COMMA 2014 Summer School

Argument Analysis in Dislog

Course notes, Patrick Saint-Dizier

IRIT-CNRS, Toulouse, France

stdizier@irit.fr

September 2014

Table of Contents

1. AN INTRODUCTION TO TEXTCOOP AND DISLOG

1.Introduction

In this chapter, we first introduce the foundational elements which are at the basis of the programming language Dislog and the platform <TextCoop> on which Dislog runs. Dislog has been primarily designed for discourse processing. It can also be used in other contexts where the recognition of complex patterns in texts is a central issue. Dislog is based on logic and logic programming and offers a declarative way of describing linguistic structures. It integrates reasoning components and offers a very modular way of describing language phenomena.

The Dislog language (Dislog stands for *Discourse in logic*, or discontinuities in logic since discourse structure analysis is based on marks which often are in long-distance dependency relations) is presented in detail in this chapter. The <TextCoop> platform is then introduced, in particular the <TextCoop> engine and the linguistic architecture of the system. Dislog is illustrated in the chapter that follow. Finally, we present and discuss in this chapter performance issues which are crucial since processing the discourse structure of texts is in general not very efficient due to the size of texts, which are often large, and the complexity and ambiguity of discourse structures. The <TextCoop> platform is freely available from the author.

There are at the moment a few well-known and widely used language processing environments. They are essentially used for sentence processing, not for discourse analysis. The reasons are essentially that sentences and their substructures are the main level of analysis for a large number of applications such as information extraction, opinion analysis based on the structure evaluative expressions, or machine translation. Discourse analysis turns out to be not so critical for these applications. However, applications such as automatic summarization (Marcu 2000) or question-answering do require an intensive discourse analysis level, as shown in (Jin et al. 1987).

Dedicated to sentence processing, let us note the GATE platform (http://gate.ac.uk/) which is widely used, and the Linguastream (http://www.linguastream.org) system which is based on a component architecture, making this system really flexible. Besides some specific features to deal with simple aspects of discourse processing, none of these platforms allow the specification of rules for an extensive discourse analysis nor the introduction of reasoning aspects, which is however essential to introduce pragmatic considerations into discourse processing. GATE is used e.g. for semantic annotation, corpus construction, knowledge acquisition and information extraction, summarization, and investigations around the semantic web. It also includes research on audio, vision and language connections. Linguastream has components that mainly deal with part of speech and syntactic analysis. It also handles several types of semantic data with a convenient modular approach. It is widely used for corpus analysis. In a different context, the GETARUNS system (http://project.cgm.unive.it/getaruns.html), based on the LFG grammar approach, has some capabilities to process simple forms of discourse structures and can realize some forms of argumentation analysis. Finally, (Marcu 2000) developed a discourse analyzer for the purpose of automatic summarization. This system is based on the RST assumptions, which are not always realistic in a number of contexts, as developed in the section below.

Within a very different perspective, and inspired by sentence syntax, two approaches based on Tree Adjoining Grammars (TAGs) (Gardent 1997) and (Webber et al. 2004), extend the formalism of TAGs to the processing of discourse structures via tree anchoring mechanisms. The approach remains essentially lexically based. It is aimed at connecting propositions related by various discourse connectors or at relating text spans which are in a referential relation.

1.1. Some Linguistic Considerations

Most works dedicated to discourse analysis have to deal with the triad: discourse structure identification, delimitation of its textual structure (boundaries of the discourse unit, sometimes called elementary discourse unit or EDU) and discourse structure binding to form larger structures representing the articulations of a whole text. By structure identification, we mean identifying a kernel or a satellite of e.g. a rhetorical relation, such as an illustration, an illustrated expression, an elaboration, or the elaborated expression, a conditional expression, a goal expression, etc. Discourse structures are realized by textual structures which need to be accurately delimited. These are then not isolated: they must be bound to other structures, based on the kernel-satellite or kernel-kernel principle.

<TextCoop> and Dislog are based on the following features and constraints:

- ***discourse structure identification***: identifying a basic discourse structure requires a comprehensive specification of the lexical, syntactic, morphological, punctuation and possibly typographic marks (Luc et al 1999), (Longacre 1982) that allow the identification of this discourse function. In general the recognition of satellite functions is easier than the recognition of their corresponding kernel(s) because they are more strongly marked. For example, it is quite straightforward to recognize an illustration (although we had to define 20 generic rules that describe this structure), but identifying the exact text span which is its kernel (i.e. what is illustrated) is much more ambiguous. Similarly, the support of an argument is marked much more explicitly than its conclusion.

- ***structure or text span delimitation***: most of the literature on discourse analysis dedicated to the delimitation of discourse units refers either to Elementary Discourse Units (EDUs) (Schauer 2006), or to the vague notion of text span. When identifying discourse structures in texts, finding their textual boundaries is very challenging. In addition, contrary to the assumptions of RST (e.g. (Grosz et al. 1986), (Marcu 2000)) several partly overlapping textual units can be involved in different discourse relations.

- ***binding discourse units***: the last challenge is, given a set of discourse structures, to identify relations between them. For example, relating an illustration and the illustrated element, which are not necessarily adjacent. Similarly, argument conclusions and supports are difficult to relate. These may not necessarily be contiguous: independent discourse functions may be inserted between them. We also observed a number of one-to-many relations (e.g. various reformulations of a given element), besides the standard one-to-one relations. Similarly, an argument conclusion may have several supports, possibly with different orientations. As a consequence, the principle of textual contiguity cannot be applied systematically. For that purpose, we have developed a principle called selective binding, which is also found in formal syntax to deal with long-distance dependencies or movement theory.

1.2. Some Foundational Principles of Dislog

In the Dislog approach, we consider:

- ***productive principles***, which have a high level of abstraction, which are linguistically sound, but which may be too powerful,

- ***restrictive principles***, which limit the power of the productive principles on the basis of well-formedness constraints.

Another foundational feature is an integrated view of marks used to identify discourse functions, merging lexical elements with morphological functions, typography and punctuation, syntactic constructs, semantic features and inferential patterns that capture various forms of knowledge (domain, lexical, textual). <TextCoop> is the first platform that offers this view within a logic-based approach.

If machine learning is a possible approach for sentence processing, where interesting results have emerged, it seems not to be so successful for discourse analysis (e.g. (Carlson et al. 2001)). This is due to two main factors: (1) the difficulty to annotate discourse functions in texts (Saaba et al 2008) characterized by the high level of disagreement between annotators and (2) the large non-determinism encountered when processing discourse structures where linguistic marks are often immerged in long spans of text of no or little interest. For these reasons, we adopted a rule-based approach. Rules may be hand coded, based on corpus analysis using bootstrapping tools, or may emerge from a supervised learning method.

Dislog rules basically implement productive principles. Rules are composed of three main parts:

- ***a discourse function identification structure***, which basically has the form of a rule or a pattern,

- ***a set of calls to inferential forms*** using various types of knowledge. These forms are part of the identification structure, they may contribute to solving ambiguities, they may also be involved in the computation of the resulting representation or they may express semantic or pragmatic restrictions. This is developed below on page 85,

- ***a structure that represents the result*** of the analysis: it can be the same text associated with simple XML structures, or any other structure such as a graph or a dependency structure. More complex representations, e.g. based on primitives, can be computed using a rich semantic lexicon. This is of much interest for an analysis oriented towards a conceptual analysis of discourse.

Besides rules, Dislog allows the specification of a number of ***restrictive principles***, expressed as active constraints, e.g. dominance, precedence, exclusion, etc. as shall be seen below. The restrictions introduced by these principles are checked throughout the whole parsing process.

2.Structure of Dislog Rules

Let us now introduce in more depth the structure of Dislog rules. Dislog follows the principles of logic-based grammars as implemented three decades ago in a series of formalisms, among which, most notably: Metamorphosis Grammars (Colmerauer 1978), Definite Clause Grammars (Pereira and Warren 1980) and Extraposition Grammars (Pereira 1981). These formalisms were all designed for sentence parsing with an implementation in Prolog via a meta-interpreter or a direct translation into Prolog. Illustrations are given in (Saint-Dizier 1994). The last two formalisms include a simple device to deal with long distance dependencies. Various processing strategies have been investigated in particular bottom-up parsing, parallel parsing, constraint-based parsing and an implementation of the Earley algorithm that merges bottom-up analysis with top-down predictions. These systems have been used in applications, with reasonable efficiency and a real flexibility to updates, as reported in e.g. (Gazdar et al. 1989).

Dislog adapts and extends these grammar formalisms to discourse processing, it also extends the regular expression format which is often used as a basis in language processing tools. The rule system of Dislog is viewed as a set of productive principles representing the different language forms taken by discourse structures.

A rule in Dislog has the following general form, which is quite close to Definite Clause Grammars and Metamorphosis Grammars from a syntactic and semantic point of view:

L(Representation) --> R, {P}.

where:

- L is a non-terminal symbol.
- Representation is the representation resulting from the analysis, it is in general an XML structure with attributes that annotate the original text. It can also be a partial dependency structure or a more formal representation.
- R is a sequence of symbols as described below, and
- P is a set of predicates and functions implemented in Prolog that realize the various computations and controls, and that allow the inclusion of inference and knowledge into rules. These are included between curly brackets as in logic grammars to differentiate them from grammar symbols.

R is a finite sequence of the following elements:

- ***terminal symbols*** that represent words, expressions, punctuations, various existing html or XML tags. They are included between square brackets in the rules,
- ***preterminal symbols***: are symbols which are derived directly into terminal elements. These are used to capture various forms of generalizations, facilitating rule authoring and update. They should be preferred to terminal elements when generalizations over terminal elements are possible. Symbols can be associated with a type feature structure that encodes a variety of aspects from morphology to semantics,
- ***non-terminal symbols***, which can also be associated with type feature structures. These symbols refer to "local grammars", i.e. grammars that encode specific syntactic constructions such as temporal expressions or domain specific constructs. Non-terminal symbols do not include discourse structure symbols: Dislog rules cannot call each other, this feature is dealt with by the selective binding principle, which includes additional controls. A rule in Dislog encodes the recognition of a discourse function taken in isolation.
- ***optionality and iteration marks*** over non-terminal and preterminal symbols, as in regular expressions,
- ***gaps***, which are symbols that stand for a finite sequence of words of no present interest for the rule which must be skipped. A gap can appear only between terminal, preterminal or non-terminal symbols. Dislog offers the possibility to specify in a gap a list of elements which must not be skipped: when such an element is found before the termination of the gap, then the gap fails. The length of the skipped string can also be controlled. A ***skip*** predicate is also included in the language: it is close to a gap and simply allows the system to skip a maximum number of words given as a parameter.
- ***a few meta-predicates*** to facilitate rule authoring.

Symbols in a rule may have any number of arguments. However, in our current version, to facilitate the implementation of the meta-interpreter and to improve its efficiency, the recommended form is:

identifier(Representation, Typed feature structure).

where Representation is the symbol's representation. In Prolog format, a difference list (E,S) is added at the end of the symbol:

identifier(R, TFS, E, S)

The typed feature structure (TFS) can be an ordered list of features, in Prolog style, or a list of attribute-value pairs. Examples are developed in the next chapter.

