Linear Extended Annotation Graphs

Vincent Barrellon
Univ Lyon, INSA-Lyon, CNRS, LIRIS, UMR5205
Villeurbanne, France F-69621
firstname.lastname@insa-lyon.fr

Pierre-Edouard Portier
Univ Lyon, INSA-Lyon, CNRS, LIRIS, UMR5205
Villeurbanne, France F-69621
firstname.lastname@insa-lyon.fr

Sylvie Calabre/to
Univ Lyon, INSA-Lyon, CNRS, LIRIS, UMR5205
Villeurbanne, France F-69621
firstname.lastname@insa-lyon.fr

Olivier Ferret
Univ Lyon, Lyon 2, CNRS, IHRIM, UMR5317
Lyon, France F-69365
firstname.lastname@univ-lyon2.fr

1 INTRODUCTION
The emergence of Digital Humanities has lead to the development of a great number of digital scholarly publishing projects. Most favour the well-known XML-TEI annotation language for transcription and critical enrichment. Indeed, the TEI provides the scholar with an extremely well-documented schema [10], broad enough to fit almost any kind of primary document, and benefits from the assets of XML languages: it is extensible, can be queried, validated and transformed easily. The XML-TEI thus appears as the go-to technology for the editing scholar today.

Yet, editorial criticisms [26] apart, the TEI-XML language suffers from strong formal limitations, inherent to the XML model. In practice, trees are known not to fit some quite common textual description patterns [10, 20, 23]. In particular, XML does not handle overlapping elements, which is an obstacle towards multi-level [34] annotation; additionally, inclusion being represented by nesting in XML (i.e. the location of an element within the scope of another one), there is no way to represent accidental nesting or co-location, that is, the fact two elements occurring at the same place might be independent (and not included one into the other). Inter-elements relations (other than structural relations) cannot be represented but by attribute equalities (exemplified by the ID/IDREF mechanism), notoriously hard to restrict by means of a schema [3, 31] and possibly hindering querying [14]. Propositions have been made to conform TEI-XML with more expressive data models [6, 8, 10]; while interesting, those propositions are not compliant with the classic XML tools (XSD, XSLT, etc.) [17].

Some alternative ‘multistructured’ data models have been proposed to overcome the expressive limitations of XML, by relying on more general directed acyclic graph formalisms than just trees [31], or even cyclic graphs [16] – while maintaining the possibility to validate the data. Yet, acyclic models, if they do allow multilayer annotation, exhibit the same weakness as XML regarding the representation (and hence, the validation) of non-structural relations between elements; cyclic data models, that rely upon RDF, do not benefit from an efficient validation mechanism yet [29, 30].

In a former paper [2], we introduced extended Annotation Graphs (eAG), a cyclic-graph data model experimenting the simulation relation [19] as a validation mechanism. An interesting aspect of simulation is, as we evidenced, that it can be guaranteed by construction, enabling to validate cyclic, multistructured data on the fly, just like when using grammar-based validators for XML.
We introduce here LeAG, a markup syntax for textual eAG annotations. LeAG takes the shape of a classic, inline markup model. A LeAG annotation can then be written, in a familiar, human-readable form, in any notepad application, and saved as a text file — yet LeAG offers a natural syntax for overlapping, multilayer annotation. First, we provide the reader with a quick, example-based summary of the eAG data model. We then introduce the LeAG syntax based on the eAG model benefits from a schema language (SeAG) that manages multistructured, overlapping, cyclic annotation. SeAG validation relies upon the notion of rooted simulation\(^2\) [2]. Intuitively, the existence of a simulation of an eAG \(I_S\) by a schema \(S\) implies that all the paths of \(I_S\) starting at its root have a corresponding path in \(S\), whose label sequence is identical. Yet, as shown above, the syntactical structures of an eAG are made out of sequences of elements, i.e., of labelled paths. Thus, \(S\) is descriptive of \(I_S\), because any sequence of elements in \(I_S\) must have a matching sequence in \(S\). Conversely, \(S\) works like a schema: it simulates (validates) the graphs that contain solely sequences of elements it defines.

