, AP, PP, Sup, Pro\}$
- *Linearity*: $Det \prec AP$ $AP \prec N$
- *Requirement*: $\{le\ \hat{e}tre_{[N,P]}\} \overset{VP}{\Rightarrow} Clit_{[refl,N,P]}$ (Je me le suis dit / *I told that to myself*)
  If the categories *le* (clitic) and a finite verb *être* (with agreement features *N* and *P*) cooccur (characterizing a *VP*), then a reflexive clitic (which agrees with the verb) also has to be present.
- *Exclusion*: $Clit_{[refl]} \overset{VP}{\not\Leftrightarrow} lui$ (*Je me lui dis / **I told him myself*)
  In a *VP*, a reflexive clitic cannot cooccur with the clitic *lui*.
- *Dependency*: $Det \leadsto N$ $AP \leadsto N$

Please note that in spite of the fact that information is hierarchically represented, no constituency information is expressed by any constraint. In fact, one can observe that a property which would specify a set of constituents for a given category is redundant with the information contained by other properties. More precisely, the categories β_i that participate in the description of another category α are necessarily the constituents of α. Reciprocally, a constituent β_i of a category α stands necessarily in some relation with another constituent β_i of α or α itself (in the obligation relation). A category that does not have any relation with another participant of the description of α is simply not one of its constituent. Such a characteristic is useful for the representation of specific elements, in particular in spoken languages, that can appear almost everywhere during a discourse, even within a unit. It is then not necessary to stipulate explicitly a set of constituents.

3.2.3 Parsing as Constraint Resolution

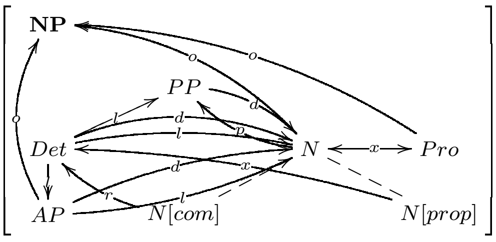
We propose in this section an overview of how does parsing works in PG. Before describing more precisely this process, let's highlight some points. First of all, insofar as syntactic information is only represented by means of constraints, a grammar then forms a constraint system. One important consequence is that all constraints are at the same level and therefore can be verified independently from each others. This is an important difference to generative approaches in which the constituency information has first to be evaluated in order to build local trees before verifying other constraints. Moreover, in contrast to Optimality Theory, no ranking between constraints has to be applied. Obviously, one can propose some heuristics at the computational level in order to modulate the importance of each constraint in the perspective of their relaxation for example. But this remains at the heuristic level.

In PG, parsing can then be implemented using constraint programming techniques such as constraint resolution. More precisely, and classically, parsing with PG consists in finding an assignment (a set of categories) that satisfies the set of constraints. Fortunately, this problem is heavily constrained by the fact that instead of finding an assignment over the entire set of categories, the problem consists in evaluating a given assignment over the constraint set: an input is a set of words that can be associated with a set of categories from which the different assignments will be built.

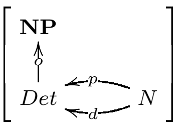
The interpretation of a constraint system in terms of constraint graphs is classical in constraint programming. We will see here how can we take advantage of that for specifying and representing linguistic information. In PG, each property is a relation between different (sets of) objects. A set of properties can then be represented as a graph in which categories constitute the nodes and constraints corresponds to the edges.

The graph given below represents the set of constraints describing the *NP*. In this representation, the types of the constraints are indicated as graph labels (*d* for dependency, *x* for exclusion, *l* for linearity, *o* for obligation, *r* for requirement).

This representation makes the role of the obligation relation which associates a category with its head precise. It is the only hierarchical relation. The target category of this relation is considered as the root of the graph and corresponds to the phrasal category described by the graph. We indicate this category in bold for better readability.



The same representation can be used for the description of a given input. Such a description consists in the set of relevant constraints (i.e., constraints that can be evaluated for a set of given categories). The description of a category in a context is then formed by the set of constraints that can be evaluated. Such graphs are called *description graphs.* The example immediately below presents the description graph for *NP* = {*Det, N*}.