Similarly to DCGs and to Prolog clauses, it is possible and often necessary to have several rules to fully describe the different realizations of a given discourse function. They all have the same identifier Ident, as it is the case e.g. for the rules that describe NPs or PPs. A set of rules with the same identifier is called a ***cluster of rules***. Rule in a cluster are executed sequentially, in their reading order, from the first to the last one, by the <TextCoop> engine. Then, clusters are called in the order they are given in a ***cascade***. This is explained below.

As an illustration, let us consider a generic rule that describes conditional expressions:

Condition(R) -->

conn(cond,_), gap(G), ponct(comma).

with:

conn(cond,_) --> if.

For example, in the following sentences, the underlined structures are identified as conditions:

If all of the sources seem to be written by the same person or group of people*, you must again seriously consider the validity of the topic.*

If you put too many different themes into one body paragraph*, then the essay becomes confusing.*

For essay conclusions, don't be afraid to be short and sweet ***if you feel that the argument's been well-made****.*

The gap G covers the entire conditional statement between the mark *if* and the comma. The argument R in the above rule contains a representation of the discourse structure, for example, in XML:

<condition> .... </condition>....

<condition> If all of the sources seem to be written by the same person or group of people, </condition> you must again seriously consider the validity of the topic.

2.1.Dislog Advanced Features

In this section, we describe the features offered by the Dislog language that complement the grammar rule system. These mostly play the role of restrictive principles. At the moment we have defined three sets of devices: selective binding rules to link discourse units identified by the rule system, correction rules to revise incorrect representations e.g. erroneously placed tags made by previous rules, and concurrency statements that allow a correct management of clusters of rules. Concurrency statements are closely related to the cascade system. They are constrained by the notion of bounding node, which

delimits the text portion in which discourse units can be bound. Similarities with sentence formal syntax are outlined when appropriate, however, phenomena at discourse level are substantially different.

Selective Binding Rules

Selective binding rules are the means offered by Dislog to construct hierarchical discourse structures from elementary ones (discourse functions), identified by the rule system. Selective binding rules allow the system to link two or more already identified discourse functions. The objective is e.g. to bind a kernel with a satellite (e.g. an argument conclusion with its support) or with another kernel (e.g. for the concession or parallel relations).

Related to this latter situation is the binding of two argument conclusions, which e.g. share a similar support, as in:

Do not use butter to cook vegetables because it contains too much cholesterol, similarly, avoid palm oil.

The *similarly* linguistic mark binds two argument conclusions related to the same topic. This mark introduces a kind of ellipsis. Selective binding rules can be used for other purposes than implementing rhetorical relations. These can be used more generally to bind any kind of structure in application domains. For example, in procedural discourse, they can be used to link a title with the set of instructions, prerequisites and warnings that realize the goal expressed by this title.

From a syntactic point of view, selective binding rules are expressed using the Dislog language formalism. Different situations occur that make binding rules more complex than any system of rules used for sentence processing, in particular:

- discourse structures may be embedded to a high degree, with partial overlaps,

- they may be chained (a satellite is a kernel for another relation),

- kernels and related satellites may be non-adjacent,

- kernels may be linked to several satellites of different types,

- some satellites may be embedded into their kernel.

Selective binding rules allow the binding of:

- two adjacent structures, in general a kernel and a satellite, or another kernel. A standard case is an argument conclusion followed by its support.

- more than two adjacent structures, a kernel and several satellites. For example, it is quite common to have an argument conclusion associated with several supports, possibly with different orientations:

  The Maoists will win the elections because they have a large audience and because they threat Terrai tribe leaders.

-two or more non-adjacent structures, which may be separated by various elements (e.g. causes and consequences, conclusion and supports may be separated by various elements). This is a frequent situation in everyday language, where previous items are referred to via pronominal references or via various kinds of marks (e.g. *coming back to*):

  Avoid seeding by high winds. Avoid also frost periods. Besides the fact that the wind will disperse most of your seeds, your vegetables will not grow where you expect them to be.

However limits must be imposed on the "textual distance" between units. In discourse, such a constraint is not related to well-formedness constraints as it is in sentence syntax, but it captures the fact that units which are very distant (e.g. several paragraphs) are very difficult to conceptually relate. One of the reasons is that focus or even topic shifts are frequent over paragraphs or sections and memorizing the topic chain is somewhat difficult.

In terms of representation, the two first cases above can be dealt with using a standard XML notation where the different structures are embedded into a parent XML structure that represents the whole structure. This is realized via the use of logic variables in logic programming which offer a very powerful declarative approach to structure building. The latter case requires a different kind of representation technique. We adopt a notation similar to the neo-Davidsonian notation used for events in sentence logical representations (Davidson 1963). An ID, which can be interpreted as a discourse event, is associated with each tagged discourse function.

A selective binding rule for two adjacent structures can then be stated as follows:

argument(R) -->

[<conclusion>], gap(G1), [</conclusion>],

connector(C,[type:cause]), [<support>], gap(G2), [</support>].

where the first gap covers the conclusion (textual unit G1) and the second one covers the support (textual unit G2), the connector is defined by C, with the constraint that it is of type cause.

To limit the textual distance between argument units, we introduce the notion of ***bounding node***, which is also a notion used in sentence formal syntax to restrict the way long-distance dependencies can be established (Lasnik et al. 1988). Bounding nodes are also defined in terms of barriers in Generative syntax (Chomsky 1986). In our case, the constraint is that a gap must not go over a bounding node. This allows the system to restrict the distance between the constituents which are bound.

For example, we consider that an elaboration and the elaborated element or an argument conclusion and its support must be in the same paragraph, therefore, the node "paragraph" is a bounding node. This declaration is taken into account by the <TextCoop> engine in a transparent way, and interpreted as an active constraint which must be valid throughout the whole parsing process.

The situation is however more complex than in sentence syntax. Indeed, bounding nodes in discourse depend on the structure being processed. For example, in the case of procedural discourse, a warning can be bound in general to one or more instructions which are in the same sub-goal structure. Therefore, the bounding node will be the sub-goal node, which may be much larger than a paragraph.

Bounding nodes are declared as follows in Dislog:

boundingnode(Node name, type of discourse structure), as in:

boundingNode(paragraph, argument).

Rule Concurrency Management

The current <TextCoop> engine is close to the Prolog execution schema. However, to properly manage rule execution, the properties of discourse structures and the way they are usually organized, we introduce additional constraints, which are, for most of them, borrowed from sentence syntax.

Within a cluster of rules, the execution order is the rule reading order, from the first to the last one. Then, elementary discourse functions are first identified and then bound to others to form larger units, via selective binding rules. Following the principle that a text unit has one and only one discourse function (but may be bound to several other structures via several rhetorical relations) and because rules

can be ambiguous from one cluster to another, the order in which rule clusters are executed is a very crucial parameter.

To handle this problem, Dislog requires that rule clusters are executed in a precise, predefined order, implemented in a ***cascade of clusters of rules***. This notion was introduced by (Stabler 1992) with the notion of layers and folding-unfolding mechanisms in an implementation of Generative Syntax theory in Logic Programming.

For example if, in a procedure, we want first titles, then prerequisites and then instructions to be identified, the following constraint must be specified:

title < prerequisite < instruction.

Since titles have almost the same structure than instructions, but with additional features (bold font, html specific tags, etc.), this prevents titles from being erroneously identified as instructions.

Similarly, it is much preferable to process argument supports, which are easier to identify, before argument conclusions. Processing advice before warnings also limits risks of ambiguities:

advice-support < advice-conclusion < warning-support

< warning-conclusion.

In the current version of the search engine, there is no backtracking between clusters. Next, when there is no a priori complete order between clusters, those not mentioned in the cascade are executed at the end of the process.

In relation with this notion of cascade, it is possible to declare ***closed zones***, e.g.:

closed_zone([title]).

indicates that the textual span recognized as a title must not be considered again to recognize other functions within or over it (via a gap). In this example, there will be no further attempt to recognize any discourse structure within a title.

Structural constraints

Let us now consider basic structural principles, which are very common in language syntax. This allows us to contrast the notion of consistuency with the notion of relation in discourse. Consistuency is basically a part-of relation applied to language structures (nouns are parts of NPs) while discourse is basically relational.

Let us introduce here dominance and precedence constraints. Discourse abound in various types of constraints, which may be domain, style or structure dependent (Barenfanger et al. 2006). Dislog can accommodate the specification of a number of such structural constraints.

Dominance constraints are stated as follows:

dom(instruction, condition).

This constraint states that a conditional expression is always dominated by an instruction. This means that a condition must always be part of an instruction (it is a constituent of that instruction), it cannot stand in a discourse relation with an instruction. In that case, there is no discourse link between a condition and an instruction, the implicit structure being consistuency: a condition is a constituent, or a part of, an instruction.

Similarly, ***non-dominance constraints*** can be stated to ensure that two discourse functions appear in different branches of a discourse representation, e.g.:

not_dom(instruction, warning).

states that an instruction cannot dominate a warning. However, a warning may be associated with an instruction via a rhetorical relation if its scope is that instruction. This is implemented by a selective binding rule.

Finally, ***precedence constraints*** may be introduced. We only consider here the case of immediate linear precedence, for example:

prec(elaborated, elaboration).

This constraints indicates that an elaboration must immediately follow what is elaborated. This is a useful constraint for the cases where a nucleus must necessarily precede its satellite: it contributes to the efficiency of the selective binding mechanism and resolves some recognition ambiguities.

Introducing Reasoning Aspects into Discourse Analysis

Discourse relation identification often require some forms of knowledge and reasoning. This is in particular the case to resolve ambiguities in relation identification when there are several candidates or to clearly identify the text span at stake. While some situations are extremely difficult to resolve, others can be processed e.g. via lexical inference or reasoning over ontological knowledge. This problem is very vast and largely open, with exploratory studies e.g. reported in (Van Dijk 1980), (Kintsch 1988), and more recently some debates reported in:

http://www.discourses.org/UnpublishedArticles/SpecDis\&Know.htm.

Dislog allows the introduction of reasoning, and the <TextCoop> platform make it possible to integrate knowledge and functions to access knowledge.

Within our perspective, let us give a simple motivational example. The utterance (found in our corpus):

... red fruit tart (strawberries, raspberries) are made ...

contains a structure*: (strawberries, raspberries)* which is ambiguous in terms of discourse functions: it can be an elaboration or an illustration, furthermore the identification of its kernel is ambiguous:

red fruit tart, red fruit ?

A straightforward access to an ontology of fruits indicates that those berries are red fruits, therefore:

- the unit (*strawberries, raspberries)* is interpreted as an illustration, since no new information is given (otherwise it would have been an elaboration)

- its kernel is the *red fruit* unit only.

Note that these two constituents, which must be bound, are not adjacent.

Very informally, the binding rule that binds an illustration with the illustrated text span can be defined as follows, assuming that these are all NPs, with well-identified semantic types:

Illustrate -->

<illustrated>, NP(Type), </illustrated>, gap(G),

<illustration> NP1(Type1), NP2(Type2), </illustration>,

[subsume(Type,Type1), subsume(Type, Type2)]

The subsume control makes sure that the type of the illustrated element (Type) is more general than the type of the elements in the illustration (Type1, Type2).

The relation between an argument conclusion and its support may not necessarily be straightforward to identify and may involve various types of domain and common-sense knowledge:

Do not park your car at night near this bar: it may cost you fortunes.

