1-1989

A Logic Programming Elucidation of ODA - Document Descriptions and Processes

Howard A. Blair
Syracuse University, School of Computer and Information Science, blair@top.cis.syr.edu

Allen Brown Jr.

SU-CIS-90-32

A Logical Programming Elucidation of ODA - Document Descriptions and Processes

Howard A. Blair and Allen L. Brown, Jr.

January 1989

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, New York 13244-4100

SU-CIS-90-32

Working Paper:

A Logic Programming Elucidation of ODA Document Descriptions and Processes

Howard A. Blair and Allen L. Brown, Jr.

January 1989

School of Computer and Information Science
Syracuse University
Suite 4-116
Center for Science and Technology
Syracuse, NY 13244-4100

(315) 443-2368

Working Paper: A Logic Programming Elucidation of ODA Document Descriptions and Processes

Howard A. Blair
School of Computer and Information Science
Syracuse University
Syracuse, New York 13210
U.S.A.
blair@top.cis.syr.edu

Allen L. Brown, Jr.
Systems Sciences Laboratory
Webster Research Center
Xerox Corporation
Webster, New York 14580
U.S.A.
abrown.wbst128@xerox.com

*Appeared *Special Working Group 3: International Standards Organization*, Berlin, Jan. 1989.

January 13, 1989 8:25

We are pursuing a programme of research in document representation. The principal aim of this research is to develop a document description language that has a precise formal semantics, that is fully expressive of the constructs typical of traditional (procedural) document description languages, that is constraint-based, and that cleanly separates specifications of form and content. The research is currently in the first of three envisioned three phases. In the first phase we are formalising the Office Document Architecture (ODA) by faithfully translating ODA document descriptions into logic programmes. The translation utilises highly restricted forms of Prolog programmes.[1] In the second phase we will explore various enhancements of ODA's expressive power that are immediately apparent upon freeing the translation from having to adhere to the initial restrictive conventions. Finally, we will explore and articulate a constraint logic programming language having "built-in" constructs for expressing both primitive and composite document description concepts.

[1]Our translation is actually into definite clause grammars (DCG's) [4,9,10], which, in turn, have their own stylised translation into Prolog definite clauses.

In the present essay we sketch our translation (into a DCG framework) of ODA document descriptions and (layout) processes. As it turns out the resulting translation is closely related to so called functional attribute grammars [4]. Indeed, we hope eventually to exploit that relationship to enable efficient interpretation of the resulting translation. For now, however, we hope to convince our readers that (definite clause) grammars are a natural and powerful generalisation of the ODA framework, and that the ODA layout process can be specified entirely by *declarative* means by appealing to properties of the grammars in question.

1 Our view of ODA

As a language, ODA expresses a syntactically well-defined collection of *document constituents* of which the principal sorts are content portions (graphic characters, raster graphic elements and geometric graphic elements), logical objects and logical object classes, layout objects and layout object classes, and attributes. A logical (layout) structure is a tree-like arrangement of logical (layout) objects and object classes, with the trees' being "leafed" with content portion constituents. The logical structure of an ODA document is a partitioning of the document's content based on meaning. In that context, logical object classes are elements of generic logical structure from which a set of logical objects with common characteristics may be derived (*e.g.* composite logical objects representing sections) while logical objects are elements of a document having specific interpretations (*e.g.* particular chapters, sections and paragraphs). The layout structure of an ODA document is a partitioning of the document's content based on presentation. In that context, layout object classes are elements of a generic layout structure from which a set of layout objects with common characteristics may be derived (*e.g.* pages with common headers and footers) while layout objects are elements of a specific layout structure of document having specific geometric properties (*e.g.* particular pages and blocks). An attribute is an element of a document constituent that has a name and a value, and that expresses a characterisitic of that constituent or relationship with one or more other consituents (*e.g.* the "presentation style" attribute establishes the relationship between a basic component description and a presentation style).

The ODA language allows the composition of the above-mentioned con-

stituents into *document descriptions*, each of the latter being composed of a *document profile* and a *document body*. A document profile is a collection of predefined (and interpreted) attributes that apply globally to the document description. The document body consists of a generic logical structure, a generic layout structure, specific logical structure, specific layout structure, and style constituents. The last are predetermined (and interpreted) collections of attributes that explicitly and implicitly link logical constituents with layout constituents.

2 An initial attempt at translation

We translate particular ODA document descriptions into particular logic programme fragments by making a one-to-one mapping between certain ODA constituents and corresponding predicate definitions in the logic programme being constructed. Observe, however, that the real utility of ODA comes only through the descriptive interpretations of various attributes (*e.g.* presentation style) and processes (*e.g.* document layout). In fact, the interpretation of these attributes is the main task of *document processing* as exemplified by the layout process. These interpretations are given in [11] in a decidedly informal fashion. Indeed, the main task of formalising ODA is to define these interpretations *formally*. They will turn out to be other logic programme fragments relative to which we define *each* (and every) logic programming translation of an ODA document description. Hereafter, we shall refer to the translation of an ODA document description as a logic programme as the *data* description and to the logic programme interpretation of attributes (in the document processing context) as the *process* description. The latter rendering can be thought of as defining *interpreters* for various document processors.

The basic recipe for generating the data description is as follows: Generic (logical and layout) objects are represented as monadic predicates. We may think of these generic objects as *types* whose *tokens* are specific (logical and layout) objects. In particular, tokens are are individual terms in the logic programming language. Generic attributes, *i.e.* attributes of generic logical or layout objects, are represented as dyadic predicates. So, for example, the fact that the generic letter has a presentation style attribute with value `'letter_layout'` is represented by

```
presentation_style(X,V) :- letter(X), letter_layout(V)
```

which says that the `presentation_style` of the generic `letter` is the generic `letter_layout`. That a specific letter object `#Letter` had the specific presentation style `#Letter_Layout` would result from asserting the fact

```
presentation_style(#Letter,#Letter_Layout)
```

