Language-based document processing

Dennis S. Arnon, Isabelle Attali, Paul Franchi-Zannettacci

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Sophia Antipolis
B.P. 109
06561 Valbonne Cedex
France
Tél.: 93 65 77 77

Rapports de Recherche

N°1731

Programme 2
Calcul symbolique, Programmation
et Génie logiciel

LANGUAGE-BASED DOCUMENT PROCESSING

Dennis S. ARNON
Isabelle ATTALI
Paul FRANCHI-ZANNETTACCI

Juillet 1992

Language-based Document Processing

Dennis S. Arnon
Xerox PARC,
3333 Coyote Hill Road,
Palo Alto, CA 94304, USA,
Arnon.pa@Xerox.COM

Isabelle Attali
INRIA Sophia Antipolis - BP 93
06902 Sophia Antipolis - France
ia@trinidad.inria.fr

Paul Franchi-Zannettacci
CNRS I3S - Univ. Nice Sophia Antipolis
250 Av. Einstein - 06560 Valbonne - France
pfz@essi.cerisi.fr

Abstract

This paper proposes an application of programming environments generation to structured documents manipulation. We use Centaur as a formal tool to model and implement logical and physical structure, logical editing and layout processing, document analysis, re-use and conversion for a sample class of documents: scientific articles including equations and figures. To make connections with real document systems, we choose to give two particular external forms to the logical structure: Tioga source and LaTeX source.

From the specifications of the logical and physical structures of the Article document class on one hand, and, on the other hand, the specification of the layout processing (viewed as its semantics according to the Tioga or the LaTeX layout model) and other semantic tools, the Centaur system automatically generates structured environments for Tioga and LaTeX documents and conversions between them.

1 Introduction

Our aim is to apply software tools that have been developed for the effective modelling and implementation of the formal semantics of programming languages to build document manipulation systems according to given classes of documents. We will postulate that the documents we consider have a logical structure that we can model as an abstract syntax tree, and that we can associate some semantics with such trees.

In order to illustrate our approach, we define a document class for scientific articles. The partitioning of the logical structure for the *Article* document class is the following: the content items (words, figures, equations or citations) are partitioned into paragraphs, which are in turn partitioned into subsections, which in turn are partitioned into sections. Finally, the whole document is logically viewed as a collection of sections.

From the physical point of view, an Article document is partitioned into lines, which in turn are partitioned into blocks, which are partitioned into pages. The document is physically viewed as a collection of pages. The natural abstraction of this hierarchy of partitions is a preordered tree. Finally, a *layout* tree, representing the complete physical hierarchical structuring of an Article document, might be displayed on a screen or printed out.

Besides these two internal representations for the logical and the layout structure of the Article document class, we need to give an external form to the document. In other words, we need to define the commands for a user to build an Article (make a new section, insert a figure, or a citation). We choose two well-known examples of document preparation systems and get inspired of their concrete syntax: Tioga [16] and LaTeX [11]. Thus a user can either use the Tioga or the LaTeX concrete syntax to build an Article document. The layout process is accomplished accordingly to the Tioga or LaTeX layout model.

We choose LaTeX because it is the famous representative of the traditional "batch formatting" approach: the layout structure is strictly derived, in an offline fashion, from the logical structure. In the spirit of language-based interactive system generators (Synthesizer Generator [15], Centaur [4]), we associate with the logical structure and the layout structure two distinct abstract syntaxes, and we consider the formatting as a translation.

On the contrary, Tioga belongs to the family of interactive *wysiwyg*[1] editors inside the Cedar Environment: the user sees only a laid-out rendering of the document at any time and does not have a clear understanding of the internal model (a tree structure) of the document. Most Tioga users are prepared to view the appearance of the document as a significant conveyor, a "logical consequence", of its meaning. We want to put a formal system behind Tioga documents and get a "Cedar-independent" semantics in order to make easier archival and interchange of Tioga documents. Moreover, in the absence of a machine-independent semantics, it is quite impossible to design and evaluate extensions to Tioga.

One of the goals of the work reported in this paper is to provide a clear, sound, computationally effective representation of both the structures, and the layouts derived from structures, of Tioga documents.