One of the advantages of using graphs for the representation of syntactic information lies in the fact that any information can be represented separately. Under

Table 6. *Description of the four NPs in:* dans la marine tu as droit short blanc chemisette blanche

Category	Input	Constituents	Properties
NP_{21}	la marine	Det_2 N_3	$\mathcal{P}^+$=\{1, 5, 7, 9, 12\} $\mathcal{P}^- = \emptyset$
NP_{22}	short blanc	N_7 Adj_8	$\mathcal{P}^+$=\{3, 7, 10, 12\} $\mathcal{P}^-$=\{5\}
NP_{23}	chemisette blanche	N_9 Adj_{10}	$\mathcal{P}^+$=\{3, 7, 10, 12\} $\mathcal{P}^-$=\{5\}
NP_{24}	short blanc chemisette blanche	NP_{22} NP_{23}	$\mathcal{P}^+$=\{12\} $\mathcal{P}^-$=\{6\}

this perspective, one can choose to represent only some of the properties. This can be useful for example in shallow parsing.

Lets take an example from a spoken French corpus. For simplicity, we only focus here on the case of the *NP* which can be (roughly) described by the following subset of properties:

- *Linearity:*	*(1) Det* $\prec$ *N; (2) Det* $\prec$ *AP; (3) N* $\prec$ *AP; (4) N* $\prec$ *PP*
- *Requirement:*	*(5) N[com]* $\Rightarrow$ *Det; (6) NP* $\Rightarrow$ *Conj*
- *Exclusion:*	*(7) N* $\not\Leftrightarrow$ *Pro; (8) N[prop]* $\not\Leftrightarrow$ *Det;*
- *Dependency:*	*(9) Det* $\rightsquigarrow$ *N; (10) AP* $\rightsquigarrow$ *N; (11) PP* $\rightsquigarrow$ *N*
- *Obligation:*	*(12) Oblig(NP) = \{N, Pro, AP, Conj\}*

From this set of properties, we can give the characterizations of the different *NP*s. A characterization is formed by the sets of satisfied and violated constraints, respectively represented by $\mathcal{P}^+$ and $\mathcal{P}^-$. The constraints in the following examples are indicated by their indexes.

dans la marine tu as droit short blanc chemisette blanche

in the Navy you get white short white shirt

Four *NP*s participate in the description of this input, one of them being of higher level, see Tab. 3.2.3.

The first *NP* is positively characterized, it satisfies all its relevant properties. NP_{22} and NP_{23} partly satisfy the set of constraints. In both cases, a requirement property (stipulating that a determiner has to be realized together with the noun) is violated. NP_{24}, on the other hand, does not satisfy a requirement property concerning the realization of the conjunction. However the corresponding categories can be used as constituents for other categories as NP_{24}. More generally, as soon as a category can be characterized, it also can be part of the assignment (i.e., used as a constituent). These last cases illustrate the possibility of describing any kind of input, even those that can be considered as ill-formed with respect to the grammar. But this example also illustrates the fact that grammaticality is a particular characterization: in PG, ruling out ungrammatical inputs simply consists in restricting the building of $\mathcal{P}^-$ to the empty set. It is also a possibility, in terms of heuristics,

Table 7. *Some more characterizations of* dans la marine tu as droit short blanc chemisette blanche

Category	Input	Constituents	Properties
PP_{25}	dans la marine	$Prep_1$ NP_{21}	$\mathcal{P}^+ \neq \emptyset$ $\mathcal{P}^- = \emptyset$
NP_{26}	tu	Pro_4	$\mathcal{P}^+=\{7, 12\}$ $\mathcal{P}^- = \emptyset$
VP_{27}	as droit short blanc chemisette blanche	V_5 Adv_6 NP_{24}	$\mathcal{P}^+ \neq \emptyset$ $\mathcal{P}^- = \emptyset$

to restrict the cardinality of this set as well as using a constraint hierarchy favoring some constraints.

3.2.4 Determining Coverage

The core of the process is the construction of the characterization sets. A general mechanism makes use of these sets in order to find a characterization covering the entire input.