Figure 2 illustrates SeAG validation. The paths the schema \(S\) contains define the set of valid element sequences for the instances (e.g. [Unit1 - Unit2 - Unit3]). SeAG also makes use of epsilon edges, or blank annotations, to denote optional (e.g. Unit3 can be bypassed by the \(\epsilon\) edge, resulting in [Unit1 - Unit3]) or repeatable elements (e.g. since \(\epsilon\) defines a cycle, any repetition of \(\epsilon\)Letters is valid). Importantly, SeAG supports two kinds of multilayering annotation. First, two parallel paths of the schema can be instantiated, independently (i.e. without worrying about overlap), on the same resource (cf. \(I_S\) on fig. 2). We call this schema-based multilayering. Second, one path of the schema can be instantiated several times on the same portion of the resources (cf. \(I_S\), same fig.). This simulation-based multilayering allows the expression of self-overlapping elements, quite useful in linguistics, as illustrated below.

---

1. Represented by dotted, ‘epsilon’ edges’ hereafter.
2. Node-typed [2], rooted simulation actually; yet, node types can be omitted here.
4 LINEAR EXTENDED ANNOTATION GRAPHS

Linear extended Annotation Graphs (LeAG) is an inline markup syntax for eAG. The purpose of LeAG is to enable the expression of eAG annotations by means of any notepad application, in a human-readable form. LeAG must therefore: 1) support unambiguous translation into the eAG syntax, and 2) enable to represent, by means of tags, multilayer, cyclic annotation. The first part of this paragraph is a theoretical discussion about the hybrid nature of the LeAG markup, between the inline and stand-off paradigms, which will lead to the formulation of an equivalence relation for LeAG documents. We then introduce, step by step, the LeAG syntax: we gradually show how to represent the different bricks eAGs are made of in a markup manner: hierarchies, multitees (and godtrees), attributes, links and quotes. We then interrogate the correspondence between eAG and LeAG, in order to establish the parsability of LeAG into eAG.

4.1 Inline multilayer annotation

Multistructured models are meant to support the simultaneous expression of several annotation paradigms. For instance, one may want to annotate a text by identifying, independently, its grammatical (substantive, adjective, etc.) and its semantic (proposition, topic, etc.) structures. To achieve that goal, eAG makes a clear distinction between the representation of inclusion\(^4\), which is a modelling relation that makes sense within one annotation paradigm, and nesting or co-occurrence\(^5\), which is a fortuitous situation in which two independent elements occur at the same place. And indeed, the eAG syntax for inclusion is explicit (see figure 1b), while nesting happens when two elements \(X\) and \(Y\) are so that \(\text{ref}(\text{start}(X)) \leq \text{ref}(\text{start}(Y))\) and \(\text{ref}(\text{end}(Y)) \leq \text{ref}(\text{end}(X))\) – hence nesting is uniquely defined in terms of reference values. Yet the notion of chronology is quite impacted by the shift from stand-off to inline markup. In eAG, in order to fit multimedia annotation, several chronologies can be defined, and each node is associated a value from one of those chronologies. In a text-only markup setting, a natural chronology is implied by the text itself: the set of inter-character positions. As a consequence, LeAG rests upon that single, natural chronology, that does not even need to be made explicit: tags are simply inserted, within the text stream to be annotated, at the position a corresponding node of an eAG would have made reference to. E.g., annotating the substantive in “Let us garlands bring.” is done by inserting a pair of tags as follows: “Let us [Substantive]garlands[Substantive] bring.”

Still, in spite of being considerably simplified compared to eAG, the notion of chronology is still central to LeAG, because it is absolutely necessary in order to represent co-occurrence or nesting. Consider the very elementary text stream ABC. A chronology for this text is: \(\{\text{start()} = \text{before}(A), \text{after}(A) = \text{before}(B), \text{after}(B) = \text{before}(C), \text{after}(C) = \text{end()}\}\). Identifying an element \(\Omega\) between the positions \(\text{before}(A)\) and \(\text{before}(C)\) is done as follows: \(\Omega\)AB\(\Omega\)C. The text stream, since it has been added new characters (the ones that constitute the tags), has been altered by this operation.