We also want to work towards sound meta-tools for generation of interactive, *wysiwyg* editors from formal specifications. We must pay close attention to the nature and feasiblity of such "inverse" mappings within that formal system:

[1] What You See Is What You Get

the user in some fashion acts upon the layout, and we need to receive the user actions and translate them into actions upon the logical structure, which then (hopefully in an incremental fashion) give rise to a modified layout, which is then displayed to the user. Thus, we will readily provide an interactive *wysiwyg* editor for LaTeX documents (see [6] for a similar environment for TeX).

Attribute grammar-based systems (Cornell Synthesizer [15], GIGAS [5]) have been demonstrated to be sufficient for certain classes of "documents" (program source files, mathematical notation [8], ODA [18] business letters [1], G-LOTOS [9]). A key aspect of attribute grammars (AG's) that has made generation of such editors possible is the existence of efficient incremental evaluation algorithms for suitable classes of AG's. However it seems clear that the richness of constraints that must be supported for realistic document layout takes one outside the power of such AG systems.

This is the main reason we have been led to investigate the use of Centaur and its dedicated formalism for declarative semantics, Typol. The Typol formalism has been shown to be a generalization of Attribute Grammars [3]. At the present, Typol specifications are not evaluated in an incremental manner. In fact, this property has been investigated in the case of a Typol specification for a type-checker [10], equivalent to Attribute Grammars [2]. More recently, work started for a larger class of Typol programs [3]. So, we are optimistic about interactive use of Typol in the next future.

One of the goals of the examples we consider in this paper is to explicitly identify and separate successive "degrees of complexity" of document layout situations, so that we can see clearly how far Attribute Grammars can take us, and at what point we need the generality of Typol, with the attendant risk of a decline in performance.

One may describe the issue of complexity in document layout rules as a question of the nature of the *constraints* that one is allowed to impose on the layouts of the components of the logical structure. In fact there are multiple "degrees of freedom" to be addressed in layout constraints, for example, the amount of "non-locality of reference" that is permitted, the kinds of arithmetic supported (integer, fixed point, floating point, exact), number of layout "streams" allowed, "structural" constraints, etc. We expect that the nature of the layout constraints will be important in determining whether a given formal system can be adequate for document modelling. We believe that much more work is needed in expressing, classifying, and finding appropriate systems for, constraints in document modelling.

Since our approach is based on the Centaur system, we aim to answer to the following questions:

- how Centaur can give us a more consistent and precise model for Tioga and LaTeX behaviors compared to their wired implementation ?
- can we consider layout processing and document conversions as transla-

tions among logical structures that we can specify and execute within the Centaur system ?

- is Centaur powerful enough to provide well-founded tools for document management within a large data-base of documents, for instance searching, comparison, editing, collecting ?

In section 2, we briefly recall the major aspects of the generic programming environments, and namely the Centaur system. In section 3, we present the specifications for generating environments for Tioga and LaTeX documents. Section 4 concludes the paper.

2 Generating programming environments from formal specifications

In the Programming Languages setting, it is now commonly accepted to specify an interactive programming environment for a particular language, in a generative manner, via the specification of its syntax and its semantics.

The specification of syntax includes concrete and abstract syntaxes of the language. From this specification, one can derive a parser that transforms the textual form of a program (an ASCII source file) into a structural representation (an abstract syntax tree). There also exists the reverse mapping, called pretty-printing or unparsing, that transforms a structural representation into a concrete layout.

In the Centaur system, the specifications of concrete and abstract syntax, together with their relationship, are written in the Metal language [12]. On the other hand, pretty-printing of abstract syntax trees is described in PPML [14].

The abstract syntax is used to check the validity of editing operations. An abstract syntax is the many-sorted algebra of well-typed terms over a set P of sorts and a set O of operators with their signatures on P.

Beyond syntactic aspects of a language, there are also semantic aspects when the designer of a given language wants to provide, in addition to a structure editor, useful tools such as a type-checker, or an interpreter. Semantic tools deal only with the language constructs and do not know anything about the concrete syntax. In the Centaur system, these aspects are treated with the Typol formalism [7].