A characterization made up of lexical categories (cf. the NP's characterizations in the previous example), constitutes a covering of the segment of the input corresponding to the positions of the constituents. The question is different for embedded constituents. In this case, the set of categories covers the corresponding part of the input if there exists a characterization for all embedded constituents describing non-lexical categories. In the previous example, the characterization NP_{22}, NP_{23} covers the NP_{24} because there exists characterizations (resp. N_7, Adj_8 N_9, Adj_{10}) for its embedded constituents.

Let us be more precise on the evaluation of covering categorizations. Characterizations do not take into account the status of their embedded constituents. In other words, lexical and non-lexical categories play the same role. This means that each position corresponds to a category. In the case of embedded constituents, we then have several successive categories with the same type.

Let's complete the example of the previous section with other characterizations. In the following Tab. 3.2.4, insofar as we did not propose some constraints for *PP* and *VP*, we only indicate the situation with respect to the emptiness of the $\mathcal{P}$ sets.

One covering of the input will be the set of categories: PP_{25}, NP_{26}, VP_{27} which characterizes a sentence.

A naive implementation of a mechanism consisting in enumerating all the possible sets of categories is obviously not efficient. Indeed, let m be the number of words of the input, c the maximum number of categorizations for each word, then the number n of categories to analyze is bounded by $n = 2mc$. Then, the number of sets of categories to analyze is bounded by 2^n.

Several mechanisms can be added in order to reduce the number of sets to analyze and control the resolution process. In the first case, we can choose to build only sets of juxtaposed categories. At a higher level, one can consider only sets corresponding to subsets of properties. The satisfaction process itself can be controlled using several heuristics. In particular, it is possible to filter the satisfiability according to a threshold given by the cardinality of the set of unsatisfied properties $\mathcal{P}^-$: it is possible to build only characterizations with less than a certain amount of unsatisfied constraints. At the extreme, we can reduce the satisfiability to positive characterizations.

Another kind of heuristic consists in defining a hierarchization within the set of properties. In this case, the technique of the threshold mentioned above can be modulated with such a hierarchization, some constraints playing a more important role than others.

3.2.5 Implementing Property Grammars

We describe in this section an implementation of Property Grammars in CHR. We want to show with such an implementation that property grammar can be implemented directly using only constraint resolution, which is not actually possible with classical generative approaches.

For the implementation in CHR, we simply have to declare the basic properties as constraints, see Sec. 3.2 above. They are presented in Tab. 8. Our items are simply statements about the categories and their position in the input string. Since we do not have a chart parser, but rather are looking for characterizations of the input, we do not have axioms and goals as before. Instead, as one can see, we use additional constraints for the sets of satisfied/nonsatisfied constraints, i.e., we have constraints `pplus/4` and `pminus/4` which encode in their first argument the set of constraints we are considering (the module) and an identifier, in their second argument the type of constraint and in the last two the specific categories involved. We could have recorded the satisfied and violated constraints in other ways but choose this one simply to stay within CHR and use constraint propagation to characterize the input. We also associate a unique identifier with each constraint such that we are able to

Table 8. *A CHR-based property grammar parser: Items and Constraints*

Items	`cat(Cat,N)`
Constraints	`obli(Mod,Cat,Id)`
	`impl(Mod,Cat1,Cat2,Id)`
	`prec(Mod,Cat1,Cat2,Id)`
	`cooc(Mod,Cat1,Cat2,Id)`
	`depe(Mod,Cat1,Cat2,Id)`
	`alph(Mod,Cats)`
	`pplus(Mod-Id,Type,Cat1-N,Cat2-M)`
	`pminus(Mod-Id,Type,Cat1-N,Cat2-M)`

refer to particular constraints in the solution. In particular, the `alph/2` constraint encodes the possible constituents of a module and is therefore used in building the projection lines, see Tab. 9. The other constraints are self-explanatory; `obli/3` means that the category is obligatory in a module; `impl/4` means that within a module one category implies another and similarly for precedence (`prec/4`), co-occurrence (`cooc/4`) ad dependency (`depe/4`).

The program takes the input string, a Prolog list of words, and traverses it.[14] For each lexical entry, it posts the corresponding category constraint, i.e., `cat(X,Y)` where `X` is a category and `Y` its position in the numeration.