We illustrate the use of the above recipe by a partial translation of the specimen letter whose ODA document description is presented in Annex B of Part 2 of [11]. Specifically, our illustrations are drawn from sections B.5 and B.6 (respectively the "processable form document with generic logical structure and generic layout structure" and the "specific layout structure"). We translate structural aspects of the first seven entries of table B.4 (presenting the hierarchical object class descriptions the specimen letter) of [11] as:

```
letter(X) :-
  sequence(X,[U,V]),
  header(U),
  body(V).

header(X) :-
  sequence(X,[T,U,V,W]),
  date(T),
  addressee(U),
  subject(V),
  summary(W).

summary([]).
summary([X|Y]) :-
  summary_paragraph(X),
  summary(Y).
```

We leave the `date` predicate undefined because it is a basic logical object having no internal logical structure from ODA's point of view.[2] The generic attributes associated with the logical object classes would be given by:

[2]We could define basic logical objects in such a way as to accept exactly those strings of characters having the syntax of proper dates.

```
object_class(X,document_root) :- letter(X).
user_visible_name(X,"Letter") :- letter(X).

object_class(X,composite) :- header(X).
user_visible_name(X,"Header") :- header(X).

object_class(X,basic) :- date(X).
user_visible_name(X,"Date") :- date(X).
layout_style(X,Y) :- date(X),date_layout(Y).
offset(X,[trailing(710),right_hand(395)]) :- date(X).
content_architecture_class(X,processable_characters) :-
  date(X).

object_class(X,basic) :- addressee(X).
user_visible_name(X,"Addressee") :- addressee(X).
layout_style(X,Y) :- addressee(X),addressee_layout(Y).
content_architecture_class(X,processable_characters) :-
  addressee(X).

object_class(X,basic) :- subject(X).
user_visible_name(X,"Subject") :- subject(X).
layout_style(X,y) :- subject(X),subject_layout(Y).
presentation_style(X,Y) :-
  subject(X),
  subject_presentation(Y).
line_spacing(X,300) :- subject(X).
content_architecture_class(X,processable_characters) :-
  subject(X).

object_class(X,composite) :- summary(X).
layout_style(X,Y) :- summary(X),summary_layout(Y).
user_visible_name(X,"Summary") :- summary(X).

object_class(X,basic) :- summary_paragraph(X).
user_visible_name(X,"Summary paragraph") :-
  summary_paragraph(X).
layout_style(X,Y) :-
  summary_paragraph(X),
```

```
  summary_paragraph_layout(Y).
offset(X,[left_hand(705)]) :- summary_paragraph(X).
alignment(X,justified) :- summary_paragraph(X).
presentation_style(X,Y) :-
  summary_paragraph(X),
  summary_paragraph_presentation(Y).
content_architecture_class(X,processable_characters) :-
  summary_paragraph(X).
```

The translation of a specific logical document description has the same object-oriented structure as does the original ODA. That is, each object represented in Table B.6 has a corresponding term, realised as a so-called *gensym*. These gensyms are created as a side effect of a query, the nature of which need not worry us at this time. The query results in the creation of the gensyms `#Letter_1`, `#Header_2`, `#Body_3`, `#Body_paragraph_4`, `#Body_paragraph_5`, *etc.*, as well as the assertion of ancillary facts about those gensyms, such as

```
object_type(#Letter_1,letter).
subordinate(#Header_2,#Letter_1).
object_type(#Header_2,header).
subordinate(#Body_3,body,#Header_2).
object_type(#Body_3,body).
subordinate(#Body_paragraph_4,#Body_3).
object_type(#Body_paragraph_4,body_paragraph).
content_portion(#Body_paragraph_4,#A).
object_type(#A,graphic_characters).
subordinate(#Body_paragraph_5,#Body_3).
object_type(#Body_paragraph_5,body_paragraph).
content_portion(#Body_paragraph_5,#B).
object_type(#B,graphic_characters).
```

effectively translating the descriptive content of Table B.6.

The translation of generic layout structure of the specimen letter proceeds analogously to the translation of the generic logical structure of the generic logical structure. For the structural translation we have, in part:

```
letter_layout(X) :-
```

```
    sequence(X,[U,W]),
    header_page(U),
    body_page_pages(W).

 header_page(X) :-
    sequence(X,[S,T,U,V,W]),
    logo_frame(S),
    date_frame(T),
    addressee_frame(U),
    subject_frame(V),
    summary_frame(W).

logo_frame(X) :-
    sequence(X,[Y]),
    logo_block(Y).

logo_block(X) :-
    sequence(X,[Y]),
    logo(Y).

 body_page_pages([]).
 body_page_pages([X|Y]) :-
    body_page_page(X),
    body_page_pages(Y).

 body_page_page(X) :-
    components(X,[Y]),
    body_frame_frame(Y).
```

The generic attributes associated with the layout object classes would be given by:

```
object_class(X,document_layout_root) :- letter_layout(X).
user_visible_name(X,"Letter") :- letter(X).

object_class(X,page) :- header_page(X).
user_visible_name(X,"Header") :- header_page(X).
dimensions(X,[hd(9920),vd(14030)]) :- header_page(X).
```

```
object_class(X,frame) :- logo_frame(X).
position([hp(710),vp(730)]) :- logo_frame(X).
dimensions([hd(3685),vd(2495)]) :- logo_frame(X).

object_class(X,block) :- logo_block(X).
user_visible_name(X,"Logo") :- logo_block(X).
content_architecture_class(X,raster_graphics_elements) :-
  logo_block(X).
content_portion(X,#Logo).
```

The translation of the specific document layout structure parallels that of the specific document logical structure, with the object structure being given by facts of the following form:

```
object_type(#Letter_layout_5,letter_layout).
subordinate(#Header_layout_6,#Letter_layout_5).
object_type(#Header_page_6,header_page).
subordinate(#Logo_frame_7,#Header_page_6).
object_type(#Logo_frame_7,logo_frame).
subordinate(#Logo_8,#Logo_frame_7).
object_type(#Logo_8,logo_block).
```

For the most part we have left the generic attributes entirely undefined. For example, what does it *mean* for a the logical class `summary_paragraph` to have an `alignment` attribute of `justified` (*i.e.* `alignment(X,justified) :- summary_paragraph(X).`)? The real definition for this attribute lies in its intended interpreter, in this case the *layout process.* The layout process includes the *document layout process* and the *content layout process.* These processes are concerned with the creation of a specific layout structure which can be used by the *imaging process* to present the ODA-specified document in human perceptible form in a presentation medium. The document layout process creates a specific layout structure in accordance with the generic layout structure and information derived from the specific logical structure, the generic logical structure and layout styles (if present). (We sketched above the specific layout structure for the specimen letter without elucidating the processess by which this structure was created.) This process also determines the areas that are available within the created layout objects for the formatting of the document content, and is responsible for allocating the content to

these available areas. The content layout process is responsible for formatting (or laying out) the content portions into the available areas specified by the document layout process. This content layout process makes use of information contained in the presentation attributes that apply to those content portions. In order to better illuminate the idea of process description, we will now sketch a logic programming approximation of the document layout process as described in [11].