2.1 The Metal formalism

A Metal specification is a collection of grammar rules, with tree-building functions that describe what abstract syntax tree should be synthesized from a production rule. All non-terminals appearing on the right hand side of a rule

must appear on the left hand side of at least one other rule. Symbols and keywords must appear within quotes. Having successfully parsed a sentence in the language, we construct a piece of the abstract syntax tree with the instantiated non-terminals and terminals of the right hand side of the production rule.
Here is an example for a conditional instruction:

```
<instruction> ::= "if" <expression> "then" <instruction_s>
                  "else" <instruction_s>  "end" "if" ;
      if(<expression>, <instruction_s>.1, <instruction_s>.2)

    -- operators definition
    -- the language construct for if is a ternary operator
            if -> EXPR INSTR_S INSTR_S ;
            instr_s  -> INSTR + ... ;
    -- sorts definition
            INSTR_S ::= instr_s;
            INSTR   ::= if while ... ;
```

2.2 The PPML formalism

A PPML program specifies the textual representation of an abstract syntax tree. It consists in a sequence of rules of the form:

$$pattern \rightarrow format$$

where *pattern* is an abstract syntax tree containing variables and *format* is a formatting specification in terms of a box language (horizontal, vertical, combined) with separators. Ppml provides five types of boxes:

h to concatenate elements horizontally;

v to concatenate elements vertically;

v1 to concatenate elements vertically with indentation based on the box surrounding the current box rather than the adjacent box;

hv to concatenate elements horizontally until the end of the line is reached. Boxes that don't fit on the current line are folded on to the following line(s) starting at the left margin of the current hv box;

hov to concatenate elements horizontally until the end of the line is reached. If the elements don't all fit on the current line, the elements are all concatenated vertically.

Here is an example for a conditional instruction:

```
#if(*expr, *instr1, *instr2) ->
      [ <v>  [ <hv> "if" *expr]
             [ <hv> "then" *instr1]
             [ <hv> "else" *instr2]
             [ <h> "end" "if"]
      ] ;

    -- the rule for the ternary operator if
    -- # is necessary because "if" is a keyword for PPML
```

2.3 The Typol formalism

Typol is a language to implement Natural Semantics [13] inside the Centaur Programming Environment. A semantic specification is represented by a set of inference rules of the form:

$$\frac{H_1 \vdash T_1 : S_1 \qquad \cdots \qquad H_n \vdash T_n : S_n}{H \vdash T : S}$$

which constitutes, together with type information, a Typol program [7]. Each sequent

$$H_i \vdash T_i : S_i$$

is called a premise. The sequent

$$H \vdash T : S$$

is the conclusion of the rule. The terms $T, T_1, ..., T_n$ are language objects (they are the subject of the corresponding sequent). The tuple H_i represents inherited attributes and the tuple S_i represents synthesized attributes.

To compute the semantic value, say S_0, of an abstract syntax term T_0, given some initial environment H_0, means to prove the goal sequent $H_0 \vdash T_0 : S_0$ in this deductive system. This presentation allows quite compact and readable specifications of semantic properties such as static semantics, dynamic semantics, and translations. Moreover, it is an executable formalism because each rule can be translated into an Horn clause.

Here is an example of Typol rules for specifying the dynamic semantics of a conditional instruction:

$$\frac{s \vdash \text{EXP} : \mathit{true} \qquad s \vdash \text{STMS}_1 : s_1}{s \vdash \text{if EXP then STMS}_1 \text{ else STMS}_2 \text{ fi} : s_1} \tag{1}$$

$$\frac{s \vdash \text{EXP} : \mathit{false} \qquad s \vdash \text{STMS}_2 : s_1}{s \vdash \text{if EXP then STMS}_1 \text{ else STMS}_2 \text{ fi} : s_1} \tag{2}$$

Both rules apply to an abstract tree whose root is an "if". The rule (1) expresses that if the expression is evaluated to *true*, the result of the if-statement is the result of the then-part, in the same environment. Otherwise, rule (2) is tried, the expression is evaluated to *false*, and the result of the if-statement is the result of the else-part.

Some Prolog programmers may ask: "*since Typol rules are translated into Horn clauses and evaluated with the Prolog engine, why not directly write Horn clauses ?*"
The answer is twofold:

1. From the specification point of view:

   - the Typol sequents distinguish their parameters: the subject belongs to an abstract syntax, the inherited attributes are considered as parameters, and the synthesized attributes are the resulting parameters.
   - this makes it possible to provide the designer with a powerful type-checker for verifying the positions and the types of the parameters and the data flow between them.
   - finally, the order of the Typol rules in a specification is not relevant, as the order of the premises in a rule.

