An Overview of
Disjunctive Constraint Satisfaction

John T. Maxwell III and Ronald M. Kaplan

Xerox Palo Alto Research Center

Introduction

This paper presents a new algorithm for solving disjunctive systems of constraints. The algorithm determines whether a system is satisfiable and produces the models if the system is satisfiable. There are three main steps for determining whether or not the system is satisfiable:

1) turn the disjunctive system into an equi-satisfiable conjunctive system in polynomial time
2) convert the conjunctive system into canonical form using extensions of standard techniques
3) extract and solve a propositional 'disjunctive residue'

Intuitively, the disjunctive residue represents the unsatisfiable combinations of disjuncts in a propositional form based on the content of the constraints. Each of the transformations above preserves satisfiability, and so the original disjunctive system is satisfiable if and only if the disjunctive residue is satisfiable. If the disjunctions are relatively independent (as frequently happens in grammatical specifications), then the disjunctive residue is significantly easier to solve than the original system.

The first three sections of this paper cover the steps outlined above. The fourth section describes how models can be produced. Finally, the last section compares this approach with some other techniques for dealing with disjunctive systems of constraints.

Turning Disjunctions into Conjunctions

Basic Lemma

Our method depends on a simple lemma for converting a disjunction into a conjunction of implications:

(1) $\phi_1 \vee \phi_2$ is satisfiable iff $(p \rightarrow \phi_1) \wedge (\neg p \rightarrow \phi_2)$ is satisfiable,
where p is a new propositional variable.

Proof:

1) If $\phi_1 \vee \phi_2$ is satisfiable, then either ϕ_1 is satisfiable or ϕ_2 is satisfiable. Suppose that ϕ_1 is satisfiable. Then if we choose p to be true, then $p \rightarrow \phi_1$ is satisfiable because ϕ_1 is satisfiable, and $\neg p \rightarrow \phi_2$ is vacuously satisfiable because its antecedent is false. Therefore $(p \rightarrow \phi_1) \wedge (\neg p \rightarrow \phi_2)$ is satisfiable.

2) If $(p \rightarrow \phi_1) \wedge (\neg p \rightarrow \phi_2)$ is satisfiable, then both clauses are satisfiable. One clause will be vacuously satisfiable because its antecedent is false and the other will have a true antecedent. Suppose that $p \rightarrow \phi_1$ is the clause with the true antecedent. Then ϕ_1 must be satisfiable for $p \rightarrow \phi_1$ to be satisfiable. But if ϕ_1 is satisfiable, then so is $\phi_1 \vee \phi_2$. Q.E.D.

Intuitively, the new variable p is used to encode the requirement that at least one of the disjuncts be true. In the remainder of the paper we use lower-case p to refer to a single propositional variable, and upper-case P to refer to a boolean combination of propositional variables. We call $P \rightarrow \phi$ a *contexted* constraint, where P is the *context* and ϕ is called the *base* constraint.

(Note that this lemma is stated in terms of *satisfiability*, not logical equivalence. A form of the lemma that emphasized logical equivalence would be: $\phi_1 \vee \phi_2 \leftrightarrow \exists p: (p \rightarrow \phi_1) \wedge (\neg p \rightarrow \phi_2)$.)

Turning a Disjunctive System into a Conjunctive System

The lemma given above can be used to convert a disjunctive system of constraints into an flat conjunction of contexted constraints in polynomial time. The resulting conjunction is satisfiable if and only if the original system is satisfiable. The algorithm for doing so is as follows:

(2) a) push all of the negations down to the literals
b) turn all of the disjunctions into conjunctions using the lemma above
c) flatten nested contexts with: $(P_i \rightarrow (P_j \rightarrow \phi)) \Leftrightarrow (P_i \wedge P_j \rightarrow \phi)$
d) separate conjoined constraints with: $(P_i \rightarrow \phi_1 \wedge \phi_2)) \Leftrightarrow (P_i \rightarrow \phi_1) \wedge (P_i \rightarrow \phi_2)$

This algorithm is a variant of the reduction used to convert disjunctive systems to CNF in the proof that CNF is NP-complete[4], and is thus known to run in polynomial time. In effect, we are simply converting the disjunctive system to an implicational form of CNF (note that $P \rightarrow \phi$ is logically equivalent to $\neg P \vee \phi$). CNF has the desirable property that if any one clause can be shown to be unsatisfiable, then the entire system is unsatisfiable.