\(^4\)E.g. a proposition contains a topic.

\(^5\)A word may happen to be both a substantive and the topic of a proposition: topic and substantive co-occur, substantive is nested in proposition; topic is included in proposition.
As a consequence, contrary to XML, the nesting of an element $B$ inside the scope of an element $A$ cannot be a means to represent the inclusion of $B$ inside $A$. Thus a syntax is needed to represent inclusion (cf. 4.2.1). Second, since inserting tags does not alter the chronology that indexes the original text, tags can be considered not to take “any room” along that chronology. This suggests that inserting exogenous resources within the primary resources, e.g., structured comments, can be done inside special tags that open and close at the same position in the original stream (cf. 4.2.3).

### 4.2 The LeAG syntax

In the following paragraph, based on the above considerations, we gradually introduce the LeAG syntax. The content of the LeAG tags will be defined by means of formulae in which orange characters are constants and italics denotes variables. (Black) parenthesis are mathematical delimiters, not variables or constants. A field is either a variable or a formula enclosed in parenthesis. An optional field is followed by the character ? . A field that can be repeated is followed by + , one that is both optional and can be repeated is followed by * . Concatenation is implicit. Space characters are represented by underscores.

#### 4.2.1 Mono-hierarchy of attribute-less elements

As stated above, some explicit syntax is needed to represent inclusion in a markup model that supports multilayer annotation. This paragraph presents how to express single-layered annotation. The next paragraph extends LeAG towards multilayered annotation.

**Elementary spanning elements.** An elementary spanning element (ESE) is the syntactical structure dedicated to the labelling of a section of the primary resources, with the possibility to assess that the current element is included in other elements of the annotation. ESE are represented by a pair of opening and closing tags whose *substance* field has the same value, according to the following:

\[
\begin{align*}
O\text{Tag} & : [ \text{substance} ] \\
C\text{Tag} & : [ \text{substance} ] \\
\text{substance} & : \text{name fathers} \ (, \ ID) \ \\
\text{fathers} & : _{L_{\text{context}}} \ \\
\end{align*}
\]

Above, *name* is the name of the current element and works as a label on the primary resources enclosed by the pair of tags; *context* provides a designation of the elements that contain the current element. The *ID* field will be discussed in the paragraph 4.2.4.

**The content of an element** is constituted of the tags themselves and the whole text (primary resources + tags) they span over.

#### Rule 4.2.1

The opening and the closing tags defining one ESE cannot belong to the same train of tags.

**Back to the example.** In order to identify one anaphoric chain in the extract of *The Village of Ben Suc*, it suffices to define three element names *Extract, AC* and *Exp* for the identification of the extract, the *AC* and its constituting expressions respectively, and to build the following pairs of opening/closing tags:

- [Extract] and [Extract] ;
- [ AC in Extract] and [ AC in Extract], assessing that an AC is included in an extract;
- [Exp in AC] and [Exp in AC], assessing that an expression is included in an AC.

---

\(^7\) See paragraph 4.2.2 for the actual syntax for multilayer annotation.

\(^8\) We will see that an element may have more than one father, thanks to the notion of grafts. See paragraph 4.2.2.
The following LeAG $L_1$ annotates the anaphoric chain regarding the young prisoner accordingly.


4.2.2 Grafts: Multilayer annotation. We now extend the above syntax to multilayer annotation. Multilayer annotation may occur in two distinct situations: first, the schema defines several annotation paradigms; second, one path of the schema is instantiated several times onto the same resources (figure 2).

The challenge is to make sure that in any case, the tags of a multilayer LeAG shall be unambiguously associated with the layer(s) they are part of. When the set of the elements’ names of two co-existing layers do not intersect, assessing to which layer a tag belongs is trivial. At the opposite, simulation-based multilayering, which is prone to self-overlap, will be problematic: in that case, two overlapping elements cannot be discriminated neither on the basis of their name nor by looking at the name of their fathers. Anaphoric chains annotation is a canonical example of such a setting.