2. From the implementation point of view:

   The particularities of Typol (compared to Horn clauses) made it possible to provide optimized implementations of Typol specifications out of Prolog (Lisp or C, via an attribute evaluator generator or directly ([2, 3]). These implementations eliminate the space overflows experienced in Prolog, replace the Prolog full unification by pattern-matching and provide an incremental re-computation of the attributes.

3 Generating a document manipulation system

The concepts used in language-based system generators can be applied to the field of document manipulation. One has first to define a document class as a formal object and establish its relationship with its components, if any. We propose to consider documents as terms on a typed algebra. Note that this approach is general enough for multi-media documents and multi-view processing and defines sets of similar documents, i.e. a class of documents, namely scientific articles. Our formal model based on abstract syntax makes it possible to determine if a given document belongs to such a class, and to execute the processing tools associated with this class.

With each document we associate two internal structures: the logical structure (which is composed of sections, subsections, paragraphs, ...) and the layout (or physical) structure (which is composed of lines, blocks and pages).

We define two abstract syntaxes for the logical and layout structures: the operators in the logical abstract syntax are *word, figure, citation, paragraph, subsection, section, ...*; the operators in the layout abstract syntax are *word, figure, line, block, page, ...*

From a practical point of view, the two abstract syntaxes are not enough to generate an environment for a given document class. One has to specify association rules between the concrete syntax of the document class and the logical abstract syntax. In other words, this step is required to automatically generate a parser for the document class.

For the back-end process (showing the layout structure), we specify the unparsing for the layout abstract syntax and provide a previewer to display it on a screen.

We also specify associated tools to complete our document manipulation environment, such as:

- a logical structural editing (with tree transformations),
- a layout process (a translation from a given logical tree to a layout tree),
- document management tools (acting on one or more documents, such as searching, combining, deleting, ...).

Using Centaur, all these semantic tools have to be specified in the Typol formalism.

3.1 Document logical structure

We define in this section a particular class of documents, namely scientific articles.

An Article is composed by a header, a body, and an end part. The body is a collection of sections, possibly composed of subsections or paragraphs, and so on. A paragraph can be a collection of words or a figure (given by its height and width, and its caption). Citations are allowed everywhere. The end part is an acknowledgment part followed by a list of references.

We give below partial abstract syntax for the logical structure of our document class.

```
abstract syntax
        article -> HEADER BODY END ;
        header  -> TITLE AUTHOR ABSTRACT KEYWORDS ;
        title   -> WORD_OR_CITATION_S ;
        body -> SECTION_S ;
        section_s -> SECTION + ... ;
        section -> TITLE PARAGRAPH_S SUBSECTION_S ;
      ...
        ARTICLE ::= article ;
        HEADER ::= header ;
        TITLE ::= title ;
        SECTION_S ::= section_s ;
        SECTION ::= section ;
        SUBSECTION_S ::= subsection_s ;
        SUBSECTION ::= subsection ;
        PARAGRAPH_S ::=  paragraph_s ;
        PARAGRAPH ::=  paragraph ;
      ...
```

To handle document conversion between Tioga documents and LaTeX documents, we use parsing and unparsing techniques of a common abstract syntax to convert between two different concrete representations (see 3.2). So we associate with the abstract syntax for Articles two different concrete syntaxes; this leads to the generation of two parsers: one for Tioga articles, the other for LaTeX articles.
The partial concrete syntax for the LaTeX document class is given below:

```
    <article> ::=
        "\documentstyle" "{Article}"
        <header>
        <body>
        <end>
        "\end{document}" ;
      article(<header>, <body>, <end>)
    <abstract> ::= "\begin{abstract}" <text> "\end{abstract}";
        abstract(<text>)
    <section> ::= "\section{" <title> "}" <paragraph_s> <subsection_s> ;
        section(<title>, <paragraph_s>, <subsection_s>)
    ...
```

Here is the partial concrete syntax for the Tioga document class[2]:

[2] In fact, this concrete syntax is not relevant for Tioga users: they have the Tioga *wysiwyg* editor (in the Cedar environment) and unparse the document into our concrete representation in order to use the Centaur environment.

```
<article> ::= "root[" <header> <body> <end> "]" ;
  article(<header>, <body>, <end>)
<abstract> ::= "abstract[" <text> "]";
    abstract(<text>)
<section> ::= "head2[" <title> ", " <paragraph_s> ", " <subsection_s> "]" ;
    section(<title>, <paragraph_s>, <subsection_s>)
...
```

3.2 Document conversion

We use the syntactic facilities of the Centaur system for document conversion between Tioga and LaTeX articles. We define a common abstract syntax for the logical structure of our document class Article and we associate two different concrete representations for Tioga and LaTeX articles. These two concrete representations are linked to the abstract syntax in two processes : the parsing process (described in Metal) and the unparsing process (described in PPML).

Therefore, our document conversion is based on two parsers and two unparsers (see Figure 1). For more complex conversions, i.e. with two distinct logical abstract syntaxes (for instance Article document class and Book document class) we need more powerful translations than the parsing/unparsing process (namely, we may use Typol).

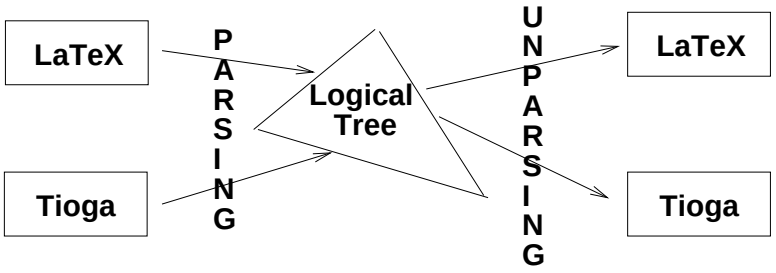

Figure 1: Document conversion

We give in Appendix A an example of an Article in the Tioga format and in Appendix B the same article processed by LaTeX .

3.3 Document layout structure

We give below a partial layout abstract syntax for the Article document class.

```
        physical_article -> PAGE_S ;
        page_s -> PAGE * ... ;
        page -> NUM BLOCK_S ;
-- NUM is PAGE_NBR
        block_s -> BLOCK * ... ;
        block -> LINE_S_OR_FIGURE ;
        line_s -> LINE * ... ;
        figure -> NUM NUM NUM ;
--  these are LINE_NBR VERT_COORD HORIZ_COORD ;
        line ->  LINE_INFO WORD_S ;
        line_info -> NUM NUM NUM NUM NUM WORD WORD ;
--  these are LINE_NBR VERT_COORD HORIZ_COORD GLUE SIZE FONT FACE
        word_s -> WORD * ... ;
    ...
        PHYSICAL_ARTICLE ::= physical_article ;
        PAGE_S ::= page_s ;
        PAGE ::= page ;
        BLOCK_S ::= block_s ;
        BLOCK ::= block ;
        LINE_S_OR_FIGURE ::= line_s figure ;
        LINE ::= line ;
        LINE_INFO ::= line_info ;
        WORD_S ::= word_s ;
        WORD ::= word font face ;
        NUM ::= page_nbr line_nbr vert_coord horiz_coord glue size ;
```

3.4 From logical structure to layout structure

In this section, we describe the layout processing that computes a layout tree from a logical tree. We specify this translation in Typol and we use the Typol facilities such as structured induction on subtrees, unification and backtracking for section numbering, forward references and multi-column. For the layout process, we handle line and page breaking, glue, as well as section, subsection and figure identification according some *style* described in a rule-based fashion (Prolog facts). This style defines, for instance, the size, the font and the style of entities such as a title, an abstract or a standard paragraph; it also defines the (top and bottom) leading, the indentation and the justification of each entity.

The computation of the physical structure is done according to the following rules:

- each part (head, body, end) begins a new page;
- each paragraph or figure starts a new block; nested blocks may incur additional left indentation according to the document style;
- titles and figures are not breakable on different pages;

- for figures, the first choice is not to break between the figure and its caption;
- sections and figures are numbered in Dewey decimal notation.

The atomic-lexical elements are ASCII words. The Typol program computes the "appropriate" place for each word from left to right in the text, accordingly to current layout informations, rules and styles such as the page width, the page height, the word width (w.r.t. the style). At the same time, the physical structure is augmented with each word.