The CHR implementation immediately starts out by building the elementary trees of the initial categories from the posted `cat/2` constraints via constraint propagation. In general, the CHR engine, i.e., constraint resolution, is triggered as soon as constraints matching the heads are posted.

Basically, the CHR program looks at two category constraints and – if they are in the set of constraints we are considering – uses them together with a constraint from the grammar, as the head of constraint propagation rules. The constraints propagated contain the information whether the pair of categories satisfy the constraint from the grammar or not. In this way, we build a complete table of results on all pairs of categories.[15]

So, we build a complete table of `pplus/4, pminus/4` constraints for all pairs of categories matching the heads of our propagation rules. These constraints store the information about the interaction of any two categories with the resulting rule and set of constraints which licensed them. Nothing more has to be implemented to achieve this behaviour since it is built into the CHR engine. All we have to do is specify the right propagation rules and to post the constraints from the grammar and input which makes for a very simple and easy to understand program.

So, again, the utilities for parsing property grammars are extremely simple, see Fig. 5. We simply post the constraints for the words and the grammar and report on the output.

Generally, in all the CHR rules dealing with the constraints from the grammar, we apply the rule if we can find two categories within the constraint store which are in the same constituent and a constraint which checks some configuration on them. Only this last constraint triggers the rule since we declared all other constraints to be passive. Note that this ensures that we did all possible projections in each constituent before we post other constraints.

Since, if two category constraints match the head of a rule, they do so in either order, we use the guard to force the linear order as it appears in the input string.

The propagation rules for parsing property grammars are given in Tab. 9. Lets pick the rule dealing with precedence constraints for a closer look. We are considering a precedence constraint `Id` on the categories `X` and `Y` within the constituent

[14] Again from left-to-right although this is arbitrary and can be varied.

[15] Since in the end we are interested in bigger chunks of lexical items then just two, we have to do some post-processing to interpret the entries in the table in the right way. For example, if a precedence constraint succeeds on two pairs of categories, this does not imply that it also succeeds for the entire triple. So, one violation among all the pairs is sufficient to cause failure of that constraint.

Table 9. *A CHR-based property grammar parser: Inference Rules*

Building Projection Lines

```
project @ cat(X,N), alph(Cat,Alph,_) ==>
        member(X,Alph) |
        cat(Cat,N).
```

Obligatory Constraints

```
obli @ cat(X,N)#I1, cat(Z,N)#I2, cat(Y,M)#I3, cat(Z,M)#I4,
                                              obli(Z,Cat,Id) ==>
        N < M |
        ( ( X == Cat ; Y == Cat ) ->
            pplus(Z-Id,obli,X-N,Y-M)
        ;   pminus(Z-Id,obli,X-N,Y-M) )
        pragma passive(I1),passive(I2),passive(I3),passive(I4).
```

Implicational Constraints

```
impl @ cat(X,N)#I1, cat(Z,N)#I2, cat(Y,M)#I3, cat(Z,M)#I4,
                                              impl(Z,Cat1,Cat2,Id) ==>
        N < M |
        ( ( (X == Cat1, Y \== Cat2);(X \== Cat2, Y == Cat1) )  ->
            pminus(Z-Id,impl,X-N,Y-M)
        ;   pplus(Z-Id,impl,X-N,Y-M) )
        pragma passive(I1),passive(I2),passive(Id),passive(I4).
```

Precedence Constraints

```
prec @ cat(X,N)#I1, cat(Z,N)#I2, cat(Y,M)#I3, cat(Z,M)#I4,
                                              prec(Z,Cat1,Cat2,Id) ==>
       N < M |
       ( ( X == Cat2, Y == Cat1 ) ->
            pminus(Z-Id,prec,X-N,Y-M)
       ;    pplus(Z-Id,prec,X-N,Y-M) )
       pragma passive(I1), passive(I2),  passive(I3), passive(I4).
```