The jist of ODA's layout model is as follows: Each ODA content portion is mapped on to one or more blocks where the blocks may be generated "on the fly". The situation of multiple mapping arises when content layout permits a content object to be split and the (yet to be mapped) content cannot be fit into the space remaining in the containing frame. The content portions are totally ordered and it is in that order that blocks for content objects are generated. The order derives from the depth-first (pre-)ordering of the tree implicit in the specific content description. To capture the layout process We define the predicate `layout_process(X,Y,U,V)` where `V` is the specific layout structure (produced by side-effect) resulting from laying out the specific logical structure `U` (an instance of the generic logical structure `X`) according to the generic layout structure `Y`. In particular the query `?-layout_process(letter,letter_layout,#Letter_1,V)` will result in binding `V` to `#Letter_layout_5`, as well as the creation of subordinate specific objects and the asserting of the facts about them that we mentioned above.

The `layout_process` predicate just described can unquestionably be fully, and even *lucidly*, defined. Nonetheless, it is fundamenatally unsatisfying in that the answer (in the main) occurs by side-effect, and the definition mimics the usual procedural style of document layout. In the sections that follow we shall endeavour to address these objections by substantially raising the level of abstraction of the logic programming account of ODA data descriptions, and thereby enabling an entirely declarative account of process descriptions.

3 Translating ODA logical structure

Let us again consider the specimen document of Annex B of Part 2 of [11] in its processable form as presented in B.5. We proceed by focussing on the *structure diagram* notation for document structures as the fundamental representational framework for ODA documents, rather than focussing on

logical and layout (generic and specific) document objects as we did above. In particular, we view both generic and specific structure diagrams as definite clause grammars.[3] In fact, the grammars for generic structures (generic grammars) will serve as generators for grammars for specific structures (specific grammars). Roughly speaking, each node in each structure diagram will correspond to the left-hand-side of one or more productions in the corresponding grammar. The exact nature of those rules will depend on the structural relationships of the particular node to those below it in the structure diagram. To begin with, we have the productions:

```
root_grammar(G) --> letter_grammar(G).
root_grammar(G) --> doc_grammar(G).
root_form_grammar(G) --> letter_form_grammar(G).
root_form_grammar(G) --> doc_form_grammar(G).
```

which describe the root grammars corresponding to the root objects of the object-oriented descriptions. More explicitly these are interpreted (in part) as saying that any grammar G, that is a `letter_grammar`, is also a `root_grammar`.

Now we are ready to look at the logical `letter_grammar`'s definition:

```
letter_grammar(LG) -->
  letter_rule_grammar([LR]) + header_grammar(HG) +
  body_grammar(BG),
  {dunion([LR],HG,G1), dunion(G1,BG,LG)}.
```

which says that LG is a specific grammar for letters just in case it is a disjoint union of the (singleton) set of letter "rules" with the disjoint union of the set of rules in some specific header grammar and some specific body grammar. The SEQ operator is subsumed by the ordering of the items appearing on the right-hand-sides of productions. Note that the items between the braces are predicates that constitute side conditions on the (sets of) rules. It is through this mechanism that grammars can be "attributed" and thereby gain in a logic programming setting the power of attribute grammars. Specific grammars are multisets of rules, *i.e. unordered lists.* We can now proceed to give the remaining productions of the generic grammar for letters (which

[3]As it happens, many standard Prologs, including the C-Prolog and the Quintus™ Prolog that we use, have built-in facilities for translating DCG's into clauses. We implicitly rely on such facilities here and illustrate the resulting translations in the appendices.

we shall intersperse with additional commentary as appropriate) in the usual descending (toward terminals) fashion:

```
letter_rule_grammar([(letter --> header, body)]) -->
  [(letter --> header, body)].
```

The `letter_rule_grammar` above consists of precisely one rule that corresponds to the nodes `Letter`, `Header` and `Body` and their topological juxtaposition in figure B.8. We may regard this as saying that a specific letter has a header part and a body part. We should also point out that items embracketed in "`[]`" on the right-hand-sides of productions are terminal symbols of the grammar being developed.

```
header_grammar(HG) -->
  header_rule_grammar([HR]) + date_grammar(DG) +
  addressee_grammar(AG) + subject_grammar(SBG) +
  summary_grammar(SMG),
  {dunion(HR,DG,G1), dunion(G1,AG,G2), dunion(G2,SBG,G3),
   dunion(G3,SMG,HG)}.

header_rule_grammar([
    (header --> date, addressee, subject, summary)]) -->
  [(header --> date, addressee, subject, summary)].

date_grammar([DR]) --> date_rule_grammar([DR]).

date_rule_grammar([(date --> [Date], {is_date(Date)})]) -->
  [(date --> [Date], {is_date(Date)})].
```

In general, predicates of the form `is_...`, such as `is_date`, are applied to terminals of specific (logical) grammars. The terminals in question are the content portions of actual documents. The predicates not only test the well-formedness of the portions but also solicit user input. In this way a generic logical grammar used as a generator is in effect a syntax-directed editor for the class of documents denoted by the grammar. Having completed such an editing process, the result is a specific logical grammar bound to the variable `LG` of the production defining `letter_grammar`.

```
addressee_grammar([AR]) --> addressee_rule_grammar([AR]).
```

```
addressee_rule_grammar(
    [(addressee -->
        [Addressee], {is_addressee(Addresee)})]) -->
  [(addressee --> [Addressee], {is_addressee(Addresee)})].

subject_grammar([SR]) --> subject_rule_grammar([SR]).

subject_rule_grammar(
    [(subject --> [Subject], {is_subject(Subject)})]) -->
  [(subject --> [Subject], {is_subject(Subject)})].

summary_grammar(SG) -->
  summary_rule_grammar([R]) +
  summary_paragraph_grammar(SPG),
  {dunion([R],SPG,SG)}.

summary_rule_grammar([(summary --> RuleBody)]) -->
  [(summary --> RuleBody)], {summary_rule_body(RuleBody)}.

summary_rule_body(RuleBody) :-
  rep(RuleBody,summary_paragraph).
```