Let us focus on the kernel of the Typol program: the three rules which deal with a word are given in Figure 2.

```
compute_current_line_width(Lex, Style, Clw −> Clw')
hold_in_page_width(Clw')
append_new_word(Ps, Lex, Style, Cph −> Ps')
-------------------------------------------------------------- (1)
Ps, _, Style, Clw, Cph |− word Lex : Ps', Clw', Cph ;

compute_current_line_width(Lex, Style, Clw −> Clw')
not_hold_in_page_width(Clw')
compute_current_page_height(Style, Cph −> Cph')
hold_in_page_height(Cph')
append_new_line(Ps, Lex, Style, Cph'−> Ps', Clw')
-------------------------------------------------------------- (2)
Ps, _, Style, Clw, Cph |− word Lex : Ps', Clw', Cph' ;

compute_current_line_width(Lex, Style, Clw −> Clw')
not_hold_in_page_width(Clw')
compute_current_page_height(Style, Cph −> Cph')
not_hold_in_page_height(Cph')
append_new_page(Ps, Lex, Style, Cph'−> Ps', Clw'', Cph'')
-------------------------------------------------------------- (3)
Ps, "break", Style, Clw, Cph |− word Lex : Ps', Clw'', Cph'';
```

Figure 2: Limited backtracking on word

A decision must be made here for any word matching one of the three rules. All the needed information (the current physical structure *Ps*, the "breakable" property *"break"*, _, the style *Style*, the current line width *Clw*, the current page height *Cph*) must be known. The result consists in three values: the augmented physical structure *Ps'*, the new current line width *Clw', Clw"* and the new current page height *Cph'*. Rule (1) describes the case when the current word fits on the current line (and the word is added to the physical structure). Note that the current page height is unchanged. Rule (2) expresses the fact that, when the current word does not fit on the current line, but fits on the current page, it must be added to the physical structure starting a new line. Rule (3)

describes the remaining case: the word does not fit on the current line nor on the current page. If the current paragraph can be broken into two pages (see the second inherited attribute), then the word is added to the physical structure starting a new page.

So the computation for one word may backtrack in a very limited way. Note there is no rule (3') for the "no_break" case; this forces the program to fail and backtrack up to the first word of the current paragraph (see Figure 3).

```
new_bloc("abstract", Cph −> Cph')
Ps, "no_break", "abstract", 0, Cph' |−
        word_s[word "abstract:".Words] : Ps', Clw', Cph''
──────────────────────────────────────────────────────────────────── (4)
Ps, Clw, Cph |− abstract(Words) : Ps', Clw', Cph'' ;

new_page("abstract" −> Ps', Cph')
Ps', "no_break", "abstract", 0, Cph' |−
        word_s[word "abstract:".Words] : Ps'', Clw', Cph''
──────────────────────────────────────────────────────────────────── (5)
Ps, Clw, Cph |− abstract(Words) : Ps'', Clw', Cph'' ;

Ps, Clw, Cph |− abstract(Words) : Ps, Clw, Cph ;                     (6)
   do writeln("the abstract is longer than the page height");
```

Figure 3: General backtracking on abstract

The rules given in Figure 3 describe what to do for the layout of the abstract. Rule (4) applies when the abstract fits on the current page. If, during the computation of the word sequence, there is a failure (none of the rule (1) (2) or (3) apply) the backtracking mechanism occurs until the failure of rule (4). Then rule (5) is tried, and the physical structure is augmented with a new page. If rule (5) fails again, the new page is removed (thanks to backtracking) and rule (6) is applied and produces an error message.

The Typol program contains similar rules for the title, the keywords, and standard paragraphs.

3.5 Document management tools

In this section, we briefly describe the tools we provide in our Centaur document manipulation environment. These tools are based on a context-sensitive analysis of one or more documents and make it possible to:

- on one document:
  - manage the bibliography (used and permitted citations);
  - check the references to sections or figures;
  - handle table of contents or cross-references;

- on a set of documents:
  - make a digest of publications (list of titles, authors, and abstracts);
  - build a cumulative bibliography;
  - gather figures.

4 Conclusion and future work

We wanted to apply techniques used for programming languages to the field of document processing. This approach has already been explored with Centaur in [17] for Hypertext applications. We claim that Centaur (with Typol) is powerful enough to express formal and executable specifications for layout processing, document conversion and document manipulation.