Women's living standards have progressed in Nepal: we now see long lines of young girls early morning with their school bags. (Nepali Times).

In this latter example, *school bag* means going to school, then *school* means education, which in turn means better living standards.

The <TextCoop> Architecture and Environment

The architecture of <TextCoop> is rather standard. It is organized around the following modules:

- a module of rules or patterns written in Dislog. This module includes the various types of rules recognizing basic discourse functions and the binding rules,
- one or more modules dedicated to lexical resources. It is advised to have different modules for general purpose lexical data and lexical data specific to certain discourse constructs. This allows an easier management of lexical data and a better update and re-use over various applications,
- a module dedicated to morphological processing
- a module for the management of the system: this module includes the management of constraints and the cascade specification.
- the <TextCoop> engine, developed below, which is associated with a few utilities (Prolog basic functions, input/output management, character encoding management, etc.).

These elements are given in the freeware archive which can be obtained upon request to the author.

In a second stage, we defined a working environment for <TextCoop> in order to help the grammar writer to develop applications. This environment is in a very early stage of development: many more experiments are needed before reaching a stable analysis of the needs. Accessing already defined and formatted resources is of much interest for authors.

The following sets of resources are available for French and English:

- lists of connectors, organized by general types: temporal, causal, concession, etc. (Miltasaki et al. 2004) developed an original learning method to classify connectors and related marks,
- list of specific terms which are typical of discourse functions, e.g.: terms specific of illustration, summarization, reformulation, etc.
- lists of verbs organized by semantic classes, close to those found in WordNet, which have been adapted or refined for discourse analysis, with a focus e.g. on propositional attitude verbs, report verbs, (Wierzbicka 1987),
- list of terms with positive or negative polarity, essentially adjectives, but also some nouns and verbs. This is useful in particular to evaluate the strength of arguments,
- local grammars for e.g.: temporal expressions, expression of quantity, etc.,

- some already defined modules of discourse function rules to recognize general purpose discourse functions such as illustration, definition, reformulation, goal and condition.

- some predefined functions and predicates to access knowledge and control features (e.g. subsumption),

- morphosyntactic tagging functions,

- some basic utilities for integrating knowledge (e.g. ontologies) into the environment.

This environment is compatible with sentence parsers which can operate on the text independently of the tags, or within tag fields. Some of the elements of this environment are illustrated in the next chapter.

The <TextCoop> Engine

Let us now introduce the way the <TextCoop> engine runs. The engine and its environment are implemented in SWI Prolog, using the standard Prolog syntax without referring to any libraries to guarantee readability, ease of update and portability. It is therefore a stand-alone application.

A major principle is that the declarative character of constraints and structure building is preserved in the system. The engine, implemented in Prolog, interprets them at the appropriate control points.

The engine follows the cascade specification for the execution of rule clusters. Within each cluster, rules are activated in their reading order, one after the other. Backtracking manages rule failures. If a rule in a rule cluster succeeds on a given text span, then the other possibilities for that cluster are not considered (but rules of other clusters may be considered in a later stage of the cascade).

A priori, the text is processed via a left to right strategy. However, <TextCoop> offers a right to left strategy for rules where the most relevant marks are to the right of the rule, in order to limit backtracking. For the two types of readings, the system is tuned to recognize the smallest text span that satisfies the rule structure.

The engine can work on different textual units: sentences, paragraphs, sections, etc. depending on the kind of structure or phenomenon to recognize (some have a very large scope such as the "frame" relation that constrains a whole paragraph or even more, while others such as the goal of an instruction or an illustration usually operate over a single sentence. "Title" relations also range over a large text fragment).

Relevant units can be specified in the cascade for each cluster of rules. The system is more efficient and generates less ambiguities with smaller units. It processes raw text, html or XML texts. A priori, the initial XML structure of the processed document is preserved.

The code associated with the <TextCoop> engine is rather small. We present here its main features. It is based on the notion of meta-interpreter in Prolog. In a meta-interpreter, different processing strategies can be developed besides the top-down, left-to-right strategy offered in Prolog.

Before developing a few details concerning the <TextCoop> engine, let us consider, as an introduction, a meta-interpreter of Definite Clause Grammar rules (DCGs). Most Prolog versions automatically translate DCGs into Prolog. However, designing a meta-interpreter can be useful e.g. to change the processing strategy.

Let us assume that a DCG rule is represented as:

Axiom --> Body.

and a lexical entry as:

```
Axiom --> [Word].
```

Axiom and Body may have any kind of structure (term with arguments, feature structure, etc.). The clause parse that enables the processing of the grammar rule format given above is defined as follows. The two first arguments respectively manage the input and output list of words (corresponding to a sentence) to process:

```
:-op(1100,xfy,-->).    % definition of the priority of the operator -->


parse(X, X1, Axiom) -->
  (Axiom --> Body),                 % if unification with Axiom
  parse(X, X1, Body), !.            %  succeeds then process the Body
parse([M|X], X, [M]) :- !.        % processes lexical entry
parse(X, X1, (C1, C2)) :-         % Body processing from left to right
  parse(X, X2, C1),
  parse(X2, X1, C2).                % Body = (C1, (C2))
```

To implement a right-to-left processing strategy, useful for example for languages where the head is to the right, the only modification is to change the order C1 and C2 are processed as follows:

```
parse(X, X1, (C1, C2)) :-          % Body processing from right to left
  parse(X2, X1, C2),
  parse(X, X2, C1).
```

Let us now consider the main features of the <TextCoop> engine. More details on the rule format and on how to launch the system are given in the next chapter. The general form of a rule is:

```
forme(Name,
        Input string,
        Output string,
        [right hand side symbols in the rule],
        Reasoning terms,
        Result).
```

The main rule of the meta-interpreter is then:

```
corr(Forme,E,S,C):-  %recognizes the rule "Forme" in the sequence E-S
                        % result is stored in C (annotated text in general,
  forme(Forme,E,S, [A1| F1], Re, C),
                        % call to a rule according to the cascade
```

```
typeOK(A1,E,S1),
        % checks whether the first symbol in the rule, A1, is found
suiteOK(S1,S,F1),                % next steps till the end of the rule
dominanceFinOK(Forme,S),     % dominance controls
        % in case of failure of the above terms, there is backtracking
executer(Re),                    % processing of the reasoning aspects.
non_dominance(Forme,C).      % non dominance control.
```

2. PROGRAMMING IN DISLOG

In this chapter, the use of <TextCoop> and Dislog is explained in detail. This chapter has essentially a practical purpose, it is a kind of user manual. It is of interest to the reader who wants to develop applications, otherwise it can be skipped. The TextCoop archive can be obtained from the author (stdizier@irit.fr), it is freely available for research and training purposes. In this chapter, we first describe how to install TextCoop from the archive. Next, we develop principles and a simple method to write Dislog rules so that the reader can start writing simple rules. Detailed comments are given so that the reader can define himself his own rules or write variants of the examples which are given.

1.Using Dislog and <TextCoop>

Let us first show how <TextCoop> and Dislog can be installed from the archive and how to run a Dislog programme.

Installation

<TextCoop> is a meta-interpreter which is implemented in Prolog. To make the system portable, only the kernel syntax of Prolog is used, therefore, most versions of Prolog using the Edinburgh syntax should allow <TextCoop> to run adequately (possibly a few predefined predicates will need to be revised if a Prolog implementation such as Sicstus is used). We recommend to use SWI Prolog, which is free and runs efficiently on several platforms. Note that SWI Prolog seems to run faster on a Linux environment than on MS Windows.

The only thing you have to do is to unzip the <TextCoop> archive into a directory of your choice. A priori it is simpler to keep all the files in a single directory. However, the text files you analyse can be stored in another directory. Basically, the archive contains two directories, one for the French version and the other one for the English version (files end by Fr or Eng depending on the language).

Each directory contains the following files:

- the engine: textcoopV4.pl

- a specific file for user declarations and parameters: decl.pl

- a set of basic functions that you do not need to update or even look at: fonctions.pl

- a set of lexicons that contain various types of data: lexiqueBase.pl, lexSpecialise.pl, lexiqueIllustr.pl. You can obviously construct several additional lexicons, but you must add the call to their compilation at the beginning of the textcoopV4.pl file. The French version also contains a list of categorized verbs (eeeaccents.pl). It must be noted that all the rules corresponding to a specific predicate must be defined in the same file

- a file with rules or patterns: rules.pl

- a toy file with "local" grammar samples written in DCG format: gram.pl

- a file for the input-output operations: es.pl and another one for reading files of various text formats and transforming them into Prolog: lire.pl

- a few files to run and test the system: demo.txt, after processing a text file, the system produces two output files: demo-out.html (tags, no colours for further processing) and demo-c-out.html (same thing but with colours and spaces to facilitate reading). However, note that we have not developed at this stage any user interface

- additional files can be added, for example to include knowledge or specific reasoning procedures. Nothing is included at this level in this beta version.

Starting

There are several ways to read and modify your files and to call Prolog. Emacs and similar editors are particularly well-adapted. We recommend the use of Editplus V3 for those who do not have any preference. Prolog can be launched directly from the editor and the code, or selected parts of it, can be easily re-interpreted when modifications are made. It is important to keep in mind that text files encoded in the UTF8 or UTF16 formats may be problematic for Prolog, which basically takes as input texts under an ISO format. Character encoding may be tuned in some environments, such as in Linux. It is recommended that the texts you want to analyse are all in .txt format.

To start the system, you must launch Prolog from your environment, e.g. from Editplus. You must then "consult" your file(s). Since the file textcoopV4.pl contains consult orders for all the other files, you just need to consult it, via the menu of the Prolog window, or, in the window by typing:

['textcoopV4.pl'].

care about ERROR messages, but you can ignore warnings.

Then to launch TextCoop, type the main call:

annotF.

then you are required to enter your file name, between quotes, ended by a dot, as usual in Prolog:

?- 'demo.txt'.

The system processes the text and you will then see the display on your screen of a large number of intermediate files which are created and re-used. Each cycle of these intermediate files corresponds to the execution of a cluster of rules given in the rule cascade. Results are stored in two files: demo-out.html (html tags, no colour) and demo-c-out.html (same thing but with colours and spaces to facilitate reading).

The file es.pl contains a few other input/output calls that you may wish to explore. You can also change the display colours in this file, or add or withdraw the display of some tags. The contents of the tags are produced by discourse analysis rules, described in the "Representation" argument.

2.Writing Rules in Dislog

Rules are stored in this archive in the rules.pl file. These rules have been produced from a manual analysis of linguistic phenomena, they could have been the result of a statistical analysis. A priori, the Dislog language is flexible enough to accept a large variety of forms. Elements of a simple methodology for designing rules are given below in the chapter to facilitate rule authoring.

The first thing to do if you want to analyse a discourse structure is to define and characterize the phenomena to treat via a linguistic analysis. This means a corpus analysis, then abstracting over corpus observations using grammar symbols and generalizing these abstractions at an appropriate linguistic

level. The next stages are to develop the lexical resources (cues typical of the structure being investigated and other useful resources) and write rules, following the syntax given above. These rules must then be encoded in Dislog, where a few more arguments and formatting issues must be considered, as explained in a section at the end of this chapter.