To account for the REP structure that appears in the definition of a Summary in figure B.9, we introduce the meta-syntactic predicate, rep. In effect we treat REPpetition as a construct for denoting finite sets of axioms all havinh the same form but with distinct local variables and mutually exclusive applicability, in other words, a kind of axiom scheme.

```
summary_paragraph_grammar([R]) -->
  summary_paragraph_rule_grammar([R]).

summary_paragraph_rule_grammar(
    [(summary_paragraph -->
        [Summary_paragraph],
        {is_summary_paragraph(Summary_paragraph)})]) -->
  [(summary_paragraph -->
      [Summary_paragraph],
```

```
        {is_summary_paragraph(Summary_paragraph)})].

  body_grammar(BG) -->
    body_rule_grammar([BR]) +
    paragraph_and_figure_grammar(PFG) +
    ending_grammar(EG) + signature_and_name_grammar(SNG),
    {dunion(BR,PFG,G1), dunion(G1,EG,G2), dunion(G2,SNG,BG)}.

  body_rule_grammar([(body --> RuleBody)]) -->
    [(body --> RuleBody)], {body_rule_body(RuleBody)}.

  body_rule_body(RuleBody) :-
    conjappend(FirstPart,(ending,signature_and_name),RuleBody),
    repcho(FirstPart,[paragraph,figure]).
```

repcho, like rep, is meta-syntactic constructor of axiom schemata. In this case it represents the ODA structure diagram combination of REP and CHO.

```
  paragraph_and_figure_grammar(PG) --> paragraph_grammar(PG).
  paragraph_and_figure_grammar(FG) --> figure_grammar(FG).
  paragraph_and_figure_grammar(PFG) -->
    paragraph_grammar(PG) + figure_grammar(FG),
    {dunion(PG,FG,PFG)}.
```

To capture the idea of alternation as denoted by CHO we simply use multiple productions, each corresponding to one of the possible CHOice combinations.

```
  paragraph_grammar([R]) --> paragraph_rule_grammar([R]).

  paragraph_rule_grammar(
      [(paragraph -->
          [Paragraph], {is_paragraph(Paragraph)})]) -->
    [(paragraph --> [Paragraph], {is_paragraph(Paragraph)})].

  figure_grammar(FG) -->
    figure_rule_grammar([R]) +
    drawing_grammar(DG) + caption_grammar(CG),
    {dunion([R],DG,G1), dunion(G1,CG,FG)}.
```

```
figure_rule_grammar([(figure --> drawing, caption)]) -->
  [(figure --> drawing, caption)].

drawing_grammar([DR]) --> drawing_rule_grammar([DR]).

drawing_rule_grammar(
     [(drawing --> [Drawing], {is_drawing(Drawing)})]) -->
  [(drawing --> [Drawing], {is_drawing(Drawing)})].

caption_grammar([CR]) --> caption_rule_grammar([CR]).

caption_rule_grammar([(caption -->
                          [Caption],
                          {is_caption(Caption)})]) -->
  [(caption --> [Caption], {is_caption(Caption)})].

ending_grammar([R]) --> ending_rule_grammar([R]).

ending_rule_grammar([(ending --> [])]) --> [(ending --> [])].

signature_and_name_grammar(SNG) -->
  signature_and_name_rule_grammar([SNR]) +
  signature_grammar(SG) + name_grammar(NG),
  {dunion([SNR],SG,G1), dunion(G1,NG,SNG)}.

signature_and_name_rule_grammar(
    [(signature_and_name --> signature, name)]) -->
  [(signature_and_name --> signature, name)].

signature_grammar([SR]) --> signature_rule_grammar([SR]).

signature_rule_grammar(
             [(signature -->
                 [Signature],
                 {is_signature(Signature)})]) -->
  [(signature --> [Signature], {is_signature(Signature)})].
```

```
name_grammar([NR]) --> name_rule_grammar([NR]).

name_rule_grammar([(name --> [Name], {is_name(Name)})]) -->
  [(name --> [Name], {is_name(Name)})].
```