For instance, in the excerpt of The Village of Ben Suc, consider the ACs relative to the American observer and the beating respectively. A naïve approach making use of the syntax for single-layered annotation would yield the following annotation – which is faulty:


Indeed, it is undecidable whether the Exp element starting at the tag 4 ends at tag 5 or 7. Moreover, there would be no way to ascertain to which AC an Exp ranging from tag 4 to tag 5 would belong to.

An intuitive disambiguating solution – at least to the human eye – consists in colouring the tags belonging to distinct layers:


Now it is clear that the element starting at tag 2 ends at tag 5, overlapping with the element starting at tag 4 and ending at tag 7. Importantly, not only have we coloured differently the elements (2-5) and (4-7) in order to make their respective opening and closing tags match, but also have we given a common colour to the elements (3-10), (4-7) and (8-9), which indicates that the two expressions (4-7) and (8-9) belong to the same AC (3-10), for instance.

Grafts. The notion of grafts follows the above intuition. Grafts are coloured LeAGs that are anchored onto an existing LeAG. They express, either locally or at the scale of the whole document, some additional enrichment on top of the annotation that has, at a certain point in time, been done already.

Consider the LeAG $L_1$ at the end of paragraph 4.2.1. $L_1$ identifies one AC and its constituting expressions (Exp), within an extract. A graft must be defined in order to identify, in the same extract, another AC, e.g. the AC regarding the beating, since this addition will result in a non-hierarchical LeAG. This is done as follows:

1. The element of the existing annotation that will serve as the context of the graft is identified: Extract, here.
2. A name of ‘colour’, nameC, is defined, in the form:

   $\text{nameC := \# colour}$

   where colour is a string that identifies the graft, e.g. “#Red”.
3. The range of the graft is specified by inserting, within the frame of the context element, a pair of colour tags:

   $\text{Otag := [ nameC in context >}$
   $\text{Ctag := < nameC in context ]}$

   with the nameC and context fields as defined above. For instance, the span of the AC regarding the beating is the following:

   [Extract] An American observer who saw [Exp over Extract] the beating that happened then reported that the officer "really worked [Exp in AC] him[Exp in AC] over". After the beating[Exp over Extract], [...] [Extract]

4. Then nameC serves as a context for the top elements of the graft. Here, one AC element spans over the whole graft:

   [Extract] [...] An American observer who saw [Exp over Extract] the beating that happened then reported that the officer "really worked [Exp in AC] him[Exp in AC] over". After the beating[Exp in AC] the prisoner [...] [Extract]

5. Elements included in the top elements of the graft are defined, their context field keeping record of the colour of the upper element. For instance, here, two Exp belong to the red AC:


   Similarly, had one Exp element had any child, the context field of the tags defining that element would have been Exp#Red.

Based on that principle, the LeAG $L_2$ on figure 5 identifies the three anaphoric chains regarding the prisoner, the beating and the American observer respectively – which is a case of simulation-based multilayer annotation with self-overlap.
Complements. (1) Grafts are added on top of an existing annotation spanning over the whole document. Before the first graft is defined, the annotation has to be hierarchical. Thus we can refer to this underlying hierarchical annotation as the uncoloured hierarchy of a LeAG. Tags of this hierarchy either have no explicit colour or, when they also belong to a coloured graft, the colour of that graft plus a 'blank' colour, # – see element α in figure 4. (2) A graft may be defined either on the underlying hierarchy (figure 4) or on an element from another graft. (3) An element may have several fathers, belonging to grafts or to the uncoloured hierarchy indifferently (cf. element B, figure 4). (4) The span of the graft shall not necessarily equal the one of its element (figure 5).