We are convinced that Attribute Grammars were not powerful enough to express so naturally the layout process (since we used a lot of unification and backtracking). We claim that Typol was a suitable specification formalism for this experiment, even though we are not totally satisfied with the resulting document manipulation environment, from a end-user point of view (mainly because of the lack of incrementality).
To make oursystem more user-friendly, we still have to provide some facilities such as:

- produce Interpress/Postcript (we just have a previewer on the screen);
- maintain the linkage of logical and layout structures;
- offer *wysiwyg* editing of logical structures;
- incrementally update the layout structure;
- compute the layout structure in a demand-driven way.

References

[1] D. Arnon & P. Franchi-Zannettacci "Context-sensitive Semantics as a Basis for Processing Structured Documents". WOODMAN'89, Rennes, 1989.

[2] Attali I. "Compilation de programmes Typol par attributs sémantiques" Thèse de Doctorat, Université de Nice, Avril 1989

[3] Attali I. and Chazarain J. "Functional Evaluation of Natural Semantics Specifications" INRIA Research Report 1218, May 1990, also in Proc. of WAGA "International Workshop on Attribute Grammars and their Applications" Paris sept 90, Lecture Notes in Computer Science 461.

[4] Borras P., Clément D., Despeyroux T., Incerpi J., Kahn G., Lang B., & Pascual V. "CENTAUR: the System", INRIA Research Report 777, 1987, and SIGSOFT'88, Third Annual Symposium on Software Development Environments, Boston, Nov. 88.

[5] Chabrier B., Franchi-Zannettacci P. & Lextrait V. "GIGAS : a Graphical Interface Generator by Attribute Specification". Le Génie logiciel et ses applications, Toulouse, 1988.

[6] Chen P., Coker J., Harrison M., MacCarrell J., and Procter S. "The VorTeX Document Preparation Environment". Proceedings Second European Conference on TeX for Scientific Documentation, Strasbourg, Lecture Notes in Computer Science 236, 1986.

[7] Despeyroux T. "TYPOL: a Formalism to Implement Natural Semantics" INRIA research report 94, 1988

[8] Franchi-Zannettacci P. "Attribute Specifications for Graphical Interface Generation", IFIP'89 World Congress Conference, San Francisco, 1989.

[9] Franchi-Zannettacci P. & Zarli A. "An Incremental and Graphical Structure-oriented Editor for G-Lotos", Proceedings of the IFIP Third International Conference on Formal Description Techniques (FORTE), 1990, Madrid, ELSEVIER Science Publishers B.V.

[10] Hascoët L., "Transformations automatiques de spécifications sémantiques", Thèse de Doctorat, Université de Nice, 1987

[11] Lamport L. "LATEX: a Document Preparation System" Addison-Wesley, 1986

[12] Kahn G., Lang B., & Mélèse B. "Metal : a Formalism to Specify Formalisms" Science of Computer Programming, volume 3, North-Holland, 1983

[13] Kahn G. "Natural Semantics" Proceedings of Symposium on Theoretical Aspects of Computer Science, Passau, Germany, Lecture Notes in Computer Science 247, feb 87

[14] Morcos-Chounet E. & Conchon A. "PPML: a General Formalism to Specify Pretty-printing" Proceedings of the IFIP Congress Dublin, North-Holland 1986

[15] Reps T. "Generating Language based Environments" M.I.T. Press, Cambridge, Mass, 1984

[16] Swinehart, D., Zellweger, P., Beach, R. & Hagmann, R. "A structural view of the Cedar programming environment", ACM Trans. Prog. Lang. Systems, 8, 4, 419-490, 1986

[17] Vercoustre A. M. "Edition structurée approche hypertexte: coopération et complémentarité", INRIA Research Report 1052, 1989.

[18] H. C. Weisz, I.R. Campbell-Grant, R. Hunter, R. Pierce, L.J. Zeckendorf, and B. J. Woods. Information Processing, Text and Office Systems, Office Document Architecture (ODA) and Interchange Format, Technical Report DIS 8613, International Standards Organization (ISO), March 1988.

Appendix A: the Tioga model

Appendix B: The LaTeX model