4 Translating ODA layout structure

The translation of structure diagrams for generic and specific layout have the same story as for logical structure diagrams. We simply present the grammar below without further commentary. While each node of the generic logical structure diagram is mapped into productions with node_grammar on the left-hand-side, each node of the generic layout structure diagram will map into node_form_grammar.

```
letter_form_grammar(LFG) -->
  letter_form_rule_grammar([LR]) +
  header_page_form_grammar(HPFG) +
  body_page_form_grammar(BPFG),
  {dunion([LR],HPFG,G1), dunion(G1,BPFG,LFG)}.

letter_form_rule_grammar([(letter --> RuleBody)]) -->
  [(letter --> RuleBody)], {letter_frule_body(RuleBody)}.

letter_frule_body(RuleBody) :-
  conjappend(header_page,CdrRuleBody,RuleBody),
  rep(CdrRuleBody,body_page).

header_page_form_grammar([HPFG]) -->
  header_form_rule_grammar([HR]) +
  logo_frame_form_grammar(LFFG) +
  date_frame_form_grammar(DFFG) +
  addresee_frame_form_grammar(AFFG) +
  subject_frame_form_grammar(SBFFG) +
  summary_frame_form_grammar(SUFFG),
  {dunion([HR],LFFG,G1), dunion(G1,DFFG,G2),
   dunion(G2,AFFG,G3), dunion(G3,SBFFG,G4),
```

```
   dunion(G4,SUFFG,HPFG)}.

header_form_rule_grammar([(header_page -->
    logo_frame, date_frame, addressee_frame, subject_frame,
    summary_frame)]) -->
  [(header_page -->
      logo_frame, date_frame, addressee_frame,
      subject_frame, summary_frame)].

logo_frame_form_grammar(LFFG) -->
  logo_frame_form_rule_grammar([LFR]) +
  logo_block_form_grammar(LBFG), {dunion([LFR],LBFG,LFFG)}.

logo_frame_form_rule_grammar([(logo_frame -->
    logo_block)]) -->
  [(logo_frame --> logo_block)].

logo_block_form_grammar([LBR]) -->
  logo_block_form_rule_grammar([LBR]).

logo_block_form_rule_grammar([(logo_block -->
    [logo_block])]) -->
  [(logo_block --> [logo_block_thing])].

date_frame_form_grammar([DFR,DBFR]) -->
  date_frame_form_rule_grammar([DFR]) +
  date_block_form_rule_grammar([DBFR]).

date_frame_form_rule_grammar([(date_frame -->
    date_block)]) -->
  [(date_frame --> date_block)].

date_block_form_rule_grammar([(date_block -->
    [Date_out], {is_date_out(Date_out)})]) -->
  [(date_block --> [Date_out], {is_date_out(Date_out)})].

addressee_frame_form_grammar([AFR,ABFR]) -->
```

```
  addressee_frame_form_rule_grammar([AFR]) +
  addressee_block_form_rule_grammar([ABFR]).

addressee_frame_form_rule_grammar([(addressee_frame
    --> addressee_block)]) -->
  [(addressee_frame --> addressee_block)].

addressee_block_form_rule_grammar([(addressee_block -->
    [Addressee_out], {is_addressee_out(Addresse_out)})]) -->
  [(addressee_block -->
      [Addressee_out], {is_addressee_out(Addresse_out)})].

subject_frame_form_grammar([SFR,SBFR]) -->
  subject_frame_form_rule_grammar([SFR]) +
  subject_block_form_rule_grammar([SBFR]).

subject_frame_form_rule_grammar([(subject_frame -->
   subject_block)]) -->
  [(subject_frame --> subject_block)].

subject_block_form_rule_grammar([(subject_block -->
    [Subject_out],{is_subject_out(Subject_out)})]) -->
  [(subject_block --> [Subject_out],
  {is_subject_out(Subject_out)})].

summary_frame_form_grammar([SMFR,SMBFR]) -->
  summary_frame_form_rule_grammar([SMFR]) +
  summary_block_form_rule_grammar([SMBFR]).

summary_frame_form_rule_grammar([(summary_frame -->
    summary_block)]) -->
  [(summary_frame --> summary_block)].

summary_block_form_rule_grammar([(summary_block -->
    [Summary_out], {is_summary_out(Summary_out)})]) -->
  [(summary_block -->
      [Summary_out], {is_summary_out(Summary_out)})].
```

```
body_page_form_grammar(BPFG) -->
  body_page_rule_grammar([BPRG]) +
  body_frame_form_grammar(BFFG),
  {dunion([BPRG],BFFG,BPFG)}.

body_frame_form_grammar([BFRG]) -->
  body_frame_rule_grammar([BFRG]).
```

5 Translating ODA attributes

We translate ODA attributes by essentially the same device we employed in section 2, but this time predicating relative to sets of rules (often being grammars) or terminals rather than objects. Thus we write `attr(A,X,V)`, meaning that the set of rules `X` has an `A` attribute with value `V`. We can distinguish among various categories of attributes with clauses:

```
/* an identification attribute */
attr(A,X,V) :- id_attr(A,X,V).
/* a relationship attribute */
attr(A,X,V) :- rel_attr(A,X,V).
/* a layout attribute */
attr(A,X,V) :- lay_attr(A,X,V).
/* a logical attribute */
attr(A,X,V) :- log_attr(A,X,V).
/* a presentation style attribute */
attr(A,X,V) :- sty_attr(A,X,V).
/* a layout directive */
attr(A,X,V) :- dir_attr(A,X,V).
```

and so on. To indicate that the presentation syle of the generic letter is the generic letter syle we would have:

```
sty_attr(presentation_style,X,Y) :-
  letter_grammar(X), letter_form_grammar(Y).
```

To translate layout style attributes, we predicate with respect to a singleton rule grammar rather than with respect to a set of rules and write:

```
lay_attr(layout_style,X,justified) :-
   letter_rule_grammar(X).
```

We can proceed in this fashion to account for all of the ODA attributes.

6 A first cut at specifying the layout process

In order to sketch the declarative approach to specifying layout we will appeal to a considerably simpler example than the ODA specimen letter. We consider a simple document whose logical structure consists of a REPetition of paragraphs of text and whose layout structure is a REPetition of pages, which in turn contain REPetitions of layout blocks. We first present the relevant logical grammar:

```
doc_grammar([DG]) --> doc_rule_grammar([DG]).
doc_rule_grammar([(doc --> Rulebody)]) -->
  [(doc --> Rulebody)], {rep(Rulebody, para)}
```

and layout grammar:

```
doc_form_grammar(DFG) -->
  doc_form_rule_grammar([DFR]) +
  page_form_grammar(PFG),
  {dunion([DFR],PFG,DFG)}.

doc_form_rule_grammar([(doc --> DocBody)]) -->
  [(doc --> DocBody)], {rep(DocBody,page)}.

page_form_grammar([PFG]) -->
  page_form_rule([PFG]).

page_form_rule_grammar([(page --> PageBody)]) -->
  [(page --> PageBody), {rep(PageBody, parablock)}
```

Let us now assume a layout process where

1. all paragraphs are small enough to fit within page boundaries,
2. paragraphs cannot be broken across page,

3. and that a paragraph is assigned to the earliest page on which it fits having processed all prior paragraphs.