Example

The functional structure f of an uninflected verb in English has the following constraints in the formalism of Lexical-Functional Grammar[6]:

(3) ((f INF) = - $\wedge$ (f TENSE) = PRES $\wedge$ $\neg$[(f SUBJ NUM) = SG $\wedge$ (f SUBJ PERS) = 3]) $\vee$ (f INF) = +

(In LFG notation, a constraint of the form (f a) = v asserts that f(a) = v, where f is a function, a is an attribute, and v is a value. (f a b) = v is shorthand for f(a)(b) = v.) These constraints say that an uninflected verb in English is either a present tense verb which is not third person singular or it is infinitival. In the left column below this system has been reformatted so that it can be compared with the results of applying algorithm (2) to it, shown on the right:

reformatted:		*converts to*:	
(	(f INF) = -	($p_1 \rightarrow$	(f INF) = -) $\wedge$
$\wedge$	(f TENSE) = PRES	($p_1 \rightarrow$	(f TENSE) = PRES) $\wedge$
$\wedge \neg$ [	(f SUBJ NUM) = SG	($p_1 \wedge p_2 \rightarrow$	(f SUBJ NUM) $\neq$ SG) $\wedge$
$\wedge$	(f SUBJ PERS) = 3])	($p_1 \wedge \neg p_2 \rightarrow$	(f SUBJ PERS) $\neq$ 3) $\wedge$
$\vee$	(f INF) = +	($\neg p_1 \rightarrow$	(f INF) = +)

Converting the Constraints to Canonical Form

A conjunction of contexted constraints can be put into an equi-satisfiable canonical form that makes it easy to identify all unsatisfiable combinations of constraints. The basic idea is to start with algorithms that determine the satisfiability of purely conjunctive systems and extend each rule of inference or rewriting rule so that it can handle contexted constraints. We illustrate this approach by modifying two conventional satisfiability algorithms, one based on deductive expansion and one based on rewriting.

Deductive Expansion

Deductive expansion algorithms work by determining all the deductions that could lead to unsatisfiability given an initial set of clauses and some rules of inference. The key to extending a deductive expansion algorithm to contexted constraints is to show that for every rule of inference that is applicable to the base constraints, there is a corresponding rule of inference that works for contexted

constraints. The basic observation is that base constraints can be conjoined if their contexts are conjoined:

(4) $(P_1 \rightarrow \phi_1) \wedge (P_2 \rightarrow \phi_2) \Rightarrow (P_1 \wedge P_2 \rightarrow \phi_1 \wedge \phi_2)$

If we know from the underlying theory of conjoined base constraints that $\phi_1 \wedge \phi_2 \rightarrow \phi_3$, then the transitivity of implication gives us:

(5) $(P_1 \rightarrow \phi_1) \wedge (P_2 \rightarrow \phi_2) \Rightarrow (P_1 \wedge P_2 \rightarrow \phi_3)$

Equation (5) is the contexted version of $\phi_1 \wedge \phi_2 \rightarrow \phi_3$. Thus the following extension of a standard deductive expansion algorithm works for contexted constraints:

(6) For every pair of contexted constraints $P_1 \rightarrow \phi_1$ and $P_2 \rightarrow \phi_2$ such that:
a) there is a rule of inference $\phi_1 \wedge \phi_2 \rightarrow \phi_3$
b) $P_1 \wedge P_2 \neq$ FALSE
c) there are no other clauses $P_3 \rightarrow \phi_3$ such that $P_1 \wedge P_2 \rightarrow P_3$
add $P_1 \wedge P_2 \rightarrow \phi_3$ to the conjunction of clauses being processed.

Condition (6b) is based on the observation that any constraint of the form FALSE $\rightarrow \phi$ can be discarded since no unsatisfiable constraints can ever be derived from it. This condition is not necessary for the correctness of the algorithm, but may have performance advantages. Condition (6c) corresponds to the condition in the standard deductive expansion algorithm that redundant constraints must be discarded if the algorithm is to terminate. We extend this condition by noting that any constraint of the form $P_i \rightarrow \phi$ is redundant if there is already a constraint of the form $P_j \rightarrow \phi$, where $P_i \rightarrow P_j$. This is because any unsatisfiable constraints derived from $P_i \rightarrow \phi$ will also be derived from $P_j \rightarrow \phi$. Our extended algorithm terminates if the standard algorithm for simple conjunctions terminates. When it terminates, an equi-satisfiable disjunctive residue can be easily extracted, as described below.

