PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

FOUNDATIONS OF DISLOG, PROGRAMMING IN LOGIC WITH DISCONTINUITIES

Patrick Saint-Dizier

IRISA-INRIA, Campus de Beaulieu, 35042
RENNES Cedex, FRANCE

Abstract

In this paper we present an extension to PROLOG we call DISLOG which is designed to deal with relations between non-contiguous elements in a structure. This extension turns out to be well suited for syntactic analysis of natural and artificial languages. It is also well adapted to express traversal constraints in applications such as planning and expert systems and deductive systems involving, for example, temporal reasoning. DISLOG turns out to be more declarative, transparent and simple than PROLOG to deal with long distance relations.

1 Introduction to DISLOG

In this document, we present DISLOG, an extension to PROLOG. DISLOG stands for Discontinuities in Logic because it has emerged from a generalization to logic programs of Contextual Discontinuous Grammars (Saint-Dizier 87) designed to deal with movements in natural language parsing systems. This grammar formalism is both a specialization and a generalization of a global framework proposed by V. Dahl (Dahl 87) called Static Discontinuity grammars.

The discontinuous family of logic grammars (Dahl and Abramson 84, Dahl 85, Dahl and Saint-Dizier 86) - named at first Gapping Grammars- was originally jointly developed by V. Dahl, H. Abramson and M. McCord. The idea was to allow for unidentified strings of symbols in a rule (refered to through a symbol skip(X)) to be arbitrarily repositioned. Several theoretical as well as practical problems with this first definition lead to a more constrained definition of a discontinuous grammar. Work on Government and Binding Theory produced the idea of keeping the type-0 power of Discontinuous grammars while maintaining the simplicity of context-free parse trees. This new type of grammar is now called Static Discontinuity Grammars by V. Dahl (Dahl 87). A Static Discontinuous rule is a rewriting rule in the Metamorphosis Grammar style, it looks like:

a, *skip(X)*, b, *skip(Y)*, c → a1, *skip(X)*, b1, *skip(Y)*, c1.

where a, b, c, a1, b1 and c1 are non-terminal symbols. The symbol skip(X) stands for an unknown string of symbols of no present interest.

The definition given in (Dahl 87) is of much interest, but it is however vague and its declarative meaning can be understood in a number of ways. Some elements of the formalism can be simplified while others can be generalized. In (Saint-Dizier 87), we generalize and make more explicit this framework and we also provide more linguistic motivations and an implementation in PROLOG. We call our formalism Contextual Discontinuous Grammars. Very briefly, its main characteristics and advantages with respect to Static Discontinuous Grammars are that a1, b1 and c1 are a sequence of terminal and non-terminal symbols; the skip notation is given up and derivations of a, b and c into respectively a1, b1 and c1 are given, in a principled way, a much greater freedom; modalities and the possibility of specifying linear precedence restrictions have also been added. This is described in detail in (Saint-Dizier 87).

From the principles of Contextual Discontinous Grammars we designed DISLOG. The idea of discontinuity in grammars became the more general problem of expressing constraints or relations between non-contiguous elements in a structure (Saint-Dizier 88b). In the above mentioned paper, we motivate from a psychological point of view the idea of thinking and reasoning in terms of such relations and constraints. This is illustrated by reasoning about temporal events and by the expression of meaning postulated like those encountered in Montague semantics. This generalization can also be viewed as imposing constraints on the way a problem can be solved by controlling the form of the proof tree.

DISLOG turns out to be particularly well adapted to express in a modular, concise, simple and transparent way constraints and relations between non-contiguous elements in a structure. We call those constraints and relations *traversal* constraints and relations to differenciate them from the more common hi-

erarchical ones.

In this paper we first informally introduce the basic concepts of DISLOG and then present some of its major formal aspects. Classes of applications for which DISLOG is appropriate are presented in (Saint-Dizier 88a).

2 Basic Concepts of DISLOG

The two statements specific to DISLOG are DISLOG facts and DISLOG rules. We now present and illustrate them.

2.1 DISLOG facts

A DISLOG fact is a finite, unordered set of PROLOG facts f_i of the form:

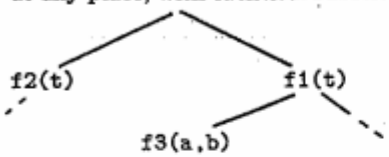
$\{ f_1, f_2, \ldots, f_n \}$.