A Clausal translation of the generic logical grammar

```
summary_paragraph_grammar([_26],_18,_19) :-
  summary_paragraph_rule_grammar([_26],_18,_19).

summary_paragraph_rule_grammar([(summary_paragraph -->
    [_26],{is_summary_paragraph(_26)})],_18,_19) :-
  m(_18,(summary_paragraph  -->
          [_26],{is_summary_paragraph(_26)}),_19).

summary_grammar(_17,_18,_19) :-
  summary_rule_grammar([_26],_18,_30),
  summary_paragraph_grammar(_31,_30,_19),
  dunion([_26],_31,_17).

summary_rule_grammar([(summary-->_26)],_18,_19) :-
  m(_18,(summary-->_26),_19),
  summary_rule_body(_26).

rep(_17,_18) :- is_of_the_form([_17],[_18]).
rep((_25,_26),_18) :-
  rep(_26,_18), is_of_the_form([_25],[_18]).

signature_and_name_grammar(_17,_18,_19) :-
  signature_and_name_rule_grammar([_26],_18,_30),
  signature_grammar(_31,_30,_32),
  name_grammar(_33,_32,_19),
  dunion([_26],_31,_34),
  dunion(_34,_33,_17).

signature_and_name_rule_grammar([(signature_and_name -->
```

```
    signature,name)],_18,_19) :-
  m(_18,(signature_and_name-->signature,name),_19).

append([],_18,_18).
append([_26|_27],_18,[_26|_28]) :- append(_27,_18,_28).

member(_17,[_17|_26]).
member(_17,[_25|_26]) :- member(_17,_26).

repcho(_17,_18) :-
  member(_26,_18), is_of_the_form([_17],[_26]).
repcho((_25,_26),_18) :-
  repcho(_26,_18), member(_27,_18),
  is_of_the_form([_25],[_27]).

m(_17,_18,_19) :-
  append(_30,[_18|_27],_17), append(_30,_27,_19).

dunion(_17,_18,_19) :-
  intersection(_17,_18,[]),
 append(_17,_18,_19).

header_grammar(_17,_18,_19) :-
  header_rule_grammar([_26],_18,_30),
  date_grammar(_31,_30,_32),
  addressee_grammar(_33,_32,_34),
  subject_grammar(_35,_34,_36),
  summary_grammar(_37,_36,_19),
  dunion(_26,_31,_38),
  dunion(_38,_33,_39),
  dunion(_39,_35,_40),
  dunion(_40,_37,_17).

header_rule_grammar([(header -->
    date,addressee,subject,summary)],_18,_19) :-
  m(_18,(header --> date,addressee,subject,summary),_19).
```

```
subset([],_18).
subset(_17,_17).
subset(_17,[_25|_26]) :- subset(_17,_26).
subset([_25|_26],[_25|_27]) :- subset(_26,_27).

ending_grammar([_26],_18,_19) :-
  ending_rule_grammar([_26],_18,_19).

ending_rule_grammar([(ending-->[])],_18,_19) :-
  m(_18,(ending-->[]),_19).

is_of_the_form([],[]).
is_of_the_form([_25|_26],[_27|_28]) :-
  var(_27), !, is_of_the_form(_26,_28).
is_of_the_form([_25|_26],[_27|_28]) :-
  var(_25), !, functor(_27,_29,_32),
  functor(_25,_29,_32), _27=..[_29|_30],
  _25=..[_29|_31], is_of_the_form(_31,_30),
  is_of_the_form(_26,_28).
is_of_the_form([_25|_26],[_27|_28]) :-
  _27=..[_29|_30], _25=..[_29|_31],
  is_of_the_form(_31,_30), is_of_the_form(_26,_28).

supset(_17,_18) :- subset(_18,_17).

figure_grammar(_17,_18,_19) :-
  figure_rule_grammar([_26],_18,_30),
  drawing_grammar(_31,_30,_32),
  caption_grammar(_33,_32,_19),
  dunion([_26],_31,_34),
  dunion(_34,_33,_17).

figure_rule_grammar([(figure-->drawing,caption)],_18,_19) :-
  m(_18,(figure-->drawing,caption),_19).

body_rule_body(_17) :-
  conjappend(_25,(ending,signature_and_name),_17),
```

```
  repcho(_25,[paragraph,figure]).

letter_grammar(_17,_18,_19) :-
  letter_rule_grammar([_26],_18,_30),
  header_grammar(_31,_30,_32), body_grammar(_33,_32,_19),
  dunion([_26],_31,_34), dunion(_34,_33,_17).

letter_rule_grammar([(letter --> header,body)],_18,_19) :-
  m(_18,(letter --> header,body),_19).

nonmember(_17,[]).
nonmember(_17,[_25|_26]) :-
  _17\==_25, nonmember(_17,_26).

addressee_grammar([_26],_18,_19) :-
  addressee_rule_grammar([_26],_18,_19).

addressee_rule_grammar([(addressee -->
    [_26],{is_addressee(_27)})],_18,_19) :-
  m(_18,(addressee --> [_26],{is_addressee(_27)}),_19).

date_grammar([_26],_18,_19) :-
  date_rule_grammar([_26],_18,_19).

date_rule_grammar([(date -->
    [_26],{is_date(_26)})],_18,_19) :-
  m(_18,(date -- >[_26],{is_date(_26)}),_19).

name_grammar([_26],_18,_19) :-
  name_rule_grammar([_26],_18,_19).

paragraph_grammar([_26],_18,_19) :-
  paragraph_rule_grammar([_26],_18,_19).

paragraph_rule_grammar([(paragraph -->
    [_26],{is_paragraph(_26)})],_18,_19) :-
  m(_18,(paragraph --> [_26],{is_paragraph(_26)}),_19).
```

```
name_rule_grammar([(name -- >
    [_26],{is_name(_26)})],_18,_19) :-
  m(_18,(name --> [_26],{is_name(_26)}),_19).


body_grammar(_17,_18,_19) :-
  body_rule_grammar([_26],_18,_30),
  paragraph_and_figure_grammar(_31,_30,_32),
  ending_grammar(_33,_32,_34),
  signature_and_name_grammar(_35,_34,_19),
  dunion(_26,_31,_36), dunion(_36,_33,_37),
  dunion(_37,_35,_17).

body_rule_grammar([(body-->_26)],_18,_19) :-
  m(_18,(body --> _26),_19), body_rule_body(_26).

signature_grammar([_26],_18,_19) :-
 signature_rule_grammar([_26],_18,_19).

signature_rule_grammar([(signature -->
    [_26],{is_signature(_26)})],_18,_19) :-
  m(_18,(signature --> [_26],{is_signature(_26)}),_19).

intersection([],_18,[]).
intersection([_26|_27],_18,_19) :-
  nonmember(_26,_18), intersection(_27,_18,_19).
intersection([_26|_27],_18,[_26|_28]) :-
  member(_26,_18), intersection(_27,_18,_28).

summary_rule_body(_17) :- rep(_17,summary_paragraph).

drawing_grammar([_26],_18,_19) :-
  drawing_rule_grammar([_26],_18,_19).

caption_grammar([_26],_18,_19) :-
  caption_rule_grammar([_26],_18,_19).
```

```
drawing_rule_grammar([(drawing -->
    [_26],{is_drawing(_26)})],_18,_19) :-
  m(_18,(drawing --> [_26],{is_drawing(_26)}),_19).

subject_grammar([_26],_18,_19) :-
 subject_rule_grammar([_26],_18,_19).

caption_rule_grammar([(caption -->
    [_26],{is_caption(_26)})],_18,_19) :-
  m(_18,(caption --> [_26],{is_caption(_26)}),_19).

conjappend(_17,_18,(_17,_18)).
conjappend((_26,_27),_18,(_26,_28)) :-
  conjappend(_27,_18,_28).

subject_rule_grammar([(subject -->
    [_26],{is_subject(_26)})],_18,_19) :-
  m(_18,(subject --> [_26],{is_subject(_26)}),_19).

paragraph_and_figure_grammar(_17,_18,_19) :-
  paragraph_grammar(_17,_18,_19).
paragraph_and_figure_grammar(_17,_18,_19) :-
  figure_grammar(_17,_18,_19).
paragraph_and_figure_grammar(_17,_18,_19) :-
  paragraph_grammar(_29,_18,_30),
  figure_grammar(_31,_30,_19), dunion(_29,_31,_17).
```