Rewriting

Rewriting algorithms work by repeatedly replacing conjunctions of constraints with logically equivalent conjunctions until a normal form is reached. This normal form usually has the property that all unsatisfiable constraints can be determined by inspection. Rewriting algorithms use a set of rewriting rules that specify what sorts of replacements are allowed. These are based on logical equivalences so that no information is lost when replacements occur. Rewriting rules are interpreted differently from logical equivalences, however, in that they have directionality: whenever a logical expression matches the left-hand side of a rewriting rule, it is replaced by an instance of the logical expression on the right-hand side, but not vice-versa. To distinguish the two, we will use $\leftrightarrow$ for logical equivalence and $\Leftrightarrow$ for rewriting rules. (This corresponds our use of $\rightarrow$ for implication and $\Rightarrow$ for deduction above.)

A rewriting algorithm for contexted constraints can be produced by showing that for every rewrite rule that is applicable to the base constraints, there is a corresponding rewrite rule for contexted constraints. Suppose that $\phi_1 \wedge \phi_2 \Leftrightarrow \phi_3$ is a rewriting rule for base constraints. An obvious candidate for the contexted version of this rewrite rule would be to treat the deduction in (5) as a rewrite rule:

(7) $(P_1 \rightarrow \phi_1) \wedge (P_2 \rightarrow \phi_2) \Leftrightarrow (P_1 \wedge P_2 \rightarrow \phi_3)$ (incorrect)

This is incorrect because it is not a logical equivalence: the information that ϕ_1 is true in the context $P_1 \wedge \neg P_2$ and that ϕ_2 is true in the context $P_2 \wedge \neg P_1$ has been lost as the basis of future deductions. If we add clauses to cover these cases, we get the logically correct:

(8) $(P_1 \rightarrow \phi_1) \wedge (P_2 \rightarrow \phi_2) \Leftrightarrow (P_1 \wedge \neg P_2 \rightarrow \phi_1) \wedge (P_2 \wedge \neg P_1 \rightarrow \phi_2) \wedge (P_1 \wedge P_2 \rightarrow \phi_3)$

This is the contexted equivalent of $\phi_1 \wedge \phi_2 \Leftrightarrow \phi_3$. Note that the effect of this is that the contexted constraints on the right-hand side have unconjoinable contexts (that is, their conjunction is tautologically false). Thus, although the right-hand side of the rewrite rule has more conjuncts than the left-hand side, there are fewer implications to be derived from them.

Loosely speaking, a rewriting algorithm is constructed by iterative application of the contexted versions of the rewriting rules of a conjunctive theory. Rather than give a general outline here, let us consider the particular case of attribute value logic.

Application to Attribute-Value Logic

Attribute-value logic is used by both LFG and unification-based grammars. We will start with a simple version of the rewriting formalism given in Johnson[5]. For our purposes, we only need two of the rewriting rules that Johnson defines[5 pp. 38-39]:

(9) $t_1 \approx t_2 \Leftrightarrow t_2 \approx t_1$ when $\|t_1\| < \|t_2\|$ ($\|t_i\|$ is Johnson's norm for terms.)

(10) $t_2 \approx t_1 \wedge \phi \Leftrightarrow t_2 \approx t_1 \wedge \phi[t_2/t_1]$ where ϕ contains t_2 and $\|t_2\| > \|t_1\|$

($\phi[t_2/t_1]$ denotes "ϕ with every occurrence of t_2 replaced by t_1".)

We turn equation (10) into a contexted rewriting rule by a simple application of (7) above:

(11) $(P_1 \rightarrow t_2 \approx t_1) \wedge (P_2 \rightarrow \phi)$
$\Leftrightarrow (P_1 \wedge \neg P_2 \rightarrow t_2 \approx t_1) \wedge (\neg P_1 \wedge P_2 \rightarrow \phi) \wedge (P_1 \wedge P_2 \rightarrow (t_2 \approx t_1 \wedge \phi[t_2/t_1]))$

We can collapse the two instances of $t_2 \approx t_1$ together by observing that $(P \rightarrow A \wedge B) \leftrightarrow (P \rightarrow A) \wedge (P \rightarrow B)$ and that $(P_i \rightarrow A) \wedge (P_j \rightarrow A) \leftrightarrow (P_i \vee P_j \rightarrow A)$, giving the simpler form:

(12) $(P_1 \rightarrow t_2 \approx t_1) \wedge (P_2 \rightarrow \phi) \Leftrightarrow (P_1 \rightarrow t_2 \approx t_1) \wedge (P_2 \wedge \neg P_1 \rightarrow \phi) \wedge (P_2 \wedge P_1 \rightarrow \phi[t_2/t_1])$

Formula (12) is the basis for a very simple rewriting algorithm for a conjunction of contexted attribute-value constraints.