4.2.3 Standard inserts: Attributes; structured comment. So far, we have seen how to label the primary resources by means of entangled hierarchies of elementary spanning elements. Still, editing is not only about labelling: sometimes, additional, structured information must be added on top of the labels. In XML, this kind of information constitutes elements’ attributes; still, adding attributes to an element is like annotating the element itself, that is, for the editor, inserting secondary, structured data that does not bear on the primary resources but on the tags. Similarly, providing the editor with means to express critical information, not by labelling the primary resources, but by inserting assertions is a useful feature. Introductions, comments and punctual notes, in their digital form, fall into that category of annotations. Attributes and punctual comments share the property of not being expressible with elementary spanning elements. In LeAG, both will be represented by means of inserts. An insert is similar to void elements in XML in that: (1) it is both opening and closing, which means, in the LeAG vocabulary, that inserts start and end at the same position; (2) it is self-contained, in the sense that the tag representing the insert is the insert’s content.

Attribute insert: general syntax. The syntax of an attribute insert respects the following formula:

\[ \text{Insert} \triangleq [ \textit{Att.of.context} \ldots \text{LeAG} ] \]

where context is the coloured name of the element whose attributes are described in the insert, and LeAG is some structured data conform to the LeAG model, constituting the content of the attributes.

This is not a tough constraint, since a single element spanning over the whole corpus is an elementary hierarchical annotation.

Attribute insert: example. So far, the passage of *The Village of Ben Suc* as a whole was simply labelled as an Extract. The following LeAG provides, as attributes of the Extract, the author’s name, the title and the publication year of the novel:

\[ \text{[Extract]} [\text{Att of Extract} ; \text{[author]} \text{Jonathan Schell} ; \text{[title]} \text{The Village of Ben Suc} ; \text{[year]} \text{1967} ] \]

An ARVN officer, [...] for hours. [Extract]

![Figure 5: Three-layered LeAG.](image)

In the LeAG, there is no need neither to specify the :Att suffix for the elements defined inside the attribute insert, nor to indicate in the context field of the top elements among them, that they are included in the insert. The same applies to comment inserts.

Comment inserts: general syntax. The general syntax of a comment insert is the following:

\[ \text{Insert}_c \triangleq [ \text{name} : \text{Att.in.context} (\ldots \text{ID}) \ldots \text{LeAG} ] \]

where name is the name of the insert, context is the coloured name of the elements the insert is the son of and LeAG structured data conform to the LeAG model, constituting the content of the comment. The ID field will be discussed in the paragraph 4.2.4.

Comment insert: an example. The following LeAG incorporates a comment regarding the context of *The Village of Ben Suc*:

\[ \text{[Extract]} \text{An ARVN[Comment:Att in Extract ; \text{[Att of Comment ; \text{[authorOfComment]} \text{Barrellon et al.} \text{[authorOfComment]} \text{The mention of the [acronym] ARVN[acronym] refers to the Vietnam War.]} \text{officer asked [...] for hours.}] Extract]} \]

A comment being an element, it may possess attributes, as illustrated above (e.g. to specify the name of its authors).

Inserts in a train of tags. The case of inserts within a train of tags has to be discussed. Consider the LeAG \[ A \ldots [B \text{ in A} ; \text{LeAG} ] \ldots ] A.\] In the absence of a schema, it is not possible to assess to which A element B belongs. If there is a schema that does not restrict the position of the element B either at the beginning or at the end of the element A, neither.

Second, consider \[ A \ldots [B \text{ in A} ; L_1 \ldots [C \text{ in A} ; L_2 ] \ldots A.\] The LeAG itself is ambiguous: it states that the inserts \( B \) and \( C \) occur at the same position. Yet, in the perspective of parsing the LeAG into an eAG (cf. paragraph 6), since in the corresponding eAG, two inserts will form a sequence, there is no indication in the LeAG about which insert will come first. The following conventional rule clarifies those situations:

**Rule 4.2.3** When there is no schema or when the schema does not clarify the following situations, it shall be considered that (1) when an insert occurs in a train of tags where an opening and a closing tags identically match the context field of the insert, then the insert conventionally belongs to the opening element; (2) when two inserts with the same context field occur in the same train of tags, the alphabetical order between the tags considered as strings provides a conventional order between the inserts.