Coocurrence

```
cooc @ cat(X,N)#I1, cat(Z,N)#I2, cat(Y,M)#I3, cat(Z,M)#I4,
                                              cooc(Z,Cat1,Cat2,Id) ==>
        N < M |
        ( ( ( X == Cat1, Y == Cat2 );( X == Cat2, Y == Cat1 ) )   ->
            pminus(Z-Id,cooc,X-N,Y-M)
        ;   pplus(Z-Id,cooc,X-N,Y-M) )
        pragma passive(I1),passive(I2),passive(I3),passive(I4).
```

Absorption

```
absorb1 @ cat(X,N) \ cat(X,N) <=> true.
absorb2 @ pplus(X,Y,A,B) \ pplus(X,Y,A,B) <=> true.
absorb3 @ pminus(X,Y,A,B) \ pminus(X,Y,A,B) <=> true.
```

```
process(In) :-
        numeration(In,1),
        post_grammar,
        generate_output.

numeration([],_).
numeration([H|T],N):-
        lex(H,Cat),
        cat(Cat,N),
        N1 is N + 1,
        numeration(T,N1).
```

Fig. 5. The utilities for parsing property grammars

`Z` with indices `N` and `M`. We check satisfiability of the constraint in the head of the implication and then add the corresponding `pplus` or `pminus` constraint to the store.[16] Note that all the constraints actually appearing in the constraint store are ground such that we can test for literal identity.

The post-processing then just collects all the sets, gets all pairs of appearing categories if they are within the same range and collects all positive and negative constraints attached to them. While doing so, we interpret its members in the following way:

- if we have a successful obligatory or implicational constraint on any two categories within a constituent, we can ignore all the failed ones;
- if we have a failed precedence or co-occurrence constraint on any two categories within a constituent, we have to ignore all the successful ones.

Furthermore, the output of the program – consisting of the `pplus/pminus`-sets generated by the resolution process – can be varied according to the given threshold of failed constraints, in the extreme case reducing the process to finding only the positive characterizations, see Fig. 6 for the output of processing *le livre*. As can be seen in this admittedly very small example, the program generates two characterizations, one without violations for the fact that *le livre* indeed is an NP with determiner *le* and noun *livre* and the other one under the assumption that *le* is part of a superlative which leads to two violated constraints. The constraint store which follows in the printout contains the constraints in the order they are posted. As one can see, we start out with the category constraints which are followed by some constraints of the grammar. As soon as we have enough information to infer further constraints, those are added to the store and the process continues until no new constraints can be deduced. The final constraint store contains both the `pminus` and `pplus` constraints for all possible characterizations.

[16] We need four `cat` constraints to ensure that both categories belong to the same constituent, i.e., that `Z` indeed covers `N` and `M`.

```
| ?- process([le,livre]).

++++++++++++++++++++++++++++++++++++++
Categories: det-1 and n-2
P+( np ) = [ 2, 3, 4, 5, 6, 7, 8, 9 ]
P-( np ) = [  ]

++++++++++++++++++++++++++++++++++++++

Categories: sup-1 and n-2
P+( np ) = [ 2, 4, 5, 6, 7, 9 ]
P-( np ) = [ 3, 8 ]

++++++++++++++++++++++++++++++++++++++


cat(det,1),
cat(n,2),
cat(np,2),
cat(np,1),
cat(sup,1),
alph(np,[det,n,adjp,sup],1),
alph(adjp,[adj,adv],1),
alph(sup,[det,adv,adj],1),
obli(np,n,2),
obli(adjp,adj,2),
obli(sup,adj,2),
pplus(np-2,obli,sup-1,n-2),
pplus(np-2,obli,det-1,n-2),
impl(np,n,det,3),
pminus(np-3,impl,sup-1,n-2),
pplus(np-3,impl,det-1,n-2),
impl(sup,adj,det,3),
impl(sup,adj,adv,4),
prec(np,det,n,4),
pplus(np-4,prec,sup-1,n-2),
pplus(np-4,prec,det-1,n-2),
prec(np,det,adjp,5),
pplus(np-5,prec,sup-1,n-2),
pplus(np-5,prec,det-1,n-2),
prec(np,det,sup,6),
pplus(np-6,prec,sup-1,n-2),
pplus(np-6,prec,det-1,n-2),
prec(np,n,adjp,7),
pplus(np-7,prec,sup-1,n-2),
pplus(np-7,prec,det-1,n-2),
prec(np,n,sup,8),
pminus(np-8,prec,sup-1,n-2),
pplus(np-8,prec,det-1,n-2),
prec(adjp,adv,adj,3),
prec(sup,det,adj,5),
prec(sup,det,adv,6),
prec(sup,adv,adj,7),
cooc(np,adjp,sup,9),
pplus(np-9,cooc,sup-1,n-2),
pplus(np-9,cooc,det-1,n-2),
depe(np,det,n,10),
depe(np,adjp,n,11),
depe(np,sup,n,12),
depe(adjp,adv,adj,4),
depe(sup,det,adj,8),
depe(sup,adv,adj,9) ? ;
```