(13) For each pair of clauses $P_1 \rightarrow t_j \approx t_i$ and $P_2 \rightarrow \phi$:
a) if $\|t_j\| > \|t_i\|$, then set t_2 to t_j and t_1 to t_i, else set t_2 to t_i and t_1 to t_j
b) if ϕ mentions t_2 then replace $P_2 \rightarrow \phi$ with $(P_2 \wedge \neg P_1 \rightarrow \phi) \wedge (P_2 \wedge P_1 \rightarrow \phi[t_2/t_1])$

Notice that since $P_1 \rightarrow t_2 \approx t_1$ is carried over unchanged in (12), we only have to replace $P_2 \rightarrow \phi$ in (13b). Note also that if $P_2 \wedge P_1$ is FALSE, there is no need to actually add the clause $P_2 \wedge P_1 \rightarrow \phi[t_2/t_1]$ since no unsatisfiable constraints can be derived from it. Similarly if $P_2 \wedge \neg P_1$ is FALSE there is no need to add $P_2 \wedge \neg P_1 \rightarrow \phi$.

Example

The following example illustrates how this algorithm works. Suppose that (15) is the contexted version of (14):

(14) $[f_2 = f_1 \vee (f_1\ a) = c_1] \wedge [(f_2\ a) = c_2 \vee (f_1\ a) = c_3]$ where $c_i \neq c_j$ for all $i \neq j$

(15) a. $p_1 \rightarrow f_2 = f_1$
b. $\neg p_1 \rightarrow (f_1\ a) = c_1$
c. $p_2 \rightarrow (f_2\ a) = c_2$
d. $\neg p_2 \rightarrow (f_1\ a) = c_3$

(For clarity, we omit the ∧'s whenever contexted constraints are displayed in a column.) There is an applicable rewrite rule for constraints (15a) and (15c) that produces three new constraints:

(16)
$$\begin{array}{lll} p_1 \rightarrow f_2 = f_1 & \Leftrightarrow & p_1 \rightarrow f_2 = f_1 \\ p_2 \rightarrow (f_2\ a) = c_2 & & \neg p_1 \wedge p_2 \rightarrow (f_2\ a) = c_2 \\ & & p_1 \wedge p_2 \rightarrow (f_1\ a) = c_2 \end{array}$$

Although there is an applicable rewrite rule for (15d) and the last clause of (16), we ignore it since $p_1 \wedge p_2 \wedge \neg p_2$ is FALSE. The only other pair of constraints that can be rewritten are (15b) and (15d), producing three more constraints:

(17)
$$\begin{array}{lll} \neg p_1 \rightarrow (f_1\ a) = c_1 & \Leftrightarrow & \neg p_1 \rightarrow (f_1\ a) = c_1 \\ \neg p_2 \rightarrow (f_1\ a) = c_3 & & p_1 \wedge \neg p_2 \rightarrow (f_1\ a) = c_3 \\ & & \neg p_1 \wedge \neg p_2 \rightarrow c_1 = c_3 \end{array}$$

Since no more rewrites are possible, the normal form of (15) is thus:

(18)
$$\begin{array}{ll} \text{a.} & p_1 \rightarrow f_2 = f_1 \\ \text{b.} & \neg p_1 \rightarrow (f_1\ a) = c_1 \\ \text{c.} & \neg p_1 \wedge p_2 \rightarrow (f_2\ a) = c_2 \\ \text{d.} & p_1 \wedge \neg p_2 \rightarrow (f_1\ a) = c_3 \\ \text{e.} & p_1 \wedge p_2 \rightarrow (f_1\ a) = c_2 \\ \text{f.} & \neg p_1 \wedge \neg p_2 \rightarrow c_1 = c_3 \end{array}$$

Extracting the Disjunctive Residue

When the rewriting algorithm is finished, all unsatisfiable combinations of base constraints will have been derived. But more reasoning must be done to determine from base unsatisfiabilities whether the disjunctive system is unsatisfiable. Consider the contexted constraint $P \rightarrow \phi$, where ϕ is unsatisfiable. In order for the conjunction of contexted constraints to be satisfiable, it must be the case that $\neg P$ is true. We call $\neg P$ a *nogood*, following deKleer's terminology[1]. Since P contains propositional variables indicating disjunctive choices, information about which conjunctions of base constraints are unsatisfiable is thus back-propagated into information about the unsatisfiability of the conjunction of the disjuncts that they come from. The original system as a whole is satisfiable just in case the conjunction of all its nogoods is true. We call the conjunction of all of the nogoods the *residue* of the disjunctive system.