Let us first take an example on how to write Dislog rules. The rule that describes a purpose satellite can be written in an external format as follows:

```
purpose -->
  connector(goal),
  verb([action,impinf]),
  gap(G),
  punctuation.
```

e.g, where the purpose clause is underlined:

***To write a good essay on English literature**, you need to do five things, first start [...].*

This rule says that a purpose satellite is composed of a connector of type goal, followed by an action verb in the imperative or infinitive form followed by a gap. The structure ends by a punctuation mark. Labels such as goal, impinf or action are defined by the rule author, they are not imposed by the system. These tags may be encoded in a variety of ways: as lists (as in this example) or as a feature structure. It is preferable to have all the features stored in a single argument.

The general form of a rule coded in Dislog is:

```
forme(LHS, E, S, RuleBody, Constraints, Result).
```

where:

- **LHS** is the symbol on the left-hand side of the rule. It is the name of the cluster of rules representing the various structures corresponding to a phenomenon (e.g. purpose satellites). It is used in various constraints and in the cascade to refer to this cluster.
- **E and S** are respectively the input and output strings representing the sentence or text to process, similarly to the DCGs difference list notation. The informal meaning is that between E and S there is a purpose clause. E and S are lists of words coded in Prolog.
- **RuleBody** is the right-hand part of the rule, it is described below,
- **Constraints** is a list that contains a variety of constraints to check. They may also be calls to knowledge and reasoning procedures which are written in Prolog and automatically executed at the end of the rule. An empty list means that there is no constraints. It is always evaluated to true.
- **Result** denotes the result which includes the string of words of the input structure with tags included at the appropriate places. Tags in rules may include variables.

The rule body is encoded as follows. Each grammar symbol has four arguments (this is a choice which can be modified, but seems optimal and easy to use):

```
name(String,Feature,E,S).
```

where:

- **String** denotes the String which is reproduced in the result in conjunction with tags. In general it is the string that has been read for that symbol (e.g. the difference between E and S), but it can be any other form (e.g. a normalized form, a reordered string, etc.).

- **Feature** is the argument that contains information about the symbol, encoded e.g. as a list of values or as a typed feature structure. The format is unconstrained, but the rule writer must manage it.

- **E** and **S** are respectively the input and output lists of words, as above, for the analysis of this particular symbol. E and S form what is called the difference list in logic-grammars.

Gap symbols have a different format:

gap(NotSkipped,Stop,E,S,Skipped).

where:

- **NotSkipped** is a list of symbols which must not be skipped before the gap stops. If it finds such a symbol, then the gap fails. So far, this list is limited to a single symbol for efficiency reasons. We have not found so many cases where multiple restrictions are needed. If really needed, these must be coded in the "gap" clause.

- **Stop** is a list: [Symbol, Restrictions] that describes where the gap must stop: it must stop immediately before it finds a symbol Symbol with the restrictions Restrictions. In general this is the explicit symbol that follows the gap in the rule, but this is not compulsory.

- **E** and **S** are as above,

- **Skipped** is the difference between E and S, namely the sequence of words that have been skipped by the gap.

It must be noted that a gap must only appear in a rule between two explicit symbols. While processing a rule, if a gap reaches the end of a sentence (or a predefined ending mark such as a dot) without having found the symbol that follows in the rule, then it fails and the rule also fails.

The symbol skip is slightly different from the gap symbol. It allows the parser to skip a maximum number of N words given as parameter. It has the same structure as a gap, except that the second argument is an integer (in general small) that indicates the maximum number of words to skip:

skip(NotSkipped, Number, E, S, Skipped).

The symbol in the rule that immediately follows a skip symbol defines the termination of the skip. It is not specified in the skip symbol. If the maximum number of words to be skipped is not followed by the expected symbol, then the skip fails.

The rule given above for a purpose satellite then translates as follows in Dislog:

```
forme(purpose-eng,
E, S,
[ connector(CONN,goal,E,E1),
  verb(V,[action,impinf],E1,E2),
  gap([],[ponct,_],E2,E3,Saute1),
  ponct(Ponct,_,E3,S)],
```

```
 [],
['<purpose>', CONN, V, Saute1, '</purpose>', Ponct ]).}
```

This rule is represented as a Prolog fact so that the TextCoop meta-interpreter can read it and then process it. The reader can note the sequences of input-output variables E-E1-E2-E3-S used as in DCGs. The last argument encodes the result, for example the way the original sentence is tagged. Tags may be inserted at any place; they may contain variables elaborated in the inference (also called Constraints) field. In fact, a priori any form of representation can be produced in this field. The above example shows that the result is produced in a very declarative and readable way.

Symbols in a rule can be marked as optional or can appear several times. This is respectively encoded using the operators opt and plus applied to grammar symbols:

```
forme(purpose-eng,
E, S,
[ connector(CONN,goal,E,E1),
 opt(verb(V,[action,impinf],E1,E2)),
 gap([],[ponct,_],E2,E3,Saute1),
 ponct(Ponct,_,E3,S)],
 [],
['<purpose>', CONN, V, Saute1, '</purpose>', Ponc ]).
```

In this example, the verb is indicated as optional: if it is not found, then the gap starts immediately after the connector. If there is no verb, the variable V in the result field is not instantiated and does not produce any result in the last argument (variables are ignored in the representation sequence since their contents is empty).

In the example below, a sequence of auxiliaries is allowed between the connector and the verb:

```
forme(purpose-eng,
E, S,
[  connector(CONN,goal,E,E1),
   plus(aux(Aux,_,E1,E2),
   verb(V,[action,_],E2,E3),
   gap([],[ponct,_],E3,E4,Saute1),
   ponct(Ponct,_,E4,S)],
 [],
['<purpose>', CONN, Aux, V, Saute1, '</purpose>', Ponc ]).
```

The operators plus and opt are defined in the kernel of <TextCoop> and are implemented in the textcoopV4.pl file. These operators can be modified and variants can be implemented without affecting

the remainder of the system. For example a predicate plus2 that would require at least two instances of the symbol it includes could easily be added to the <TextCoop> engine.

Finally, as the reader can note it, the transformation of rules written in the external format into the Dislog internal format obeys very regular and strict principles. Similarly to e.g. DCGs, it is possible, but however more complex than for DCGs, to write a few lines of code in Prolog that produce such an internal form from the external one.

Writing Context-Dependent Rules

A closer look at the Dislog rule formalism shows that it is possible to use this formalism to implement context-dependent rules. In fact, the left-hand side symbol, the cluster name, can be viewed as an identifier, and the power of the rule formalism can be shifted to the pair left-hand part (RuleBody) and representation or right-hand part or the rule (Result). The left-hand part is indeed often the input form to identify and its relation with the Result introduces a large diversity of treatments.

We already advocated the case of binding rules, which are clearly type 1 if not type 0 rules. It is possible in Dislog to develop any other kind of rules, e.g. to realize structure transformation with some form of context sensitivity.

To illustrate this point, let us consider again the example that deals with the binding of a warning conclusion with its support. The following rule:

```
Warning -->

<warning-concl>,gap(G1),</warning-concl>, gap(G2),

<warning-supp>, gap(G3),  </warning-supp>, gap(G4), eos.
```

binds a warning conclusion with a warning support. The result is a warning, represented by the following XML structure:

```
<warning>

<warning-concl>,G1,</warning-concl>, G2,

<warning-supp>, G3,  </warning-supp>, G4,

</warning>.
```

The same type of rule can be written to bind any kernel with its satellite, or more complex structures, e.g. a title (denoting an action to realize) with its prerequisites and instructions.

Parameters and Structure Declarations

In contrast with Prolog, but with the aim of improving efficiency, it is necessary in Dislog to declare a few elements, among which non-terminal symbols. This is realized in the decl.pl file. A number of standard symbols are already declared, but check that yours are declared; otherwise the rules that contain these symbols will fail.

First, in order to enable a proper variable binding, any symbol used in rules must be declared as follows:

```
tt(adv(Mot,Restr,E,S), E,S).

tt(adj(Mot,Restr,E,S), E,S).
```

```
tt(neg(Mot,Restr,E,S), E,S).

tt(np(Mot,Restr,E,S), E,S).

tt(det(Mot,Restr,E,S), E,S).
```

etc.

In this example, the symbols adv, adj, neg, np and det are declared. The co-occurrence of the symbols E and S allows us to bind the variables of the symbols (adv, adj, etc.) with the string of words to process in the meta-interpreter (the <TextCoop> engine).

Similarly, any symbol which can be optional must be declared by means of a piece of code. The code must be reproduced from the following example which encodes optionality for auxiliaries:

```
opt(aux(AUX,A,E1,E2)) :-
    aux(AUX,A,E1,E2), !.

opt(aux([],_,E,E)).
```

A similar declaration is necessary for multiple occurrences:

```
plus(adv(T,_,E1,S)) :-
  adv(T1,_,E1,E2), !,
  plus(adv(T2,_,E2,S)),
  conc(T,S,E1).
plus(adv(T,_,S,S)).
```

This portion of code must be duplicated for all relevant symbols.

Instead of writing these declarations, a higher-order encoding could have been implemented, but it does affect efficiency quite substantially. In a future version of the system (V5), an alternative solution will be found so that this constraint is no longer necessary. Some of these declarations could be automated from the grammar, but they may impose additional rule format constraints that rule authors may not want to have.

Constraints must also be declared (at least one instance of each type must be present in the code to avoid failures), a few examples are provided here:

```
exclut_unite(title).


termin(['<',condition,'>'],['<','/',condition,'>']).

termin(['<',purpose,'>'],['<','/',purpose,'>']).

termin(['<',circumstance,'>'],['<','/',circumstance,'>']).

termin(['<',restatement,'>'],['<','/',restatement,'>']).
```

```
dom(instr-eng,[but, condopt-eng, restatement]).


non_dom(instr-eng,[avt,cons]).
```

A few constraints are given in the decl.pl file as examples. These must be kept to avoid system failures. This file also contains the cascade declaration, as explained below.

Lexical Data

Lexical data is specified in the lexique.pl file. Lexical data can follow the standard categories and features of linguistic theories or be *ad hoc*, depending on situations. Lexical data is given in DCG format (Pereira et al. 1980) (Saint-Dizier 1994). In general, you have to design yourself your own lexicon with its features. In this first version, we simply provide a few examples to help rule authors. However, in addition to those available on the net, we are developing sets of lexical markers and other resources which are useful for discourse analysis. These will be made available in a coming version (V5).

Here are a few examples included into this version:

```
% pronouns

pro([we],_) --> [we].

pro([you],_) --> [you].


 % goal connectors

connector([in,order,to], goal) --> [in,order,to].

connector([in,order,that], goal) --> [in,order,that].

connector([so], goal) --> [so].

connector([so,as,to], goal) --> [so,as,to].


 % specific marks describing the beginning of a sentence

mdeb([debph],_) --> [debph].          % by-default mark internal to the system

mdeb(['<li>'],_) --> ['<',li,'>'].

mdeb([1],_) --> ['-',1].


 % condition

expr([if],cond) --> [if].


 % specific marks for reformulation (no features associated)
```

```
expr_reform([in,other,words],_) --> [in,other,words].

expr_reform([to,put,it,another,way],_) --> [to,put,it,another,way].

expr_reform([that,is,to,say],_) --> [that,is,to,say].


 % tag (= balise) represented as lexical data: this is useful for rules

 % which basically bind structures on the basis of already produced tags,

 % each element has a type specified in the second argument.

balise([<, instruction, >],instruction) --> [<, instruction, >].

balise([<, /, instruction, >],endinstruction) --> [<, /, instruction, >].


balise([<, goal, >],goal) --> [<, goal, >].

balise([<,/,goal, >],endgoal) --> [<, /,goal, >].
```

Other Types of Resources

This first version is relatively limited and does not contain so many additional tools and facilities. These will come in the next available version of the tool (version V5). However, version V4 contains the kernel necessary to implement the recognition of most discourse structures and to bind them adequately.