The informal meaning of a DISLOG fact is: *if f_i is used as a leaf in a given proof tree, then all the f_j must also be used as leaves in that proof tree. There is a priori no occurence order for facts f_i but the same substitutions have to be applied to all identical variables that appear in a DISLOG fact.*

For example, if we have the following DISLOG fact:

$\{ f_1(X), f_2(X), f_3(a,b) \}$.

then, if one of the facts f_i is used to build a proof tree, with, for example t being substituted for X, then, the other two facts must also appear in that proof tree, at any place, with identical substitutions, as in:



DISLOG facts can be used in regular PROLOG rule bodies as shall be seen in the forthcoming example or in the body of rules in a DISLOG rule, as shall be presented in the next subsection. Facts f_i in a DISLOG fact can also be given as regular facts in a program, independently of the co-occurence constraints. In that case, we have a definition of f_i in two parts:

```
{ f1, f2, f3 }.
f1.
```

Discontinous facts seem to be mainly useful to express traversal constraints. For instance, the DISLOG fact:

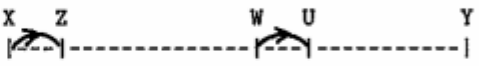
$\{$ arc(a,b), arc(c,d) $\}$.

means that all path going through arc from a to b must also go through arc from c to d, or conversely since we impose the co-occurence of the two facts in a proof, but with no constraint on use order. This traversal constraint in a graph cannot be expressed very easily in PROLOG: additional arguments in the arc predicate and a checking procedure are necessary. Using DISLOG facts considerably enhance the transparency and the conciseness of the program.

Here is the program that checks whether there is a path between two nodes in a graph with possible traversal constraints:

```
{arc(a,b), arc(c,d)}.
arc(c,e).
arc(b,c).

/% regular PROLOG program    %/

path(X,Y) :- arc(X,Y).
path(X,Y) :- arc(X,Z), path(Z,Y).

/%  additional rules   %/

path(X,Y) :-
   {arc(X,Z), arc(W,U)},
   path(Z,W),
   path(U,Y).

path(X,Y) :-  %case where the second arc is
the last arc.
   {arc(X,Z), arc(W,Y)},
   path(Z,W).
```

The first rule which contains a DISLOG fact states that there is a path from X to Y if there is an arc from X to Z and and arc from W to U and a path between Z and W and between U and Y. This can be diagrammatically represented as follows:



2.2 DISLOG rules.

The basic form of a DISLOG rule is a finite, unordered set of PROLOG rules r_i of the form:

$\{ r_1, r_2, \ldots, r_m \}$.

Informally, the procedural meaning of a DISLOG rule is: *if r_i is used in a proof tree, then all the other r_j in that DISLOG rule must be used to build that proof tree. If identical variables appear in different r_i, for any $i \in [1,n]$, then the same substitutions must be applied to them. The r_i can a priori appear at any level and in any order in the proof tree.* DISLOG rules are exemplified in section 3.

A PROLOG rule can be interpreted as a logical axiom. This also holds for a DISLOG rule. If X1, X2, ... Xm is the set of free variables that appear in the r_i, for all $i \in [1,n]$, then the declarative meaning of a DISLOG rule is of the following general form:

V X1, X2, .. , Xm, $p(r_1) \wedge p(r_2) \wedge .. \wedge p(r_n)$;

$p(r_j)$ being the declarative reading of r_j.

2.3 DISLOG programs

A DISLOG program is then a finite set of PROLOG facts and rules and of DISLOG facts and rules. The declarative meaning of a DISLOG program D is the set of ground unit goals deducible from D.

PROLOG rules can contain calls to DISLOG facts and calls to rules in DISLOG rules in their bodies. Rules in DISLOG rules can also have such calls.

Rules r_i as well as facts f_i in DISLOG rules and facts can, in addition, be defined respectively as regular PROLOG rules and facts. This is exemplified in section 2.1.

2.4 More on DISLOG Rules.

We now propose some simple variants and restrictions on the general form of DISLOG facts and rules.

A first type of restriction is to impose some restrictions on the order of use of rules r_i in a DISLOG rule. In many applications, at least a partial ordering is required. We say that a rule r_i precedes another rule r_j if it appears:

- either to the left of r_j in the proof tree,
- or if r_i dominates r_j in the proof tree.