For example, clause (18f) asserts that $\neg p_1 \wedge \neg p_2 \rightarrow c_1 = c_3$. But $c_1 = c_3$ is unsatisfiable, since we know that $c_1 \neq c_3$. Thus $\neg(\neg p_1 \wedge \neg p_2)$ is a nogood. Since $c_1 = c_3$ is the only unsatisfiable base constraint in (18), this is also the disjunctive residue of the system. Thus (14) is satisfiable because $\neg(\neg p_1 \wedge \neg p_2)$ has at least one solution (e.g. p_1 is true and p_2 is true).

Since each nogood may be a complex boolean expression involving conjunctions, disjunctions and negations of propositional variables, determining whether the residue is satisfiable may not be easy. In fact, the problem is NP complete. However, we have accomplished two things by reducing a disjunctive system to its residue. First, since the residue only involves propositional variables, it can be solved by propositional reasoning techniques (such as deKleer's ATMS) that do not require specialized knowledge of the problem domain. Second, we believe that for the particular case of linguistics, the final residue will be simpler than the original disjunctive problem. This is because the disjunctions introduced from different parts of the sentence usually involve different attributes in the feature structure, and thus they tend not to interact.

Another way that nogoods can be used is to reduce contexts while the rewriting is being carried out, using identities like the following:

(19) $\neg P_1 \wedge (\neg P_1 \wedge P_2 \rightarrow \phi) \Leftrightarrow \neg P_1 \wedge (P_2 \rightarrow \phi)$

(20) $\neg P_1 \wedge (P_1 \wedge P_2 \rightarrow \phi) \Leftrightarrow \neg P_1$

(21) $P_1 \wedge \neg P_1 \Leftrightarrow$ FALSE

Doing this can improve the performance since some contexts are simplified and some constraints are eliminated altogether. However, the overhead of comparing the nogoods against the contexts may outweigh the potential benefit.

Producing the Models

Assuming that there is a method for producing a model for a conjunction of base constraints, we can produce models from the contexted system. Every assignment of truth values to the propositional variables introduced in (1) corresponds to a different conjunction of base constraints in the original system, and each such conjunction is an element of the DNF of the original system. Rather than explore the entire space of assignments, we need only enumerate those assignments for which the disjunctive residue is true.

Given an assignment of truth values that is consistent with the disjunctive residue, we can produce a model from the contexted constraints by assigning the truth values to the propositional variables in the contexts, and then discarding those base constraints whose contexts evaluate to false. The minimal model for the remaining base constraints can be determined by inspection if the base constraints are in normal form, as is the case for rewriting algorithms. (Otherwise some deductions may have to be made to produce the model, but the system is guaranteed to be satisfiable.) This minimal model will satisfy the original disjunctive system.

Example

The residue for the system given in (18) is $\neg(\neg p_1 \wedge \neg p_2)$. This residue has three solutions : p_1 and p_2 both true, p_1 true and p_2 false, and p_1 false and p_2 true. We can produce models for these solutions by extracting the appropriate constraints from (18), and reading off the models. Here are the solutions for this system:

	solution:	*constraints:*	*model:*
(22)	p_1 true, p_2 true:	$f_2 = f_1 \wedge (f_1\ a) = c_2$	$\begin{matrix} f1 \\ f2 \end{matrix}\begin{bmatrix} a & c2 \\ & \end{bmatrix}$
(23)	p_1 true, p_2 false:	$f_2 = f_1 \wedge (f_1\ a) = c_3$	$\begin{matrix} f1 \\ f2 \end{matrix}\begin{bmatrix} a & c3 \\ & \end{bmatrix}$
(24)	p_1 false, p_2 true:	$(f_1\ a) = c_1 \wedge (f_2\ a) = c_2$	$f1\begin{bmatrix} a & c1 \end{bmatrix}$ & $f2\begin{bmatrix} a & c2 \end{bmatrix}$

Comparison with Other Techniques

In this section we compare disjunctive constraint satisfaction with some of the other techniques that have been developed for dealing with disjunction as it arises in grammatical processing. These other techniques are framed in terms of feature-structure unification and a unification version of our approach would facilitate the comparisons. Although we do not provide a detailed specification of context-extended unification here, we note that unification can be thought of as an indexing scheme for rewriting. We start with a simple illustration of how such an indexing scheme might work.