The file gram.pl of the archive contains a few grammar rules written in DCG style. These are compatible with the Dislog symbol formats. Indeed, it is possible to have non-terminal symbols in rules which are associated with a grammar in that module. This is useful for example to capture specific constructions which are better described by means of rules and not simply by lexical entries. Here is a very simple example for noun phrases (np):

```
np([A,B],_,E,S) :-
    det(A,_,E,S1),
    n(B,_,S1,S).
np([A],_,E,S) :-
   pro(A,_,E,S).
```

This short sample of a grammar for noun phrases can be directly called from Dislog rules. Note that the first argument of the np symbols contains the string of words which corresponds to the NP. This argument could include any other form, e.g. a normalized form or a tree. The second argument (features associated with the symbol, left empty here) argument must have the same structure as in the Dislog rules.

Input-Output Management

The management of the input-output files is realized in several files. The main file is es.pl which contains the main calls and dynamically produces names for output and intermediate files (to avoid conflicts between parses). This file contains procedures that produce two kinds of output displays: a file

for further processing which is a basic XML file and a similar file where structures get a colour for easier reading. This latter file can be read by most XML editors.

File names are created dynamically from the input file name: demo-out.html (tags, no colours) and demo-c-out.html (tags and colours) for the demo.txt file. If you have sufficient Prolog programming skills, you can modify this file, e.g. changing colours, if you need it.

It is important to note that, in this first version, structure processing is realized on a sentence basis. We have improved and parameterized this situation which is somewhat limited. It will shortly be possible to implement other units besides sentences such as paragraphs. Meanwhile, you can end the text portions you want to process as a single unit by a dot, and replace dots ending sentences in these portions by another symbol, e.g. the word "dot", which can be re-written later by a real dot in the output file.

Basically, input files must be plain text, possibly with XML marks. Word files cannot be processed. It must also be noted that Prolog has some difficulties with UTF8 encoded texts, therefore ISO encodings must be preferred.

The other files for input-output operations are internal to the system and should not be modified: lire.pl reads files under various formats and produces a list of words, which is the entry for the structure processing. In this module, some characters are transformed into words in order to avoid any interference with Prolog predefined elements. These are then restored in their original form when the final output is produced. This is an important issue to keep in mind, since some elements in the lexicon must take these transformations into account.

The module functions.pl contains a variety of basic utilities, which you may use for various purposes besides the present software.

3.Execution Schema and Structure of Control

The <TextCoop> engine is a meta-interpreter written in standard Prolog. Meta-interpretation is a well-known technique in Logic Programming which is very convenient for developing e.g. alternative processing strategies or demonstrators. It also allows us to realize a fast prototyping. The strategy implemented in <TextCoop> is quite similar to the Prolog strategy. However there are some major differences you need to be aware of.

The <TextCoop> engine considers, for a given text, rule clusters one after the other. Therefore, rule clusters must be organized in a cascade that describes the cluster execution order. The cascade must be declared in the decl.pl file as follows, where eng is the cascade name:

```
cascade(eng,
 [circ-eng,
  condition-eng,
  purpose-eng,
  restate-eng,
   illus-eng]).
```

The whole text is inspected for each rule cluster, one after the other. If a cluster does not produce any result, there is no failure and the next one is activated. In case you wish to define several cascades, you must define additional identifiers. In our example, the first argument of the cascade predicate is its identifier: eng.

During execution, you can see in the Prolog window the different steps of the cascade with the intermediate files being produced and compiled. Several cascades can be used sequentially if needed or relevant (e.g. for modularity purposes or for different types of tasks on a text which are not related).

Within a cluster, rules are considered one after the other, from the first to the last one, similarly to Prolog strategy. However, there is a major difference with Prolog due to Dislog rule format. The string of words to process is traversed from left to right (the code which is in the archive also provides a right to left strategy), at each step (i.e. for each word), the engine attempts to find a rule in that cluster (starting by the first one in the cluster) that would start by this word (via derivation or lexical inspection). If this succeeds, then the rule considered at this point, independently of its position in the cluster, is activated. If the whole rule succeeds, the result is produced (annotated text) and no other rule is considered in that cluster. If the rule fails at some point then backtracking occurs.

For example, consider the following abstract sentence to process:

```
[a,n,a,d,f,b,c],
```

and the following (simplified) set of rules:

```
s --> d, f.

s --> a, b.

s --> a, d.
```

When parsing the string, the first a and then the n are considered without any success since there is no s → a, n rule, but then the second occurrence of a is the left-corner of the second and third rules above. The second rule fails since no b is found after a, but the third rule succeeds. Note that in DCGs the first rule would have succeeded with a partial parser because the sentence contains the sequence [d,f], but since it comes later than the sequence [a,d] in the reading order, it does not succeed in Dislog. This strategy favours left-extraposed or left-sensitive structures in case there are several candidates for a sentence. This situation is the most common in discourse analysis, where, in most language, marks such as connectors appear to the left of the structures (Stede 2012).

Note that if the second rule were:

```
s -->

   a, gap, b.
```

then it would have succeeded first on the segment :

[a,n,a,d,f,b]

with gap = [n,a,d,f].

In our approach, when a rule in a cluster succeeds, for efficiency reasons, no other rule in that cluster is considered and there is no backtracking at any further stage in that cluster. For users who want to recognize several occurrences of the same structure in a sentence, it is best to write a complex rule that repeats the structure to find. This is not so elegant, but this limits backtracking and entails a much better efficiency. It is also possible to suppress a cut (noted as !) in the meta-interpreter code, but this is not recommended.

It should be noted that other processing strategies can easily be implemented in <TextCoop>. For example, in the interpreter, we give an example of a right-to-left processing strategy which may be of interest for rules where the relevant tags appear at the end of the rule, i.e. to the right of the rule, although this is quite unusual in discourse analysis. Similarly, it is also possible to write another processing strategy that would proceed rule by rule in a cluster and check whether a rule can be applied at any place in the sentence to process, instead of proceeding word by word and looking for the rule that succeeds the first. This is another strategy, which may be less efficient.

4.The Art of Writing Dislog Rules

Some readers may remain sceptical on their ability to write rules in Dislog. In fact, our experience shows writing rules is very easy because of the declarative character of Dislog and of logic programmes in general. This is the case here for rules as well as for constraints. A few days of practice should suffice to be able to write rules with a certain degree of sophistication. Some familiarity with Prolog and DCGs is however useful, see (Pereira et al. 1980) and, for a survey, (Saint-Dizier 1994) among others.

The ease of writing rules and the adequacy of the rule formalism with respect to corpus observations are major properties that any rule system must offer. To write well-designed rules, experiments are needed over a large number of domains and applications in particular on the way to identify rules, to generalize them, and to reach a certain level of linguistic adequacy and predictability. Another challenge is to identify, whenever possible, a comprehensive set of linguistic marks that would make these rules as unambiguous as possible in a certain application context.

Authoring tools and rule interface systems are useful for various kinds of operations including checking for duplicates and overlaps among large sets of rules. While some tools are available for sentence processing (e.g. (Sierra et al. 2008)), there is no such tool customized for discourse.

A number of investigations have been realized to identify linguistic marks on several discourse relations (Rosner et al 1992), (Redeker, 1990) (Marcu 1997), (Takechi et al. 2003) and (Stede 2012). These mostly establish general principles and methods to extract terms characterizing these relations.

In our approach to discourse analysis, rules are written by hand (i.e. rules do not result from automatic learning procedures from corpora). Although this is not the main trend nowadays, we feel this is the most reliable approach given the complexity and variability of discourse structures and the need to elaborate semantic representations. Let us briefly review the different steps involved in the rule production.

The first step, given a discourse function one wants to investigate, is to produce a clear definition of what it exactly represents and what its scope is, possibly in contrast with other functions. This is realized via a first corpus analysis where a number of realizations of this function over several domains are collected, analyzed and sorted by decreasing prototypicality order. This must be realized not by a single person but preferably by a few people, in a collaborative manner, and in connection with the literature, in order to reach the best consensus.

Then a larger corpus must be elaborated possibly using bootstrapping tools. Morpho-syntactic tagging contributes to identifying regularities and frequencies. From this corpus, a categorization of the different lexical resources which are needed must be elaborated. Then rules and lexical entries can be produced. Rules should be expressed at the right level of abstraction to account for a certain level of predictability and linguistic adequacy. This means avoiding low level rules (one rule per exceptional case) or too high level rules which would be difficult to constrain. In order to clearly identify text spans involved in a given discourse function, rules must be well-delimited, starting and ending by non-terminal

or terminal symbols which are as specific of the construct as possible. Each rule should implement a particular form of a discourse function.

In general, the number of rules needed to describe a discourse relation (which form a cluster of rules) ranges from 5 to about 25 rules. In average, about 8 to 10 are really generic, while the others relate more restricted situations. This means that managing such a set of rules and evaluating them for a given function on a test corpus is feasible.

The next step is to order rules in the cluster, starting by the most constrained ones considering the processing strategy implemented in <TextCoop>. In general, the most constrained rules correspond to less frequent constructions than the generic ones, which could be viewed as the by-default ones. In this case, this means for the processing system going through a number of rules with little chances of success, involving useless computations. As an alternative, it is possible to start by generic rules if (1) they correspond to frequently encountered structures and (2) they start by typical symbols not present in the beginning of other rules. In this case, there is no ambiguity and backtracking will occur immediately. This is a compromise, frequently encountered in Logic Programming that needs to be evaluated by the rule author for each situation.

Overlap of new rules with already existing ones (in the same cluster or in other clusters) must be investigated since this will generate ambiguities. This is essentially a syntactic task that requires rule inspection. This task could certainly be automated in an authoring tool. If it turns out that ambiguities cannot be resolved, then preferences must be stated: a certain relation must be preferred to another one. Preferences can then be coded in the cascade, starting with the preferred rule clusters, then the recognition of competing rules can be excluded via constraints, as presented in the previous chapter (zone exclusion for example).

The last stage for rule writing is the development of selective binding rules and possibly correction rules for anomalous situations. Selective binding rules are relatively easy to produce since they are based on the binding of two already identified structures. These are essentially based on already produced html tags. Structure variability, long-distance relations or dislocations are automatically managed by the <TextCoop> engine, in a transparent way.

Finally, the rule writer must add the cluster name at the right place in the cascade and possibly state constraints as explained in Chapter 1.

Although there are important variations, the total amount of work for encoding from scratch a discourse relation of a standard complexity, including corpus collection, readings and testing should take a maximum of about one month full-time. This is a very reasonable amount of time considering e.g. the workload devoted to corpus annotation in the case of a machine learning approach.