Fig. 6. Output and constraint store for *le livre*

The full, commented code of the implementation is available in a technical report (Blache & Morawietz, 2000).

3.3 Compiling the Grammar Rules into the Inference Rules

A proposal for improving the efficiency of the proposed approach consists in moving the test for rule applicability from the guards into the heads of the CHR rules. One can translate a given context-free grammar under a given set of inference rules into a CHR program which contains constraint propagation rules for each grammar rule, thereby making the process more efficient.[17] For simplicity, we discuss only the simplest case of bottom-up parsing.

For the translation from a CF grammar into a constraint framework we have to

[17] This is essentially the approach proposed in Meyer (2000). In this paper, one can also find a proof that such a translation is correct.

distinguish two types of rules – those with from those without an empty RHS – since empty RHSs constitute a problem for bottom-up chart parsing.[18] We treat the easier case of the conversion first. For each rule in the CF grammar with a non-empty RHS we create a constraint propagation rule such that each daughter of the rule introduces an edge constraint in the head of the propagation rule with variable, but appropriately matching string positions but a fixed label. The new, propagated edge constraint spans the entire range of the positions of the daughters and is labeled with the (nonterminal) symbol of the LHS of the CF rule. In our example, the resulting propagation rule for `s` looks as follows:

```
edge(I,K,np), edge(K,J,vp) ==> edge(I,J,s)
```

As usual for bottom-up parsing, the translation is a little bit more complicated for rules with empty RHSs. Basically, we create a propagation rule for each empty rule, e.g., $\texttt{A} \longrightarrow \epsilon$, such that the head is an arbitrary edge, i.e., both positions and the label are arbitrary variables, and post new edge constraints with the LHS of the CF rule as label, using the positional variables and spanning no portion of the string, resulting in CHR rules of the following type:

```
edge(I,J,_Sym) ==> edge(I,I,A), edge(J,J,A)
```

But obviously rules of this type lead to nontermination since they would propagate further constraints on their own output which is avoided by including a guard which ensures that empty edges are only propagated for every possible string position once by testing whether the edge spans a string of length one. Recall that storing and using already existing edge constraints is avoided with an absorption rule.[19] Since these empty constraints can be reused an arbitrary number of times, we get the desired effect without having to fear nontermination. Although this is not an elegant solution, it seems that other alternatives such as analyzing and transforming the entire grammar or posting the empty constraints while traversing the input string are not appealing either since they give up the one-to-one correspondence between the rules of the CF grammar and the constraint program which is advantageous in debugging.

With this technique, the parsing times achieved were better by a factor of a third compared to the Shieber *et al.* implementation. Although now the process of the compilation obscures the direct connection between parsing-as-deduction and constraint propagation somewhat, the increase in speed makes it a worthwhile exercise. There are some further ways to improve the performance of the resulting CHR parsers involving for example pragma declarations or more sophisticated ways of handling empty categories via a precompiled link relation which we cannot go into here for reasons of space.

[18] Top-down parsing, on the other hand, has to be careful with left recursion and care has to be taken in the parser to avoid nontermination.

[19] The proper and more elegant treatment of this problem would test in the guard of the CHR rule whether the edge constraint to be posted is already entailed by the constraint store. Unfortunately, having a constraint in the guard of a clause in CHR causes that constraint to be posted which is definitely not what is needed.