Unification Indexing

Regarding unification as an indexing scheme, the main question that needs to be answered is where to index the contexts. Suppose that we index the contexts with the values under the attributes. Then the attribute-value (actually, attribute-*context*-value) matrix for (25a) would be (25b):

(25) a. $(f\,a) = c_1 \vee ((f\,b) = c_2 \vee (f\,a) = c_3)$ b.

$$\begin{bmatrix} a & \begin{bmatrix} p1 & c1 \\ -p1\&-p2 & c3 \end{bmatrix} \\ b & \begin{bmatrix} -p1\&p2 & c2 \end{bmatrix} \end{bmatrix}$$

Since the contexts are indexed under the attributes, two disjunctions will only interact if they have attributes in common. If they have no attributes in common, their unification will be linear in the number of attributes, rather than multiplicative in the number of disjuncts. For instance, suppose that (26b) is the attribute value matrix for (26a):

(26) a. $(f\,c) = c_4 \vee ((f\,d) = c_5 \vee (f\,e) = c_6)$ b.

$$\begin{bmatrix} c & \begin{bmatrix} p3 & c4 \end{bmatrix} \\ d & \begin{bmatrix} -p3\&p4 & c5 \end{bmatrix} \\ e & \begin{bmatrix} -p3\&-p4 & c6 \end{bmatrix} \end{bmatrix}$$

Since these disjunctions have no attributes in common, the attribute-value matrix for the conjunction of (25a) and (26a) will be simply the *concatenation* of (25b) and (26b):

(27)

$$\begin{bmatrix} a & \begin{bmatrix} p1 & c1 \\ -p1\&-p2 & c3 \end{bmatrix} \\ b & \begin{bmatrix} -p1\&p2 & c2 \end{bmatrix} \\ c & \begin{bmatrix} p3 & c4 \end{bmatrix} \\ d & \begin{bmatrix} -p3\&p4 & c5 \end{bmatrix} \\ e & \begin{bmatrix} -p3\&-p4 & c6 \end{bmatrix} \end{bmatrix}$$

The DNF approach to this problem would produce nine f-structures with eighteen attribute-value pairs. In contrast, our approach produces one f-structure with eleven attribute-value or context-value pairs. In general, if disjunctions have independent attributes, then a DNF approach is exponential in the number of disjunctions, whereas our approach is linear. This independence feature is very important for language processing, since, as we have suggested, disjunctions from different parts of a sentence usually constrain different attributes.

Karttunen's Disjunctive Values

Karttunen[7] introduced a special type of value called a "disjunctive value" to handle certain types of disjunctions. Disjunctive values allow simple disjunctions such as:

(28) $(f\,\mathrm{CASE}) = \mathrm{ACC} \vee (f\,\mathrm{CASE}) = \mathrm{NOM}$

to be represented in the unification data structure as:

(29) [CASE {ACC NOM}]

where the curly brackets indicate a disjunctive value. Karttunen's disjunctive values are not limited to atomic values, as the example he gives for the German article "die" shows:

(30) die =

$$\begin{bmatrix} \begin{bmatrix} \mathrm{CASE} & \{\mathrm{NOM\ ACC}\} \\ \mathrm{AGR} & \left\{ \begin{bmatrix} \mathrm{GENDER} & \mathrm{FEM} \\ \mathrm{NUMBER} & \mathrm{SG} \end{bmatrix}, \begin{bmatrix} \mathrm{NUMBER} & \mathrm{PL} \end{bmatrix} \right\} \end{bmatrix} \\ \mathrm{INFL} \end{bmatrix}$$

The corresponding attribute-context-value matrix for our scheme would be:

(31) $$\text{die} = \begin{bmatrix} \text{INFL} & \begin{bmatrix} \text{CASE} & \begin{bmatrix} p1 & \text{NOM} \\ \neg p1 & \text{ACC} \end{bmatrix} \\ \text{AGR} & \begin{bmatrix} \text{GENDER} & \begin{bmatrix} p2 & \text{FEM} \end{bmatrix} \\ \text{NUMBER} & \begin{bmatrix} p2 & \text{SG} \\ \neg p2 & \text{PL} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The advantage of disjunctive constraint satisfaction is that it can handle all types of disjunctions, whereas disjunctive values can only handle atomic values or simple feature-value matrices with no external dependencies. Furthermore, disjunctive constraint satisfaction can often do better than disjunctive values for the types of disjunctions that they can both handle. This can be seen in (31), where disjunctive constraint satisfaction has pushed a disjunction further down the AGR feature than the disjunctive value approach in (30). This means that if AGR were given an attribute other than GENDER or NUMBER, this new attribute would not interact with the existing disjunction.