We have the two following possible diagrams if r_i is the rule a → a1 and r_j is b → b1. :

Notice that this definition of precedence does not depend on the strategy used to prove a goal. A way to represent partial order restrictions is to use **linear precedence restrictions.** The statement r_i *precedes* r_j is noted as:

$r_i < r_j$

Thus, a restricted DISLOG rule is of the form:

$\{ r_1, r_2, \ldots, r_n \}\ r_i < r_j \ldots r_k < r_m$.

There are, in fact, a number of cases where a total order is required. Instead of specifying all the ordering restrictions, we prefer to adopt a new notation. A rule like the above with a complete ordering is noted as:

$r_1 / r_2 / \ldots / r_n$.

In this DISLOG rule, any r_i precedes any r_j if j > i. The symbol / can then be viewed as an accessibility relation: r_2 is accessible as soon as r_1 is activated.

Another improvement is to introduce **modalities.** The main modality we need in the present development of DISLOG is the possibility, noted m (we use a small letter to avoid confusion with the variable notation). The modality m can be attached to any rule r_i in a DISLOG rule; it allows this rule r_i to be used any number of times, including zero, in the context of that DISLOG rule. For instance, we could have the DISLOG rule:

$\{ r_1, m(r_2) \}$.

in which r_2 can be used any number of times provided r_1 is used once. Introducing the modality m permits to considerably enhance, for instance, the ease of writing compilers for programming languages. In natural language parsing and generation, modality m permits to deal with cordination (r_2 represents coordinated items) and with parasitic gaps in a very concise and convenient way.

3 A Compiler in DISLOG

In (Saint-Dizier 88), we present several classes of applications for which DISLOG is appropriate. Among those applications we have: planning techniques, compiling, semantic representation computation, problems that can be represented by graph and exhibiting traversal constraints (8 queens, sorting, etc...) and temporal reasoning. We now briefly present a simple example: writing a compiler in DISLOG.

In a conventional programming language, there are several one-to-one or one-to-many relations between non-contiguous instructions. For instance there is a relation between a procedure declaration and its corresponding calls and another relation between a label and its corresponding branching instructions. DISLOG rule format is very well suited to express those relations, permitting variables to be shared between several rules in a DISLOG rule. These variables can percolate, for instance, addresses of entry points.

We now consider the compiler given in (Sterling and Shapiro 86) that transforms a program written in a simplified version of PASCAL into a set a basic instructions (built in the argument). This small compiler can be augmented by the two pairs:

{ *procedure declaration , procedure call(s)* }.

{ *label statement , branching instruction(s) to stated label* }.

This is expressed by two DISLOG rules. In order to allow for a procedure call to appear before the declaration of the corresponding procedure we do not state any restriction on linear precedence. Furthermore, procedure calls and branching instructions description rules are marked with modality m since a procedure call or a branching instruction to a given label may appear several times in a program. Thus, we have the following two pairs:

{ *procedure declaration, m(procedure call)* } .

{ *label statement,m(branching instruction to stated label)*}.

In a parse tree corresponding to the syntactic analysis of a PASCAL-like program, we could have, for example, the following tree:

```
    proc_call(Address)

proc_decl(Address,Code)


                          proc_call(Address)
```

The main calls and the DISLOG rules are the following:

```
parse(Structure) -->
   [program], identifier(X), [';'],
statement(Structure).
statement((S;Ss)) -->
   [begin], statement(S), rest_statement(Ss).
statement(assign(X,V)) -->
   identifier(X), [':='], expression(V).
rest_statement((S;Ss)) -->
   [';'], statement(S), rest_statement(Ss).
rest_statement(void) -->  [end].
/* Procedure declaration and call   */
{ (statement(proc_decl(N,S))  -->
[procedure],
       identifier(N), statement(S), [end] ) ,
  m(statement(proc_call(N,S) -->
identifier(N) ) }.
/*   label statement and branching
instructions   */
{ (statement(label(N)) -->  identifier(N),
    [':'], ),
  m(statement(goto(N)) --> [goto],
identifier(N)) }.
```

Another advantage of using DISLOG rules is that if a compilation fails because some rules in active DISLOG rules (rules where some PROLOG rules have not yet been executed) have not been executed when the parsing is terminated, then it is quite easy to locate the errors and to produce informative messages. In addition, DISLOG rules with rules marked with modality m can be useful to produce warning messages, for instance when a PROLOG rule with modality m has not been used, although this is not forbidden. For example, when a procedure is declared and never called, it is advisable to produce some form of warning.