We feel the quality of manual encoding is also better, in particular rule authors are aware of the potential weaknesses of their descriptions. Next, if a rule or a small set of rules are already available in an informal way with the need of minor revisions, then encoding this small set in Dislog is much faster: checking for needed lexical resources, writing the rules, checking overlaps and testing the system on a toy text should not take more than a day or two for a somewhat trained person. Our current environment contains about 280 rules describing 16 discourse structures associated with argumentation and explanation. These rules are essentially the core rules for the 16 discourse structures: it is clear that they can be used as a kernel for developing variants or more specific rules for these structures, or for structures that share some similarities. This should greatly facilitate the development of new rules for trained authors as well as for novices.

Coming back to an authoring tool, it is necessary at a certain stage to have a clear policy to develop the lexical architecture associated with the rule system. Redundancies (e.g. developing marks for each

function even if functions share a lot of them) should be eliminated as much as possible via a higher level of lexical description. This would also help update, reusability and extensions.

5.Illustrations: Discourse relations

To conclude this chapter, let us comment with some detail a few examples provided in the archive. The goal is to allow the reader to be able to write his own rules.

The Circumstance relation

This function analyses the circumstances under which an instruction must be carried out. In procedural documents it is in general expressed in a very simple way.

The main rule is the following:

```
forme(circ-eng,

E, S,

[expr(EXP,circ,E,E1),

gap([],[ponct,co],E1,E2,Saute1),

ponct(Ponct,co,E2,S)],

[],

['<circumstance>', EXP, Saute1, '</circumstance>', Ponct ]).
```

The three first arguments in that rule are the rule identifier, and the input output strings (E,S). Next comes the rule body, where an expression of type circ is first expected (a word that introduces a circumstance, as given in the lexical entries below). Then a gap will skip any kind of words until a punctuation mark is found. The next argument is empty ([]) since there is no reasoning involved at this level. The rule ends by a list where the original sentence is reproduced with XML marks inserted at a certain place, which can be decided freely. To reproduce the original sentence, variables that appear in the symbols are reused in this latter structure in an appropriate order. For example, EXP represents the expression, Saute1 what gap has skipped and Ponct is the punctuation symbol. In this last structure of the rule, an opening XML tag starts the expression, followed by EXP, Saute1 and then the closing XML tag and Ponct. It is possible to have the punctuation mark before the XML tag. In that case, the result is:

```
['<circumstance>', EXP, Saute1, Ponct, '</circumstance>']
```

Tags are given between quotes since they represent strings of characters. Note that if you wish to put the words of an input sentence in a different order, it is possible to insert the variables corresponding to words in a different order in this last argument, as shown above for the punctuation. Tags may also contain variables elaborated in the rule or in the reasoning part.

In the lexicon, in DCG format, typical marks expressing circumstance are for example the following:

```
expr([when],circ) --> [when].

expr([once],circ) --> [once].
```

```
expr([as,soon,as],circ) --> [as,soon,as].

expr([after],circ) --> [after].

expr([before],circ) --> [before].

expr([until],circ) --> [until].

expr([till],circ) --> [till].

expr([during],circ) --> [during].

expr([while],circ) --> [while].
```

In these lexical entries, the string of words corresponding to the mark appears in the first argument, the second argument contains the type of the mark, which is an arbitrary symbol (here circ). Since the type is in this case a unique constant, it is not represented as a list of features, but this last option is recommended if one wants uniform representations of lexical entries. Note that some of these marks are ambiguous and may also be used in other discourse functions. For example *while* may also be used to express contrasts.

A structure such as the following is then produced:

this will allow you to make adjustment for square < circumstance> once the plumbing is connected < /circumstance>

where "once" is the mark for the circumstance function.

The Illustration relation

Let us consider here two typical rules for the illustration function, among the 20 rules we have defined. These two rules allow the recognition of the satellite part, the term which is illustrated being much more difficult to recognize.

```
% case 2a: recognizes: (for example : ……. )

forme(illus-eng,

 E, S,

[ponct(Ponct,po,E,E1),

ill_exempl(Ex,fe,E1,E2),

ponct(Ponct1,co,E2,E3),

gap([],[ponct,pf],E3,E4,Saute1),

ponct(Ponct2,pf,E4,S)],

[],

['<illustration>', Ponct, Ex, Ponct1, Saute1, Ponct2, '</illustration>']).


% case 2b: recognizes: ( …… , for example )

forme(illus-eng,
```

```
E, S,
[ponct(Ponct,po,E,E1),
gap([],[ponct,co],E1,E2,Saute1),
ponct(Ponct1,co,E2,E3),
ill_exempl(Ex,fe,E3,E4),
ponct(Ponct2,pf,E4,S)],
[],
['<illustration>', Ponct, Saute1, Ponct1, Ex, Ponct2, '</illustration>']).
```

The first rule above (case 2a) deals with illustrations which are included between parentheses starting with a mark, e.g.:

(for example, ducks, geese and hens)

If we concentrate on the fourth argument, the structure to recognize, this argument starts by the recognition of a punctuation of type po (opening parenthesis). The second symbol is the illustration mark (lexical entry ill_exempl) which is here of type fe (standing for "for example" or equivalent marks), then follows a punctuation of type co, a gap (to skip the elements of the illustration itself), and the pattern ends by a punctuation mark of type closing parenthesis, noted pf.

The result is the concatenation of the original strings, in their reading order, preceded and followed by the appropriate XML marks. It is crucial to use different variable names for the different symbols which are used (several punctuations for example), as required in Prolog syntax.

The second rule above (case 2b) recognizes illustrations where the list of elements of the illustration appear right after the opening parenthesis, they are then followed by a comma and the mark "for example", or equivalent:

(ducks, geese and hens, for instance)

Lexical entries corresponding to these rules are, for example:

```
ponct(['.'],co) --> ['.'].
ponct([','],co) --> [','].
ponct([':'],co) --> [':'].
ponct([';'],co) --> [';'].
ponct(['parentouvr'],po) --> ['parentouvr'].
ponct(['parenthferm'],pf) --> ['parenthferm'].

ill_exempl([for,example],fe) --> [for,example].
ill_exempl([for,instance],fe) --> [for,instance].
```

These lexical entries have the same format as above. Note that parentouvre and parenthferm respectively refer to opening and closing parenthesis: since these symbols belong to the Prolog language, when processing a text, it is advised to replace any parenthesis by such a word to avoid any problem. When producing the output text, it is then possible to restore the original parenthesis.

As the reader may note it, the typing of marks is a delicate problem, it may also seem somewhat *ad hoc*. It is sometimes necessary to develop different kinds of typing for distinct discourse structures. In that case, this may involve duplicating lexical entries. Similarly, in the rules above, we have introduced a different identifier for the illustration marks (ill_exempl). The goal is to show that the rule author can choose the symbol identifiers he finds the most appropriate. This approach allows us to construct different modules of marks which are proper to each discourse structure. This may introduce some form of redundancy, but this greatly facilitates updates when they are local to a structure. It should also be noted that such marks are not so numerous, making the approach tractable.

The Reformulation relation

An implementation in Dislog of the reformulation function can be realized by the following rule:

```
% case 1: e.g.: in other words  X ;

forme(restate-eng,

E, S,

[opt(ponct(Ponct,cor,E,E1)),

expr_reform(EXP,reform,E1,E2),

gap([],[ponct,cor],E2,E3,Saute1),

ponct(Ponct2,cor,E3,S)],

[],

[Ponct, '<restatement1>', EXP, Saute1, '</restatement>', Ponct2]).
```

The fourth argument of the above rule (the pattern to find in a sentence) starts by an optional punctuation: this punctuation is found when the reformulation is not at the beginning of a sentence:

A, in other words B

otherwise, if the reformulation starts a sentence the punctuation is not present. Then, follows an expression typical of a reformulation, such as: *in other words*, *said differently*, etc. The gap corresponds to the contents of the reformulation and the pattern ends by a punctuation mark.

Lexical entries are then the following :

```
expr_reform([in,other,words], reform) --> [in,other,words].

expr_reform([to,put,it,another,way], reform)

                                --> [to,put,it,another,way].

expr_reform([that,is,to,say], reform) --> [that,is,to,say].
```

```
expr_reform([put,differently], reform) --> [put,differently].

expr_reform([said,differently], reform) --> [said,differently].

expr_reform([otherwise,stated], reform) --> [otherwise,stated].

expr_reform([ie],_) --> [ie].
```

and:

```
ponct([','],cor) --> [','].

ponct([';'],cor) --> [';'].
```

6. Illustrations: recognizing arguments

Requirements:

Form a specific class of argument, relatively easy to identify. A few examples are:

(1) ***Requirements composed of a modal applied to an action verb***: a large number of requirements are characterized by the use of a modal (*must*, *have to* or *shall*) applied to an action verb in the infinitive. Other modals, which are less persuasive (*should*, *could*) must be avoided. This construction has a strong injunctive orientation. The lexical type action denotes verbs which describe concrete actions. This excludes a priori (but this depends on the context) state verbs, psychological verbs and some epistemic verbs (*know*, *deduce*, etc.). Adverb phrases are optional, they often introduce aspectual or manner considerations:

```
requirement → modal, {advP}, verb(action, infinitive), gap, eos.
```

The solution, software or equipment shall support clocks synchronization with an agreed accurate time source.

In Dislog internal format:

```
forme(req-eng, E, S, [mdeb(MDEB,_,E,E1), gap([], [modal,man], E1,E2,Saute1),
   modal(Mod,man,E2,E3), skip(rien,2,E3,E4,Saute2), verb(V,[action,inf],E4,E5), gap([],
   [mfin,_], E5,E6,Saute3), mfin(MFIN,_,E6,S)], [],

        [MDEB, '<requirement>', Saute1, Mod, Saute2,

                    V, Saute3 , ' </requirement>' , MFIN]).
```

(2) ***Requirement composed of a modal, the auxiliary be and an action verb*** used as a past participle:

```
requirement → modal, aux(be), {advP}, verb(action, past-participle),
  gap, eos.
```

Where new safety barriers are required and gaps of 50 m or less arise between two separate safety barrier installations, where practicable, the gap must be closed and the safety barrier made continuous.

forme(req-eng, E, S, [mdeb(MDEB,_,E,E1), gap([], [modal,man], E1,E2,Saute1), modal(Mod,man,E2,E3), aux(AUX,be,E3,E5), skip(rien,2,E5,E6,Saute3), verb(V,[action, pass],E6,E7), gap([], [mfin,_], E7,E8,Saute4), mfin(MFIN,_,E8,S)], [],

[MDEB, '<requirement>' , Saute1, Mod, AUX, Saute3,

V, Saute4 , ' </requirement>' , MFIN]).

(3) A special case are requirements which use an ***expression of type 'conformity'*** instead of an action verb, as, e.g. *to be compliant with*, *comply with*. The auxiliary *be* is included into this expression. The modal must be present to characterize the injunctive character of the expression:

`requirement → modal, {advP}, expr(conform), gap, eos.`

All safety barriers must be compliant with the Test Acceptance Criteria.

forme(req-eng, E, S, [mdeb(MDEB,_,E,E1), gap([], [modal,man], E1,E2,Saute1), modal(Mod,man,E2,E3), expr(EXPR, conform, E3, E5), gap([], [mfin,_], E5,E7,Saute4), mfin(MFIN,_,E7,S)], [],

[MDEB, '<requirement>' , Saute1, Mod, EXPR, Saute4, ' </requirement>' , MFIN]).