However, disjunctive values may have an advantage of reduced overhead, because they do not require embedded contexts and they do not have to keep track of nogoods. It may be worthwhile to incorporate disjunctive values in our scheme to represent the very simple disjunctions, while disjunctive constraint satisfaction is used for the more complex disjunctions.

Kasper's Successive Approximation

Kasper[8, 9] proposed that an efficient way to handle disjunctions is to do a step-wise approximation for determining satisfiability. Conceptually, the step-wise algorithm tries to find the inconsistencies that come from fewer disjuncts first. The algorithm starts by unifying the non-disjunctive constraints together. If the non-disjunctive constraints are inconsistent, then there is no need to even consider the disjunctions. If they are consistent, then the disjuncts are unified with them one at a time, where each unification is undone before the next unification is performed. If any of these unifications are inconsistent, then its disjunct is discarded. Then the algorithm unifies the non-disjunctive constraints with all possible pairs of disjuncts, and then all possible triples of disjuncts, and so on. (This technique is called "k-consistency" in the constraint satisfaction literature[3].) In practice, Kasper noted that only the first two steps are computationally useful, and that once bad singleton disjuncts have been eliminated, it is more efficient to switch to DNF than to compute all of the higher degrees of consistency.

Kasper's technique is optimal when most of the disjuncts are inconsistent with the non-disjunctive constraints, or the non-disjunctive constraints are themselves inconsistent. His scheme tends to revert to DNF when this is not the case. Although simple inconsistencies are prevalent in many circumstances, we believe they become less predominate as grammars are extended to cover more and more linguistic phenomena. The coverage of a grammar increases as more options and alternatives are added, either in phrasal rules or lexical entries, so that there are fewer instances of pure non-disjunctive constraints and a greater proportion of inconsistencies involve higher-order interactions. This tendency is exacerbated because of the valuable role that disjunctions play in helping to control the complexity of broad-coverage grammatical specifications. Disjunctions permit constraints to be formulated in local contexts, relying on a general global satisfaction procedure to enforce them in all appropriate circumstances, and thus they improve the modularity and manageability of the overall grammatical system. We have seen this trend towards more localized disjunctive specifications particularly in our developing LFG grammars, and have observed a corresponding reduction in the number of disjuncts that can be eliminated using Kasper's technique. On the other hand, the number of independent disjunctions, which our approach does best on, tends to go up as modularity increases.

One other aspect of LFG grammatical processing is worth noting. Many LFG analyses are ruled out not because they are inconsistent, but rather because they are incomplete. That is, they fail to have an

attribute that a predicate requires (e.g. the object is missing for a transitive verb). Since incomplete solutions cannot be ruled out incrementally (an incomplete solution may become complete with the addition of more information), completeness requirements provide no information to eliminate disjuncts in Kasper's successive approximation. These requirements can only be evaluated in what is effectively a disjunctive normal form computation. But our technique avoids this problem, since independent completeness requirements will be simply additive, and any incomplete contexts can be easily read off of the attribute-value matrix and added to the nogoods before solving the residue.

Kasper's scheme works best when disjuncts can be eliminated by unification with non-disjunctive constraints, while ours works best when disjunctions are independent. It is possible to construct a hybrid scheme that works well in both situations. For example, we can use Kasper's scheme up until some critical point (e.g. after the first two steps), and then switch over to our technique instead of computing the higher degrees of consistency.

Another, possibly more interesting, way to incorporate Kasper's strategy is to always process the sets of constraints with the fewest number of propositional variables first. That is, if $P_3 \wedge P_4$ had fewer propositional variables than $P_1 \wedge P_2$, then the rewrite rule in (32b) should be done before (32a):

(32) a. $(P_1 \rightarrow \phi_1) \wedge (P_2 \rightarrow \phi_2) \Rightarrow (P_1 \wedge P_2 \rightarrow \phi_5)$

b. $(P_3 \rightarrow \phi_3) \wedge (P_4 \rightarrow \phi_4) \Rightarrow (P_3 \wedge P_4 \rightarrow \phi_6)$

This approach would find smaller nogoods earlier, which would allow combinations of constraints that depended on those nogoods to be ignored, since the contexts would already be known to be inconsistent.