B Clausal translation of the generic layout grammar

```
addressee_frame_form_grammar([_26,_27],_18,_19) :-
  addressee_frame_form_rule_grammar([_26],_18,_30),
  addressee_block_form_rule_grammar([_27],_30,_19).
```

```
addressee_frame_form_rule_grammar([(addressee_frame -->
  addressee_block)],_18,_19) :-
  m(_18,(addressee_frame --> addressee_block),_19).

addressee_block_form_rule_grammar([(addressee_block -->
  [_26],{is_addressee_out(_27)})],_18,_19) :-
  m(_18,(addressee_block -->
   [_26],{is_addressee_out(_27)}),_19).

date_frame_form_grammar([_26,_27],_18,_19) :-
  date_frame_form_rule_grammar([_26],_18,_30),
  date_block_form_rule_grammar([_27],_30,_19).

date_frame_form_rule_grammar([(date_frame -->
    date_block)],_18,_19) :-
  m(_18,(date_frame --> date_block),_19).

date_block_form_rule_grammar([(date_block -->
    [_26],{is_date_out(_26)})],_18,_19) :-
  m(_18,(date_block --> [_26],{is_date_out(_26)}),_19).

body_frame_form_grammar([_26],_18,_19) :-
  body_frame_rule_grammar([_26],_18,_19).

logo_frame_form_grammar(_17,_18,_19) :-
  logo_frame_form_rule_grammar([_26],_18,_30),
  logo_block_form_grammar(_31,_30,_19),
  dunion([_26],_31,_17).

logo_block_form_grammar([_26],_18,_19) :-
  logo_block_form_rule_grammar([_26],_18,_19).

logo_frame_form_rule_grammar([(logo_frame -->
    logo_block)],_18,_19) :-
  m(_18,(logo_frame --> logo_block),_19).

logo_block_form_rule_grammar([(logo_block -->
```

```
    [logo_block])],_18,_19) :-
  m(_18,(logo_block --> [logo_block_thing]),_19).

subject_frame_form_grammar([_26,_27],_18,_19) :-
  subject_frame_form_rule_grammar([_26],_18,_30),
  subject_block_form_rule_grammar([_27],_30,_19).

subject_frame_form_rule_grammar([(subject_frame -->
    subject_block)],_18,_19) :-
 m(_18,(subject_frame --> subject_block),_19).

subject_block_form_rule_grammar([(subject_block -->
    [_26],{is_subject_out(_26)})],_18,_19) :-
  m(_18,(subject_block --> [_26],{is_subject_out(_26)}),_19).

header_page_form_grammar([_26],_18,_19) :-
  header_form_rule_grammar([_27],_18,_30),
  logo_frame_form_grammar(_31,_30,_32),
  date_frame_form_grammar(_33,_32,_34),
  addresee_frame_form_grammar(_35,_34,_36),
  subject_frame_form_grammar(_37,_36,_38),
  summary_frame_form_grammar(_39,_38,_19),
  dunion([_27],_31,_40), dunion(_40,_33,_41),
  dunion(_41,_35,_42), dunion(_42,_37,_43),
  dunion(_43,_39,_26).

summary_frame_form_grammar([_26,_27],_18,_19) :-
  summary_frame_form_rule_grammar([_26],_18,_30),
  summary_block_form_rule_grammar([_27],_30,_19).

summary_frame_form_rule_grammar([(summary_frame -->
    summary_block)],_18,_19) :-
  m(_18,(summary_frame --> summary_block),_19).

summary_block_form_rule_grammar([(summary_block -->
    [_26],{is_summary_out(_26)})],_18,_19) :-
  m(_18,(summary_block --> [_26],{is_summary_out(_26)}),_19).
```

```
header_form_rule_grammar([(header_page -->
    logo_frame, date_frame,addressee_frame,
    subject_frame,summary_frame)], _18, _19) :-
  m(_18,(header_page -->
          logo_frame, date_frame, addressee_frame,
          subject_frame, summary_frame), _19).

letter_form_grammar(_17,_18,_19) :-
  letter_form_rule_grammar([_26],_18,_30),
  header_page_form_grammar(_31,_30,_32),
  body_page_form_grammar(_33,_32,_19),
  dunion([_26],_31,_34) dunion(_34,_33,_17).

letter_form_rule_grammar([(letter --> _26)],_18,_19) :-
  m(_18,(letter  --> _26),_19), letter_frule_body(_26).

body_page_form_grammar(_17,_18,_19) :-
  body_page_rule_grammar([_26],_18,_30),
  body_frame_form_grammar(_31,_30,_19),
  dunion([_26],_31,_17).

letter_frule_body(_17) :-
  conjappend(header_page,_25,_17), rep(_25,body_page).
```