4 DISLOG and Bounding theory

The above compiler is, in fact, too permissive because, for example, it accepts a label declared inside a procedure to be refered to outside it. This type of problem is well-known in linguistics and is refered to as **bounding theory**. This theory states constraints on the way to move constituents or on the way to establish relations between non-contiguous constituents. The main type of constraint is expressed in terms of domains over the boundaries of which long distance dependencies cannot be established. This problem has already been addressed in detail in (Saint-Dizier 85, Dahl and Saint-Dizier 86) for natural language parsing. Roughly speaking if A is a bounding node, then the domain of A is the subtree it is the root of and no constituent X dominated by A can establish relations with any element B not dominated by A (or, in other terms, outside the domain of A):

This approach can be used for formal languages as well. To constraint the above program, we can state that the node:

statement(proc_decl(_,_))

in a proof tree is a bounding node. We note this as a PROLOG fact:

```
bounding_node(statement(proc_decl(_,_))).
```

The above example states that, in a syntactic tree, the node corresponding to a procedure declaration is a bounding node. As a consequence, all DISLOG rules called within the domain dominated by that node has to be fully executed within that domain. This will prevent a label declared inside a procedure body from being refered to outside that procedure. The DISLOG meta-interpreter we have implemented treats facts like *bounding_node( )* as a special directive. Notice that bounding nodes are given apart from the program, introducing thus a higher degree of explanatory adequacy, modularity and declarativity.

5 Foundations of DISLOG

Here are the basic theoretical foundations of DISLOG. Most of the material presented here has emerged from a reformulation of definitions and theorems given in [Lloyd 87]. We consider here the simpler definition of DISLOG, i.e. without modalities and linear precedence restrictions. Modalities and precedence restrictions do not however introduce fundamental differences in the logical foundations of DISLOG.

5.1 DISLOG definite programs

Definition: a DISLOG definite program clause is a finite, unordered set of program definite clauses of the form:

{ *(A ← A1, A2, ..., An), (B ← B1, B2, ..., Bm), ... , (N ← N1, N2, ..., Nj)* }.

Definition: a DISLOG unit clause is a finite, unordered set of unit clauses of the form:

{ *(A ←), (B ←), ... , (N ←)* }.

The informal semantics of a DISLOG definite program clause is: *for each assignment of each variable occuring in that definite program clause, if A1, ..., An, B1, ..., Bm, N1, ..., Nj are all true then A, B, ... , N are true.*

Definition: a DISLOG definite program is a finite set of DISLOG definite program clauses.

Notice that a definite program clause is a DISLOG definite program clause with a single element. There does not exist DISLOG definite goals with more than one element.

Definition: the definition of the predicate p in a DISLOG program is the set of all DISLOG definite program clauses which contain at least one definite program clause with head predicate symbol p.

Here is an example of a possible definition for p:

{ *(p ←)* }.

{ *(a ← t), m(p ← b)* }.

5.2 Herbrand interpretation revisited

Each program definite clause in a DISLOG definite program clause is a well-formed formula of a first-order language L. We can view a DISLOG definite program clause as the specification of a **co-occurence constraint** in a proof tree of a finite number of definite clauses.

This co-occurence constraint has two aspects:

- co-occurence of definite clauses,
- variables shared by definite clauses are subject to identical substitutions (throughout the whole parse).

A DISLOG program is then a definite program with a finite set of co-occurence constraint specifications. From this point of view, which does not introduce any restriction, we now reformulate the notion of Herbrand interpretation to meet the requirements of DISLOG.

In DISLOG, the traditional definitions of Herbrand universe and Herbrand base remain unchanged. The existence of identical substitutions applied to function and predicate symbols subject to the co-occurence constraint is indeed guaranteed by definition of the Herbrand universe and base. These two sets respectively contain all the possible ground terms and atoms which can be built out from all constants and respectively from all function and predicate symbols. The same remark holds for the Herbrand pre-interpretation.