Eisele and Dörre's techniques

Eisele and Dörre[2] developed an algorithm for taking Karttunen's notion of disjunctive values a little further. Their algorithm allows disjunctive values to be unified with reentrant structures. The algorithm correctly detects such cases and "lifts the disjunction due to reentrancy". They give the following example:

(33)
$$\left[a:\ \left\langle \begin{bmatrix} b: & + \\ c: & - \end{bmatrix}, \begin{bmatrix} b: & - \\ c: & + \end{bmatrix} \right\rangle\right] \cup \begin{bmatrix} a: & [b: \langle d \rangle] \\ d: & [\,] \end{bmatrix} = \left\langle \begin{bmatrix} a: & [b: \langle d \rangle] \\ & [c: & -] \\ d: & + \end{bmatrix}, \begin{bmatrix} a: & [b: \langle d \rangle] \\ & [c: & +] \\ d: & - \end{bmatrix} \right\rangle$$

Notice that the disjunction under the "a" attribute in the first matrix is moved one level up in order to handle the reentrancy introduced in the second matrix under the "b" attribute.

This type of unification can be handled with embedded contexts without requiring that the disjunction be lifted up. In fact, the disjunction is moved down one level, from under "a" to under "b" and "c":

(34)
$$\left[a:\ \begin{bmatrix} b: & \begin{bmatrix} p1 & + \\ \neg p1 & - \end{bmatrix} \\ c: & \begin{bmatrix} p1 & - \\ \neg p1 & + \end{bmatrix} \end{bmatrix}\right] \cup \begin{bmatrix} a: & [b: \langle d \rangle] \\ d: & [\,] \end{bmatrix} = \left[a:\ \begin{bmatrix} b: & \langle d \rangle \\ c: & \begin{bmatrix} p1 & - \\ \neg p1 & + \end{bmatrix} \end{bmatrix}, \ d: \begin{bmatrix} p1 & + \\ \neg p1 & - \end{bmatrix}\right]$$

Overall

The major cost of using disjunctive constraint satisfaction is the overhead of dealing with contexts and the disjunctive residue. Our technique is quite general, but if the only types of disjunction that occur are covered by one of the other techniques, then that technique will probably do better than our

scheme. For example, if all of the nogoods are the result of singleton inconsistencies (the result of unifying a single disjunct with the non-disjunctive part), then Kasper's successive approximation technique will work better because it avoids our overhead. However, if many of the nogoods involve multiple disjuncts, or if some nogoods are only produced from incomplete solutions, then disjunctive constraint satisfaction will do better than the other techniques, sometimes exponentially so. We also believe that further savings can be achieved by using hybrid techniques if the special cases are sufficiently common to warrant the extra complexity.

Acknowledgements

The approach described in this paper emerged from discussion and interaction with a number of our colleagues. We are particularly indebted to John Lamping, who suggested the initial forumulation of the basic lemma, and to Bill Rounds for pointing out the relationship between our conversion algorithm and the NP completeness reduction for CNF. We are also grateful for many helpful discussions with Dan Bobrow, Johan deKleer, Jochen Dörre, Andreas Eisele, Pat Hayes, Mark Johnson, Lauri Karttunen, and Martin Kay.

References

[1] deKleer, J. (1986). An Assumption-based TMS. *Artificial Intelligence* **28**, 127-162.

[2] Eisele, A. and Dörre, J. (1988). Unification of Disjunctive Feature Descriptions. *Proceedings of the 26th Annual Meeting of the ACL*. Buffalo, New York.

[3] Freuder, E.C. (1978). Synthesizing Constraint Expressions. *Communications of the ACM* **21**, 958-966.

[4] Hopcroft, J. and Ullman, J. (1979). *Introduction to Automata Theory, Languages and Computation*. p. 328-330.

[5] Johnson, M. (1988). *Attribute-Value Logic and the Theory of Grammar*. Ph.D. Thesis. Stanford University.

[6] Kaplan, R. and Bresnan, J. (1982). Lexical Functional Grammar: A Formal System for Grammatical Representation. In J. Bresnan (ed.), *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge, Massachusetts.

[7] Karttunen, L. (1984). Features and Values. *Proceedings of COLING 1984*, Stanford, CA.

[8] Kasper, R.T. (1987). *Feature Structures: A Logical Theory with Application to Language Analysis*. Ph.D. Thesis. University of Michigan.

[9] Kasper, R.T. (1987). A Unification Method for Disjunctive Feature Descriptions. *Proceedings of the 25th Annual Meeting of the ACL*, Stanford, CA.