C The specific letter grammar represented as a list of rules

```
[(letter --> header, body),

 (header --> date, addressee, subject, summary),

 (date --> [Date], {is_date(Date)}),

 (addressee --> [Addressee], {is_addressee(Addressee)}),
```

```
(subject --> [Subject], {is_subject(Subject)}),

(summary --> summary_paragraph),

(summary_paragraph -->
  [Summary_paragraph],
  {is_summary_paragraph(Summary_paragraph)}),

(body -->
   paragraph, paragraph, figure, paragraph, paragraph,
   ending, signature_and_name),

(paragraph --> [Paragraph], {is_paragraph(Paragraph)}),

(figure --> drawing, caption),

(drawing --> [Drawing], {is_drawing(Drawing)}),

(caption --> [Caption], {is_caption(Caption)}),

(ending --> []),

(signature_and_name --> signature, name),

(signature --> [Signature], {is_signature(Signature)}),

(name --> [Name], {is_name(Name)})].
```

D Various utility predicates

```
subset(X,Y) :- empty(X).
subset(X,Y) :- delete(Z,X,X1), delete(Z,Y,Y1), subset(X1,Y1).

supset(X,Y) :- subset(Y,X).
```

```
diff(X,Y,Y) :- empty(X).
diff(X,Y,Z) :-
  in(U,X), delete(U,X,X1), delete(U,Y,Y1), diff(X1,Y1,Z).

union(X,Y,Y) :- empty(X).
union(X,Y,X) :- empty(Y).
union(X,Y,Z) :-
  subset(X,Z), subset(Y,Z), diff(X,Z,Y1), subset(Y1,Y).

empty(X) :- nonvar(X), X == #empty.  /* #empty is a
                                        distinguished
                                        atom (constant) */

/* m.  m(X,Y,Z) :- X and Z are sets, Y is a member of X,
   and Z = X-{Y}. Used in translating rules containing '+'
   into definite clauses */

m(L,X,M) :- append(L1,[X|L2],L), append(L1,L2,M).

append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).

/* append for conjunctions - helps in keeping conjunctions in
   normal form */

conjappend(X,C,(X,C)).
conjappend((X,C1),C2,(X,C3)) :- conjappend(C1,C2,C3).

/* Repeat and Repeat-Choice */
rep(X,Y) :- is_of_the_form([X],[Y]).
rep((X1,X2),Y) :- rep(X2,Y), is_of_the_form([X1],[Y]).

repcho(X,L) :- member(Y,L), is_of_the_form([X],[Y]).
repcho((X1,X2),L) :-
   repcho(X2,L), member(Y,L), is_of_the_form([X1],[Y]).

/* A Constructor Utility */
```

```
is_of_the_form([],[]).
is_of_the_form([H1|L1],[H2|L2]) :-
  var(H2), !,
  is_of_the_form(L1,L2).
is_of_the_form([H1|L1],[H2|L2]) :-
  var(H1), !, functor(H2,F,N), functor(H1,F,N),
  H2 =.. [F|Args2], H1 =.. [F|Args1],
  is_of_the_form(Args1,Args2),
  is_of_the_form(L1,L2).
is_of_the_form([H1|L1],[H2|L2]) :-
  H2 =.. [F|Args2], H1 =.. [F|Args1],
  is_of_the_form(Args1,Args2),
  is_of_the_form(L1,L2).

member(M,[M|_]).
member(M,[Car|Cdr]) :- member(M,Cdr).

nonmember(X,[]).
nonmember(X,[Y|S]) :- X \== Y, nonmember(X,S).

intersection([],Y,[]).
intersection([A|X],Y,Z) :-
  nonmember(A,Y), intersection(X,Y,Z).
intersection([A|X],Y,[A|Z]) :-
  member(A,Y), intersection(X,Y,Z).

dunion(X,Y,Z) :- intersection(X,Y,[]), append(X,Y,Z).
```

References

[1] Wolfgang Appelt, Richard Carr, and Gernot Richter. The formal specification of the document structures of the oda standard. In J.C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 95–108, Cambridge University Press, 1988.

[2] Dennis S. Arnon, Allen L. Brown, Jr., Jean-Marie de La Beaujardiere,

Martin F.N. Cooper, Patrick Hayes, David M. Levy, William C. Lynch, Sidney W. Marshall, and John M. Stidd. A plan for document interchange practice and research–report of the crg document interchange task force. Xerox Corporate Research Group Internal Memorandum, December 1988.

[3] Richard J. Beach, Dennis S. Arnon, Jean-Marie de La Beaujardiere, Nicholas H. Briggs, David M. Levy, J. Mackinlay, S. Putz, Richard Southall, Larry Spitz, and Polle T. Zellweger. Final draft of the report of the ocm committee on document interchange. Xerox Corporate Research Group Internal Memorandum, October 1987.

[4] Pierre Deransart and Jan Maluszynski. Relating logic programs and attribute grammars. *Journal of Logic Programming*, (2):211–223, 1985.

[5] Nevin Heintze, Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. *The CLP(R) Programmer's Manual.* Department of Computer Science, Monash University, Clayton, Victoria 3168, Australia, June 1987.

[6] Joxan Jaffar and Jean-Louis Lassez. *Constraint Logic Programming.* Department of Computer Science Technical Report, Monash University, Clayton, Victoria 3168, Australia, June 1986.

[7] Joxan Jaffar, Jean-Louis Lassez, and Michael J. Maher. A theory of complete logic programs with equality. *Journal of Logic Programming*, (3):211–223, 1984.

[8] David Maier and David S. Warren. *Computing with Logic: Logic Programming with Prolog.* Benjamin/Cummings, Menlo Park, California, 1988.

[9] Fernando C.N Pereira and David H.D. Warren. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[10] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 1986.

[11] Henri C. Weisz, Ian R. Campbell-Grant, Roy Hunter, Roy Pierce, L.J. Zeckendorf, and Barry 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.