The Herbrand interpretation of a DISLOG program based on a first-order language L is a subset of the Herbrand base. This subset is the set of all ground atoms which are true in this interpretation, given the co-occurence constraints. This subset is included in the Herbrand interpretation of the same program without constraints. For example, if we consider the program *path* given in section 1, the ground atom *path(a,e)* is in the Herbrand interpretation of the definite program without co-occurence constraints (arc(c,d) is not used) and it is not in the interpretation with co-occurence constraints.

Finally, let B_D be the Herbrand base of a DISLOG program. For the same reasons as with definite programs 2^{B_D} (the set of all Herbrand interpretations of D) forms a complete lattice under the partial order of set inclusion $\subseteq$. The top element is B_D and the bottom element is $\emptyset$. The mapping T_D: $2^{B_D} \rightarrow 2^{B_D}$ is defined as follows:

I_D is an Herbrand interpretation and

$T_D(I_D) = \{ X_i \in B_D$ and X_i is any head symbol in a ground instance $\{ (X_i \leftarrow X_{1,1}, X_{1,2}, \ldots, X_{1,k}), \ldots (X_n \leftarrow X_{n,1}, X_{n,2}, \ldots, X_{n,f}) \}$ of a DISLOG definite program clause in D, $X_{j,k} \in I_D$ and there exists a proof tree P which can be built from the program and which includes all the X_i with appropriate substitutions $\}$.

T_D is monotonic, it is also continuous for the same reasons as for definite programs without co-occurence constraints.

5.3 Procedural semantics of DISLOG

We now introduce the procedural semantics of DISLOG programs. The basic principle is similar to that of definite programs, except that an additional data-structure, noted S, and called the set of rules to be applied in the current proof construction, is used. S originates from the use of DISLOG rules: when a definite rule in a new instance of a DISLOG rule is used,

for example:

b ← b1, b2. in:

{ (a ← a1, a2), (b ← b1, b2), (c ← c1) }.

then S is used to memorize that (a ← a1, a2) and (c ← c1) have to be applied in the current proof construction with the same substitutions. Each time a new instance of a DISLOG rule is used, S is updated. When a rule in S is used, this rule is withdrawn from S. In what follows, we consider a particular step i of the proof construction process, the set of rules to be applied represented by S is noted S_i at step i.

Definition: Let G_i be the goal: $\leftarrow A_1, A_2, \ldots, A_m, \ldots, A_k$ and D a DISLOG rule be: $\{ \alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n \}$ then G_{i+1} is derived from G_i and D using mgu θ_i if:

- (a) A_m is the selected atom in G_i,
- (b) the definite clause $\alpha_j \leftarrow \beta_j$ in the DISLOG definite clause is the rule used to reduce A_m,
- (c) θ_i is an mgu of A_m and α_q, $\forall\, q \in [1, n]$.
- (d) G_{i+1} is $(\leftarrow A_1, A_2, \ldots, A_{m-1}, \beta_j, A_{m+1}, \ldots, A_k)\theta_i$,
- (e) and we have for S_{i+1}:
  - (e1) $S_{i+1} = S_i \cup \{ (\alpha_1 \leftarrow \beta_1)\theta_i, \ldots, (\alpha_{j-1} \leftarrow \beta_{j-1})\theta_i, (\alpha_{j+1} \leftarrow \beta_{j+1})\theta_i, \ldots, (\alpha_n \leftarrow \beta_n)\theta_i \}$ if $\alpha_j \leftarrow \beta_j$ belongs to a new activated instance of a DISLOG rule.
  - (e2) $S_{i+1} = S_i - \{ (\alpha_j \leftarrow \beta_j) \}$ if α_j is an element of S_i.
  - (e3) $S_{i+1} = S_i$ if $\alpha_j \leftarrow \beta_j$ is not subject to any co-occurence constraint.

This definition calls for some remarks. In (c), notice that θ_i is the mgu of all the α_q and not only the mgu of α_i because all rules in the DISLOG rule (or fact) are affected by the substitution.

Another point is that if a rule (or a fact) r_i appears in several DISLOG rules, then the DISLOG rules are considered successively in their input order, in the same way as different clauses in a definition in a PROLOG program. The computation rule we have defined is thus very close to the regular PROLOG computation rule in its principle. Finally, in the case of a rule affected by a modality m, when the case (e2) is met, the rule is not withdrawn from S_i.