With, among others :

Expr([be,compliant,with], conform) → [be,compliant,with].

Warnings:

Also forms frequently encountered type of argument. These can be characterized on the basis of linguistic marks. Their main structure can be summarized by the following rules. Lexical elements may be quite diverse depending on the type of domain and language level.

Warnings and prevention arguments basically explain and justify an action, a decision or a behavior. These are very frequent in most types of technical documents.
Formulations with a negative polarity are frequent, their structure is given in (Saint- Dizier 2012) and summarized here. These representations originate from corpus analysis.

The structure of a conclusion is:
(1) prevention verbs like avoid' NP / to VP (*avoid hot water*)
(2) do not / never / ... VP(infinitive) ... (*never expose this product to the sun*)
(3) it is essential, vital, ... (to never) VP(infinitive).
it is essential that you switch off electricity before starting any operation.

Implementation for this third case:

forme(c-avt-eng, E, S,
[mimperatif(IMP,_,E,E1), gap([], [verb, [action, inf]], E1,E2,Saute1),
verb(V,[action,inf],E2,E3),

```
        gap([], [mfin,_], E3,E4,Saute2),  mfin(MFIN,_,E4,S)], [],
   [ '<concl-avt>' , IMP, Saute1,   V, Saute2, '</concl-avt>', MFIN ]).
```

With a local grammar :

```
mimperatif([it,is,Int,ADJ,Intr],_,[it,is|E1],S) :-
  intensifier(Int,_,E1,E2),
  adj(ADJ,imperatif,E2,E3),
  introprop(Intr,_,E3,S).

intensifier([very],_) --> [very].
intensifier([absolutely],_) --> [absolutely].
intensifier([really],_) --> [really].
intensifier([extremely],_) --> [extremely].
intensifier([highly],_) --> [highly].
intensifier([utterly],_) --> [utterly].
intensifier([also],_) --> [also].   % pas intensifieur mais apparaît souvent
intensifier([],_) --> [].

introprop([to],_) --> [to].
introprop([that,you],_) --> [that,you]. % enlever you ?
introprop([],_) --> [].

adj([capital],imperatif) --> [capital].
adj([compulsory],imperatif) --> [compulsory].
adj([crucial],imperatif) --> [crucial].
adj([essential],imperatif) --> [essential].
adj([fundamental],imperatif) --> [fundamental].
adj([imperative],imperatif) --> [imperative].
adj([important],imperatif) --> [important].  % etc.
```

Supports are realized by one of the following syntactic schemas:
(1) negative causal connector + infinitive risk verb,
(2) negative causal mark + risk verb,
(3) positive causal connector + VP(negative form),
(4) positive causal connector + prevention verb.

The grammatical and lexical elements in these constructions are in particular:
- negative connectors: otherwise, under the risk of, (e.g. *otherwise you may damage the connectors*).
- risk verb class: risk, damage, etc. (e.g. *in order not to risk your fingers*) or verbs of a ”conservative” type : *preserve*, *maintain*, etc. (e.g. *so that the axis is maintained vertical*)
- prevention verbs: *avoid*, *prevent*, etc. (e.g. *in order to prevent the card from skipping off its rack*).
- positive causal mark and negative verb form: *in order not to*, (e.g. *in order not to make it too bright*).
- modal SV: *may*, *could*, (e.g. *because it may be prematurely stop due to the failure of another component*).

Implementation of the first case:

```
forme(s-avt-eng, E, S,
[expr(EXP,cons,E,E1),
          gap([],[verb,[action,consneg]],E1,E4,Saute1), verb(V2,[action,consneg],E4,E5),
    gap([], [mfin,_], E5,E7,Saute2),  mfin(MFIN,_,E7,S)], [],
  [ '<support-avt>' , EXP, Saute1, V2,  Saute2, '</support-avt>', MFIN ]).
```

With e.g.:

```
expr([because],cons) --> [because].
expr([otherwise],cons) --> [otherwise].
```

Binding a conclusion with an adjacent support:

```
forme(liage-c-s-avt-eng, E,S,
[tag(B1,conclavt,E,E1), gap([], [tag,finconclavt], E1,E2,Saute1), , tag(B2,finconclavt), E2, E3), tag(B3,
suppavt,E3,E4), gap([], [tag,finsuppavt], E1,E2,Saute2), tag(B4,finsuppavt,E2,S)], [],
  ['<warning>', B1, Saute1,B2, B3, Saute2, B4, '</warning>']).

tag([<, concl, -, avt, >],conclavt) --> [<, concl, -, avt, >].
tag([<, /, concl, -, avt, >],finconclavt) --> [<, /,concl, -, avt, >].  % etc.
```

Performing or advice arguments
are less imperative than the previous ones, they express advices or task evaluations. These are also very frequent, in particular in documents designed for novices.

Conclusions have in general the following structure:
(1) advice or preference expression followed by an instruction. The preference expression may be a verb or a more complex expression: *is advised to*, *prefer*, *it is better*, *preferable to*, *etc.*, *prefer professional products*
(2) expression of optionality or of preference followed by an instruction: *our suggestions:* ..., or expression of optionality within the instruction: *use preferably an J78 Key*.

Their supports follow the following schemas:
(1) causal connector + performing NP,
(2) causal connector + performing verb,
(3) causal connector + modal-performing verb,
(4) performing proposition.

Resources or structures are, for example:
performing verbs: *allow*, *improve*,
performing NP: (e.g. *for a better performance*, *for more competitive fares*),
performing proposition. A comprehensive example is e.g. *<conclusion> we advise you to have small bills < /conclusion>, it is easier to tip and to pay your fare that way*.

BIBLIOGRAPHY

Adam, J.M. 1987, "Types de Sequences Textuelles Elementaires", Pratiques n°56, Metz.

Alred, G. J., Brusaw, C. T., and Walter E. O. 2012, *Handbook of Technical Writing*. St Martin's Press, New York.

Ashley, K.,D., Desai, R., Levine, J.M. 2002, *Teaching case-based argumentation concepts using dialectic arguments vs. didactic explanations*, ITS 2002, Lecture notes in computer science.

Ament, K. 2002, *Single Sourcing. Building modular documentation*, W. Andrew Pub.

Amgoud, L., Parsons and S., Maudet, N. 2001, Arguments, Dialogue, and Negotiation, in: 14th European Conference on Artificial Intelligence, Berlin.

Amgoud, L., Bonnefon, J.F., Prade, H. 2005, An Argumentation-based Approach to Multiple Criteria Decision, in 8th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty, ECSQARU'2005, Barcelona.

Aouladomar, F. 2005, Towards Answering Procedural Questions, in : KRAQ'05 - IJCAI workshop, Edinburgh, F. Benamara, P. Saint-Dizier (Eds.), pp. 24-36.

Aouladomar, F. 2005, A Semantic Analysis of Instructional Texts, IWCS05, Tilburg.

Bal, B.K. and Saint-Dizier, P. 2010, Towards Building Annotated Resources for Analyzing Opinions and Argumentation in News Editorial, LREC, Malta.

Barcellini, F., Gross, C., Albert, C. and Saint-Dizier, P. 2012, Risk Analysis and Prevention: LELIE, a Tool dedicated to Procedure and Requirement Authoring, LREC, Istanbul.

Barenfanger, M. and Hilbert, M. 2006, Cues and constraints for the relational discourse analysis of complex text types – the role of logical and generic document structure, proceedings CID'06.

Béguin, P. Design as a mutual learning process between users and designers Interacting with computers, 15 (6), 2003.

Benamara, F. and Saint-Dizier, P., Advanced Relaxation for Cooperative Question Answering. in New Directions in Question Answering, Mark Maybury (ed), MIT Press, 234-254, 2004.

Bieger, G.R. and Glock, M.D. 1984-1985, "The Information Content of Picture-text Instructions", Journal of Experimental Education, 53, 68-75.

Bourse, S. and Saint-Dizier, P. 2011, "The language of explanation dedicated to technical documents", Syntagma, vol. 27, 2011..

Buddenberg, Arnold. 2011, Guidelines for writing requirements. (personal communication).

Canitrot, M., Roger, PY and Saint-Dizier, P. 2011, How-To Question-Answering: hints on extraction, LTC conference, Poznan.

Caminada, M. and Amgoud, L. 2007, "On the evaluation of argumentation formalisms", Artificial Intelligence Journal, V.171 (5-6), 286-310.

Carlson, L., Marcu, D., Okurowski, M.E. 2001, Building a Discourse-Tagged Corpus in the Framework of Rhetorical Structure Theory. In Proceedings of the 2nd SIGdial Workshop on Discourse and Dialog, Aalborg.

Chomsky, N. 1986, *Barriers*, Linguistic Inquiry Monograph nb. 13, MIT Press.

Chomsky, N. 1990, *Some concepts and consequences of the theory of Government and Binding*, Linguistic Inquiry Monograph nb. 6, MIT Press.

Colmerauer, A. 1978, Metamorphosis Grammars, in Natural language understanding by computers, L. Bolc (ed.), LNCS no. 63, Springer Verlag.

Couper-Kuhlen, E. and Kortmann, B. 2000, *Cause, Condition, Concession, Contrast: Cognitive and Discourse Perspectives*, (Topics in English Linguistics, No 33), de Gryuter.

Cruse, A. 1986, *Lexical Semantics*, Cambridge University press.

Davidson, D. 1980, *Essays on Actions and Events*, Oxford: Oxford University Press.

Declerck, R., and Reed, S. 2001, *Conditionals: A comprehensive empirical analysis*, W. de Gruyter.

Delin, J., Hartley, A., Paris, C., Scott, D. and Vander Linden, K. 1994, Expressing Procedural Relationships in Multilingual Instructions, Proceedings of the Seventh International Workshop on Natural Language Generation, pp. 61-70, USA.

Delpech, E. and Saint-Dizier, P. 2008, Investigating the structure of procedural texts for answering How-to Questions, Language Resources and Evaluation Conference (LREC), Marrakech, European Language Resources Association (ELRA).

Di Eugenio, B. and Webber, B.L. 1996, "Pragmatic Overloading in Natural Language Instructions", International Journal of Expert Systems.

Dixon, R.M.W. 2000, A typology of causatives: form, syntax and meaning, in Changing Valency: Case Studies in Transitivity, Dixon, R.M.W. and Alexandra Y. Aikhenvald, eds.: 30–83. New York: Cambridge University Press.

Donin, J., Bracewell, R. J., Frederiksen, C. H. and Dillinger, M. 1992, "Students' Strategies for Writing Instructions: Organizing Conceptual Information in text", Written Communication 9, 209-236.

Dowty, D. 1991, "Thematic Proto-roles and Argument Selection", Language, vol. 67(3).

Eemeren, F.H. and van Grootendorst, R. 1984, *Speech acts in argumentative discussions: A theoretical model for the analysis of discussions directed towards solving conflicts of opinion*, Foris Publications.

Eemeren, F.H. and van Grootendorst, R. 1992, *Argumentation, communication, and fallacies: A pragma-dialectical perspective*, Lawrence Erlbaum Associates.