4 Conclusion

In the paper, the similarity between parsing-as-deduction and constraint propagation is used to propose a flexible and simple system which is easy to implement and therefore offers itself as a testbed for different parsing strategies (such as top-down or bottom-up), for varying modes of processing (such as left-to-right or right-to-left) or for different types of grammars (such as for example minimalist or property grammars). Compared to the Shieber approach, the pure version seems to be lacking in efficiency. This can be remedied by providing an automatic compilation into more efficient specialized parsers.

More work has to be invested in the realization of a larger system combining these techniques with constraint solvers for existing constraint-based natural language theories to see whether further benefits can be gotten from using parsing as constraint programming. To be more specific, due to the flexibility of the CHR system, one can now use the constraint programming approach to drive other constraint solving or constraint resolution techniques (also implemented in CHR) resulting in a homogenous environment which combines both classical constraint solving with a more operational generator.

Specifically, one can use each created edge to post other constraints, for example about the well-formedness of associated typed feature structures. By posting them, they become available for other constraint handling rules. In particular, systems implementing HPSG seem to suffer from the problem how to drive the constraint resolution process efficiently. Some systems, as for example ALE (Carpenter & Penn, 1998) use a phrase structure backbone to drive the process. The proposal here would allow to use the ID/LP schemata directly as constraints, but nevertheless as the driving force behind the other constraint satisfaction techniques. However, this remains speculative.

References

Abdennadher, Slim. (1998). *Analyse von regelbasierten Constraintlösern.* Ph.D. thesis, Ludwig-Maximilians-Universität München.

Abdennadher, Slim, Fruehwirth, Thom, & Meuss, Holger. (1996). On confluence of constraint handling rules. *Lecture Notes in Computer Science*, **1118**, 1–15.

Balfourier, Jean-Marie, Blache, Philippe, & van Rullen, Tristan. (2002). From shallow to deep parsing using constraint satisfaction. *Proceedings of COLING 2002.*

Blache, Philippe. (2000). Property grammars and the problem of constraint satisfaction. *Linguistic theory and grammar implementation.* ESSLLI 2000 workshop.

Blache, Philippe. (2001). *Les Grammaires de Propriétés : Des contraintes pour le traitement automatique des langues naturelles.* Herms.

Blache, Philippe, & Morawietz, Frank. (2000). A non-generative constraint-based formalism. *Pages 1–28 of:* Morawietz, Frank (ed), *Some aspects of natural language processing and constraint programming.* Arbeitspapiere des SFB 340, no. 150. Universität Tübingen.

Blache, Philippe, & Paquelin, Jean-Louis. (1996). Active constraints for a direct interpretation of HPSG. *Proceedings of HPSG'96.*

Carpenter, Bob, & Penn, Gerald. 1998 (March). *ALE: The attribute logic engine, version*

3.1. User manual. Laboratory for Computational Linguistics, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA 15213.

Chomsky, Noam. (1995). *The minimalist program*. Current Studies in Linguistics, vol. 28. MIT Press.

Christiansen, Henning. (2001). *CHR as grammar formalism, a first report*. `http://arxiv.org/abs/cs.PL/0106059`. Presented at ERCIM Workshop on Constraints, Prague.

Christiansen, Henning. (2002a). Abductive language interpretation as bottom-up deduction. Wintner, Shuly (ed), *Proceedings of NLULP 2002, Natural Language Understanding and Logic Programming*. To appear.

Christiansen, Henning. (2002b). Logical grammars based on constraint handling rules. Stuckey, S. (ed), *Proceedings of the Eighteenth International Conference on Logic Progamming, ICLP 2002*. Springer. Poster Abstract. To appear.

Frühwirth, Thom. (1998). Theory and practice of constraint handling rules. *Journal of logic programming*, **37**(1–3), 95–138. Special Issue on Constraint Logic Programming.

Gazdar, Gerald, Klein, Ewan, Pullum, Geoffrey K., & Sag, Ivan A. (1985). *Generalized Phrase Structure Grammar*. Cambridge, Massachusetts: Harvard University Press.

Götz, Thilo, & Meurers, Detmar. (1997). Interleaving universal principles and relational constraints over typed feature logic. *Pages 1–8 of: Proceedings of the ACL/EACL conference '97*. Association for Computational Linguistics, Madrid, Spain.