Definition: a proof in DISLOG is correct from the point of view of co-occurence constraints if the initial and final set of rules to be applied in the proof (noted as $S_{initial}$ and S_{final}) are both empty.

5.4 Constrained SLD-refutation

The DISLOG proof procedure is based on SLD-resolution. This SLD-resolution technique is however constrained by co-occurence restrictions. This motivates the term **constrained SLD-resolution.** We now turn to explore its main characteristics more in detail.

Definition: let D be a DISLOG program and G a definite goal. The constrained SLD-resolution of $D \cup \{G\}$ is a sequence of goals $G_{initial}$, G_1, G_2, ..., a sequence D1, D2, ... of variants of DISLOG program clauses of D, a sequence θ_1, θ_2, ... of mgu's and a sequence $S_{initial}$, S_1, S_2, ... of sets of clauses to be applied in the resolution such that:

- each G_{i+1} is derived from G_i and D_{i+1} using θ_{i+1},
- each S_{i+1} is constructed from S_i as explained in the preceding subsection.
- $S_{initial} = \emptyset$.

Definition: a constrained SLD-refutation of $D \cup \{G\}$ is a finite constrained SLD-derivation of $D \cup \{G\}$ which has:

1. the empty clause as the last goal in the derivation,
2. an empty set S of rules to be applied in the proof at the beginning and at the end of the refutation process (i.e. : $S_{initial} = S_{final} = \emptyset$).

Since (2) entails that in each proof co-occurence constraints are met, it follows that every computed answer for $D \cup \{G\}$ is a correct answer for $D \cup \{G\}$. This establishes the *soundness* of the constrained SLD-refutation of DISLOG programs.

5.5 Completeness of constrained SLD-resolution

We now show the completeness of the constrained SLD-resolution. It is based on the following considerations about DISLOG programs:

1. SLD-resolution is complete,
2. there exists a monotonic, continuous mapping T_D (cf. section 5.2),
3. every correct answer θ for $D \cup \{G\}$ is a correct answer for the same program without co-occurence constraints which satisfies the co-occurence constraints of D.

The success set ξ(D) of $D \cup \{G\}$ is included in the success set ξ(D') of $D' \cup \{G\}$ where D' is D without

co-occurence constraints. ξ(D) is deduced from ξ(D') by selecting those solutions in ξ(D') having a proof tree satisfying the co-occurence constraints of D. This construction method is a direct consequence of the definition of a DISLOG definite program (cf. section 5.2).

Then, for every correct answer θ, there exists a computed answer σ for $D \cup \{G\}$ and a substitution γ such that: $\theta = \sigma\gamma$.

6 More comparisons with related works

Dislog, as indicated in the introduction section, directly originates from Contextual Discontinuous Grammars (Saint-Dizier 87). Its application is however much wider. With respect to Static Discontinuity Grammars (Dahl 87), where a grammar rule is of the form:

a1, skip(X), a2, ..., skip(Z), an → b1, skip(X), b2, ..., skip(Z), bn.

which is a strict rewriting rule including variables (skip(X)) standing for symbols, Contextual Discontinuous Grammars are more general, are closer to the linguistic reality and have a clear declarative and procedural semantics. As opposed to Static Discontinuity Grammars, Contextual Discontinuous Grammars permit:

(1) to manipulate sets of type-2-like rules which can appear anywhere in a proof tree, whereas the rewriting rule format is much more constraining,

(2) to avoid the use of linguistically unmotivated extra-symbols (like skip, and symbols introduced to avoid loops),

(3) to limit the order of rules in proof trees by linear precedence restrictions. Other types of restrictions can also be used.

(4) to use several times a given rule, via the introduction of modalities.

Furthermore, the exact meaning of Static Discontinuity rules is unclear and subject to controversial understandings due to the lack of precise definitions.

Dislog permits to address a number of types of problems involving the expression of relations or constraints between non-contiguous elements in a structure. In (Dahl 87), it is shown how the problem of the dining philosophers can be handled by Static Discontinuity Grammars. However, instead of simplifying the expression of this very classical synchronization problem, the rules given are very complex and highly intertwined because of confusions between declarative and procedural programming. The outcome of the program, in terms of priorities among events, is unclear because it depends both on the specification given in rules and on the computation rule of the system, not given here.