Eemeren, F.H. van, (ed). 2002,, *Advances in pragma-dialectics*, Newport News, Vale Press.

Fellbaum, C. 1998, *WordNet An Electronic Lexical Database*, The MIT Press.

Fontan, L. and Saint-Dizier, P. 2008, Analyzing the explanation structure of procedural texts: dealing with Advices and Warnings, International Symposium on Text Semantics (STEP 2008), Johan Bos (Eds.), Association for Computational Linguistics (ACL).

Ferrari, G. 1988, Preliminary steps toward the creation of a discourse and text resource, In Proceedings of the First International Conference on Language Resources and Evaluation (LREC), Granada.

Fiedler, A. and Horacek, H. 2001, Argumentation in Explanations to Logical Problems, in Proceedings of ICCS 2001, Springer LNCS 2073, pp. 969–978.

Garcia-Villalba, M. and Saint-Dizier, P. 2012, Some Facets of Argument Mining for Opinion Analysis, in proceedings of COMMA'12, IOS Press, Vienna.

Gardent, C. 1997, Discourse tree adjoining grammars, report nb. 89, Univ. Saarlandes, Saarbrucken.

Garnier, M. 2010, Correcting errors produced by French speakers: the case of misplaced adverbs, Calico, Amherst.

Garnier, M. 2011, Correcting errors in N+N structures in the production of French users of English, EuroCall, Nottingham.

Gazdar, G. and Mellish, C. 1989, Natural Language Processing in Prolog: An Introduction to Computational Linguistics, Addison Wesley.

Gnesi, S. and Lami, G., 2005, “An Automatic Tool for the Analysis of Natural Language Requirements”, in IJCSSE (International Journal of Computer Systems Science & Engineering) special issue, Volume 20(1).

Grady, J. O. 2006, *System Requirements Analysis*, Academic Press, USA.

Grice, H. P. 1978, Logic and Conversation, in P. Cole and J. L. Morgan (eds) Syntax and Semantics vol. 3, Speech Acts, Academic Press 113-127.

Grosz, B. and Sidner, C. 1986, "Attention, intention and the structure of discourse", Computational Linguistics 12(3).

Hull,E., Jackson,K. and Dick, J. 2011, *Requirements Engineering*, Springer.

Jin, W., Simmons, R. 1987, Question Answering with Rhetorical Relations, IEEE conf. on AI.

Helbig., H. 2005, *Knowledge representation and the semantics of natural language.* Cognitive Technologies, Springer-Verlag.

Keil, F.C. and Wilson, R.A. 2000, *Explanation and Cognition*, Bradford Book.

Kintsch, W. 1988, "The Role of Knowledge in Discourse Comprehension: A Construction Integration Model", Psychological Review, vol. 95(2).

Kosseim, L. and Lapalme, G. 2000, "Choosing Rhetorical Structures to Plan Instructional Texts", Computational Intelligence, Blackwell, Boston.

Lakoff, R. 1971, If, ands and buts about conjunction, Studies in Linguistic Semantics, Holt, Reinhart and Winston (ed) MIT Press.

Lasnik, H. and Uriagereka, J. 1988, *A Course in GB syntax*, MIT Press, 1988.

Lemarie, J., Lorch, R. F., Eyrolle, H. and Virbel, J. 2008, "SARA: A text-based and reader-based theory of text signalling", Educational Psychologist, 43, 27.

Luc, C., Mojahid, M., Virbel, J., Garcia-Debanc, C. and Pery-Woodley, M-P. 1999, *A Linguistic Approach to Some Parameters of Layout: A study of enumerations*, In R. Powerand D. Scott (Eds.), Using Layout for the Generation, Understanding or Retrieval of Documents, AAAI 1999 Fall Symposium, pp. 20-29.

Longacre, R. 1982, "Discourse Typology in Relation to Language Typology", Sture, Allen (ed.), Text Processing, Proceeding of Nobel Symposium 51, Stockholm, Almquist and Wiksell, 457-486.

Mann, W. and Thompson, S. 1988, "Rhetorical Structure Theory: Towards a Functional Theory of Text Organisation", TEXT 8 (3) pp. 243-281.

Mann, W. and Thompson, S.A. (eds) 1992, *Discourse Description: diverse linguistic analyses of a fund raising text*, John Benjamins.

Marcu, D, 1997, The Rhetorical Parsing of Natural Language Texts, proceedings of ACL97.

Marcu, D. 2000, *The Theory and Practice of Discourse Parsing and Summarization*, MIT Press.

Marcu, D. 2002, Au unsupervised approach to recognizing Discourse relations, proceedings ACL02.

Masthoff, J., Reed, C. and Grasso, F. (eds) 2008, *Symposium on Persuasive Technology*, in conjunction with the AISB 2008, Convention Communication, Interaction and Social Intelligence, Aberdeen.

Miltasaki, E., Prasad, R., Joshi, A. and Webber, B. 2004, Annotating Discourse Connectives and Their Arguments, new frontiers in NLP.

Moeschler, J. 2007, The role of explicature in communication and in intercultural communication, in Kecskes I. et al. (eds), Explorations in Pragmatics, Linguistic, Cognitive and Intercultural Aspects, Berlin, Mouton de Gruyter.

Moeschler, J. 2002, Speech act theory and the analysis of conversation, in Vandervecken D. et al. (eds), Essays in Speech Act Theory, Amsterdam, John Benjamins.

Mollo, V. and Falzon, P. 2004, "Auto and allo-confrontation as tools for reflective activities", Applied Ergonomics, 35 (6), 531-540.

Nuseibeh, B. and Easterbrook, S. 2000, "Requirements Engineering: A Roadmap", ICSE'00 Proceedings of the 22nd international conference on Software engineering, P.37-46.

O'Brien, Sharon, 2003, Controlling Controlled English, An Analysis of Several Controlled Language Rule Sets. School of Applied Language and Intercultural Studies Dublin City University Dublin 9, Ireland.

Pereira, F. 1981, "Extraposition Grammars", Computational Linguistics, vol. 9-4.

Pereira, F. and Warren, D. 1980, "Definite Clause Grammars for Language Analysis", Artificial Intelligence vol. 13(3).

Pohl K. 2010, *Requirements Engineering: Fundamentals, Principles, and Techniques*, Springer Publishing Company.

Pollock, J.L. 1974, Knowledge and Justification, Princeton university Press.

Redeker, G. 1990, Ideational and Pragmatic Markers of Discourse Structure, Journal of Pragmatics, vol. 14.

Reed, C. 1988, Generating Arguments in Natural Language, PhD dissertation, University College, London.

Reed, C.A. and Long, D.P. 1997, Content Ordering in the Generation of Persuasive Discourse, in Proceedings of 15th International Joint Conference on Artificial Intelligence, Nagoya, Japan.

Reed, C.A. and Long, D.P. 1998, Generating the Structure of Argument, in Proceedings of the 17th International Conference on Computational Linguistics and 36th Annual Meeting of the Association for Computational Linguistics (COLING-ACL98), Montreal, Canada.

Reed, C. and Grasso, F. 2007, "Recent Advances in Computational Models of Natural Argument", International Journal of Intelligent Systems, 22(1).

Rosner, D. and Stede, M. 1992, Customizing RST for the Automatic Production of Technical Manuals, in R. Dale, E. Hovy, D. Rosner and O. Stock eds., Aspects of Automated Natural Language Generation, Lecture Notes in Artificial Intelligence, pp. 199-214, Springler-Verlag.

Saaba A. and Sawamura, H. 2008, Argument Mining Using Highly Structured Argument Repertoire, proceedings EDM08, Niigata.

Sage, A.P. and Rouse, W.B. 2009, *Handbook of Systems Engineering and Management*, 2nd Edition, Wiley, USA.

Saito, M., Yamamoto, K. and Sekine, S. 2006, Using Phrasal Patterns to Identify Discourse Relations, proceedings ACL06.

Saint-Dizier, P. 1994, *Advanced Logic programming for language processing*, Academic Press.

Sampaio, A., Loughran, N., Rashid, A. and Rayson, P. 2005, Mining Aspects in Requirements, Early Aspects 2005: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, Chicago.

Schank, R. 1986, *Explanation Patterns: Understanding Mechanically and Creatively*, Laurence Erlbaum.

Schauer, H. 2006, From elementary discourse units to complex ones, Sigdial workshop, ACL06.

Schriver, K. A. 1989, Evaluating text quality: The continuum from text-focused to reader focused methods, IEEE Transactions on Professional Communication, 32, 238-255.

Searle, J. R. and Vanderveken, D. 1985, *Foundations of Illocutionary Logic*, Cambridge University Press.

Sierra, G., Alarcon, R., Aguilar, C. and Bach, C. 2008, "Definitional verbal patterns for semantic relation extraction", Journal of terminology, 14-1.

Spenader, J. and Lobanova, A. 2007, Reliable Discourse Markers for Contrast. Eighth International Workshop on Computational Semantics, Tilburg.

Stabler, E. 1992, *The logical approach to syntax*, MIT Press.

Stede, M. 2012, *Discourse Processing*, Morgan and Claypool Publishers.

Takechi, M., Tokunaga, T., Matsumoto, Y. and Tanaka, H. 2003, Feature Selection in Categorizing Procedural Expressions, The Sixth International Workshop on Information Retrieval with Asian Languages (IRAL2003), pp.49-56.

Talmy, L. 2001, *Towards a Cognitive Semantics*, vol. 1 and 2, MIT Press.

Taboada, M. and Mann, W.C. 2006, "Rhetorical Structure Theory: Looking back and moving ahead", Discourse Studies, 8(3), 423-459.

Taboada, M. 2006, "Discourse markers as signals (or not) of rhetorical relations", Journal of Pragmatics, 38:567–592, 2006.

Van der Linden, K. 1993, Speaking of Actions Choosing Rhetorical Status and Grammatical Form in Instructional Text Generation Thesis, University of Colorado.

Van Dijk, T.A. 1980, *Macrostructures*, Hillsdale, NJ: Lawrence Erlbaum Associates.

Walton, D. 1992, *The Place of Emotion in Argument*, Pennsylvania state university press.

Walton, D., Reed, C. and Macagno, F. 2008, *Argumentation Schemes*, Cambridge University Press.

Webber, B.L. and Di Eugenio, B. 1990, Free Adjuncts in Natural Language Instructions, Proceedings COLING-90, Helsinki.

Webber, B. 2004, "D-LTAG: extending lexicalized TAGs to Discourse", Cognitive Science 28, pp. 751-779, Elsevier.

Weiss, Edmond H. 1990, *Practical exercises for technical writing*, The Oryx Press, Westport.

Weiss, Edmond H. 1991, *How to write usable user documentation*, The Oryx Press, Westport.

Wierzbicka, A. 1987, *English Speech Act Verbs*, Academic Press.

Wolf, F. and Gibson, E. 2005, "Representing Discourse Coherence: A Corpus-Based Study", Computational Linguistics 31(2): 249_288.

Wright, von G.H. 2004, *Explanation and understanding*, Cornell university Press.

Wright, S.E. and Budin G. 2001, *Handbook of Terminology Management: Application-oriented Terminology Management*, volume 2. Kent State University. University of Vienna. p.872.

Zuckerman, I., McConachy, R. and Korb, K. 2000, Using Argumentation Strategies in Automatic Argument Generation, INLG00.