Graham, G., Harrison, M. G., & Ruzzo, W. L. (1980). An improved context–free recognizer. *Pages 415–462 of: ACM Transactions on Programming Languages and Systems 2 (3)*. ACM.

Intelligent Systems Laboratory. (1995). *SICStus Prolog user's manual*. Tech. rept. Swedish Institute of Computer Science.

Manandhar, Suresh. (1994). An attributive logic of set descriptions and set operations. *Proceedings of the 32nd. Annual Meeting of the Association for Computational Linguistics*. ACL.

Marriott, Kim, & Stuckey, Peter J. (1998). *Programming with constraints*. MIT Press.

Matiasek, Johannes. (1994). *Principle-based processing of natural language using CLP techniques*. Ph.D. thesis, TU Wien.

Meurers, Detmar, & Minnen, Guido. (1998). Off-line constraint propagation for efficient HPSG processing. Webelhuth, G., Koenig, J.-P., & Kathol, A. (eds), *Lexical and constructional aspects of linguistic explanation*. CSLI.

Meyer, Bernd. (2000). A constraint-based framework for diagrammatic reasoning. *Journal of Applied Artificial Intelligence*, **14**(4), 327–344.

Morawietz, Frank. (1995). A Unification-Based ID/LP Parsing Schema. *Pages 162–173 of: Proceedings of the International Workshop on Parsing Technologies*. ACL/SIGPARSE, Prag.

Morawietz, Frank. (1999). *Bottom-up chart parsing as constraint propagation*. `http://tcl.sfs.uni-tuebingen.de/~frank/`.

Morawietz, Frank. (2000a). Chart parsing and constraint programming. *Proceedings of COLING-2000*.

Morawietz, Frank. (2000b). Chart parsing as constraint propagation. *Proceedings of the International Workshop on Parsing Technologies IWPT 2000*. ACL/SIGPARSE, Trento.

Morawietz, Frank. (2000c). Chart parsing as constraint propagation. *Pages 29–50 of:* Morawietz, Frank (ed), *Some aspects of natural language processing and constraint programming*. Arbeitspapiere des SFB 340, no. 150. Universität Tübingen.

Penn, Gerald. (1999). An optimized prolog encoding of typed feature structures. *Pages 124–138 of: Proceedings of the 16th International Conference on Logic Programming.*

Penn, Gerald. (2000). Applying constraint handling rules to HPSG. *Proceedings of the First International Conference on Computational Logic (CL2000), Workshop on Rule-Based Constraint Reasoning and Programming.*

Pereira, Fernando C. N., & Warren, David H. D. (1983). Parsing as deduction. *ACL proceedings, 21st Annual Meeting*, vol. 13.

Prince, Alan, & Smolensky, Paul. (1993). Optimality Theory: Constraint interaction in generative grammars. Technical Report RUCCS TR-2. Rutgers Center for Cognitive Science.

Sag, Ivan, & Wasow, T. (1999). *Syntactic Theory. A Formal Introduction.* CSLI.

Saraswat, Vijay A. (1993). *Concurrent constraint programming.* Cambridge, Massachusetts: MIT Press.

Shieber, Stuart M., Schabes, Yves, & Pereira, Fernando C. N. (1995). Principles and implementation of deductive parsing. *Journal of Logic Programming*, **24**(1–2), 3–36.

Sikkel, Klaas. (1997). *Parsing schemata: A framework for specification and analysis of parsing algorithms.* ETACS Series: Texts in Theoretical Computer Science. Springer.

Smolka, Gert. (1995). The Oz programming model. *Pages 324–343 of:* van Leeuwen, Jan (ed), *Computer Science Today.* Lecture Notes in Computer Science, vol. 1000. Berlin: Springer-Verlag.

Stabler, Edward P. (1997). Derivational minimalism. *Pages 68–95 of:* Retoré, Christian (ed), *Logical aspects of computational linguistics.* Berlin: Springer. LNAI 1328.

Stabler, Edward P. (2001). Minimalist grammars and recognition. *Pages 327–352 of:* Rohrer, Christian, Rossdeutscher, Antje, & Kamp, Hans (eds), *Linguistic form and its computation.* CSLI Publications.