In (Monteiro 82), L. Monteiro presents a system of distributed logic aimed at expressing and processing concurrency at a high abstract level. The formalism contains a type of clause, called *generalized clauses*, of the form:

a1, a2, ... , an ← s1, s2, ..., sn.

in which the atom ai is in correspondance with the body clause si. The rule format used is quite close in its spirit to ours, however, there are major differences with Dislog, among which:

(1) The application of a generalized rule requires the n goals ai to be simultaneously present in the list of goals to prove. Dislog offers more possibilities since goals can be embedded into others.

(2) No order of use of the goals ai is specified. We feel that many applications require the expression of some constraints on goal use order, whether linear or hierarchical.

(3) Dislog is not particularly designed to express concurrency. It does not contain any concurrent-sequential notations, although these notations could be easily added. Dislog is more general in the sense that it can deal with a large variety of types of constraints or relations between non-contiguous elements in a structure, among which we could have synchonization of processes.

7 Conclusion

In this document, we have presented the basic theoretical foundations of DISLOG, an extension to PROLOG designed to express in a transparent, modular and concise way relations and constraints between non-contiguous elements in a structure. The procedural semantics of DISLOG has been introduced and we have shown that it is sound and complete with respect to the constrained SLD-resolution mechanism. In (Saint-Dizier 88), we present several classes of applications for which DISLOG is appropriate. These applications clearly show that, although DISLOG is still in its early stage of development, it is a promising approach to deal with several kinds of problems involving the idea of expressing relations or constraints between non-contiguous elements in a structure.

We have designed and implemented two versions of DISLOG, discussed in (Saint-Dizier 87 and 88). The first implementation consist in adding two arguments to clauses in order to keep track of the rules to be used in the proof (i.e. S_i and S_{i+1}). The second implementation is a meta-interpreter, which uses the same idea. The implementation of bounding theory is much simpler in this latter case. A third implementation is now completed for natural language understanding

with a bottom-up strategy associated with a one step look-ahead mechanism and some heuristics. This latter implementation for natural language turns out to be about 2000 times more efficient than the first implementation. Work is under study to define an implementation involving partial parallel executions of DISLOG rules.

ACKNOWLEDGEMENTS

I am indebt to V. Dahl for several discussions in early 1987 on Discontinuous Grammars while I was visiting Simon Fraser University. I would also like to thank M. Borillo and P. Sebillot for their useful comments on a preliminary version of this text. Comments of four anonymous referees and by Dr. Koichi Furukawa have also been very helpful to prepare the final text. This work was supported by the INRIA and by the PRC-CNRS Communication Homme-Machine.

REFERENCES

DAHL, V., ABRAMSON, H., On Gapping Grammars, Proc. of the 3rd Logic Programming conference, Uppsala, 1984.

DAHL, V., More on Gapping Grammars, proc. of FGCS'85, Tokyo, 1985.

DAHL, V., BROWN, C., HAMILTON, S., Constrained Discontinuous Grammars and Logic Programming, Simon Fraser Research report LCCR TR 86-17, 1986; revised in December 1987 and reprinted under the title Static Discontinuity Grammars and Logic Grammars same reference as above, original date not modified however.

DAHL, V., SAINT-DIZIER, P., Constrained Discontinuous Grammars: A Linguistically Motivated Tool for Processing Language, IRISA-INRIA research report, 1986.

LLOYD, J., *Foundations of Logic Programming*, Springer-Verlag, 1984. revised 1987.

MONTEIRO, L., A Horn-Clause like Logic for Specifying Concurrency, in *Proc. of the first logic programming conference*, Marseille, 1982.

SAINT-DIZIER, P., Constraints on Long-Distance Dependencies, IRISA-INRIA research report, 1985.

SAINT-DIZIER, P., Contextual Discontinuous Grammars, Proc of the second NLULP, Vancouver BC, 1987 and in: *Natural Language Understanding and Logic Programming, II*, V.Dahl and P. Saint-Dizier Edts., North Holland, 1988.

SAINT-DIZIER, P., DISLOG: Programming in Logic with Discontinuities, *Computational Intelligence*, Vol 5-1, 1988.

SAINT-DIZIER, P., DISLOG: Thinking and Reasoning in terms of Discontinuities, *Proc. of the 3rd International Conference on Cognitive Science ARC'88*, Toulouse, 1988.

STERLING, J., SHAPIRO, E., *The Art of PROLOG*, MIT Press, Series in Computer Science, 1986.