\[10\text{Id est, there is no need to write } \text{[author} \text{ in Att of Extract], for instance.}\]

\[11\text{A refinement of the following syntax will be proposed in the paragraph 4.2.4.}\]
4.2.4 Links and Quoting Elements. The last aspect of eAGs that needs to be translated into LeAG is links or quotes. We have seen that in eAG, links and quotes are expressed harmoniously with the other elements (i.e., by means of nodes and edges) and, for that reason, can be properly validated. In particular, compared to XML where a link is but an ID/IDREF pair, in SeAG/eAG, the nature of the two elements connected by a link can inherently be restricted. Still, since links and quotes denote distance connections across the corpus that may result in cyclic annotations (i.e., along the text stream, the beginning of an element comes after its end), it is not possible to represent them by means of pairs tags along the text stream. Thus, LeAG makes use of an additional feature: the ID field. ID fields work as an identifier of either the source or the end of a connection (link/quote), hence enabling to position the extreme nodes of such elements inside the LeAG, that is, to position the elements themselves. Yet, ID fields are not tag identifiers. Indeed, regardless of the parsing strategy adopted, there is no one-to-one correspondence between the tags of a LeAG and the nodes of an eAG expressing the same annotation, as evidenced below:12

\[
\text{ID} := \text{ID}_{=1} \rightarrow \text{K} \quad \text{(singleton syntax)}
\]

\[
\text{ID} := \text{ID}_{=1} \rightarrow \text{M} \rightarrow \text{N} \quad \text{(pair syntax)}
\]

Indeed, because the element \( A \) contains other elements, the tag \([A]\) translates into two nodes whose reference values point towards the position of \([A]\) inside the document, connected by an edge \( A:In \), while the tag \([B]\) relates to one node only. Conversely, two tags may relate to the same node: since the element \( C \) starts where \( B \) ends, both \([B]\) and \([C]\) refer to the node that separates \( B \) and \( C \). Yet, a finer correspondence between the LeAG tags and a subset of the nodes of the corresponding eAG can be exploited for expressing links and quotes: (1) an opening tag positions (and hence, matches) the root of the corresponding element in the eAG; (2) a closing tag positions the leaf of the corresponding element in the eAG; (3) an insert positions both the root and the leaf of the corresponding element in the eAG. ID fields exploit that connection, as follows.

**ID Fields.** Since opening and closing tags of ESE relate to either the root or the leaf of an element in the corresponding eAG, ESE ID fields contain a singleton value \( K \). *A contrario*, an insert ID shall possibly designate the root and the leaf of the corresponding eAG element and thus contains a pair of values \( M \) and \( N \):

\[
\text{ID} := \text{ID}_{=1} \rightarrow \text{K} \quad \text{(singleton syntax)}
\]

\[
\text{ID} := \text{ID}_{=1} \rightarrow \text{M} \rightarrow \text{N} \quad \text{(pair syntax)}
\]

**Basic example.** Let us consider the following comment and a matching eAG (pink flags represent the node identifiers):

```
[Comment: Att, ID = 1 \rightarrow 2; (title) Sasebo yuki (title, ID = 3)
 [author, ID = 3] Tanizaki Junichiro (author, ID = 4)]
```

Noteworthily, the relation between ID values and root/leaf nodes is only surjective. Thus, the ID of the closing tag of an element and that of an element that immediately follows have to be the equal (e.g. \([\text{title}, ID = 3]\) and \([\text{author}, ID = 3]\), above).

The two (orange) edges permit to structurally include the quoted element inside the comment element. In LeAG, since the content of a comment has to be written inside the insert itself, quoting, inside the LeAG field of a comment, an element that has been identified elsewhere in the annotation cannot be done but by reference. Therefore, quoting elements appear as special comment inserts, whose LeAG field has been replaced by an ID field (with the pair syntax):

```
[Quote := [name, in, context, ID]?(..ID1)?..ID2 ]
```

The pair of values of the \( ID_2 \) field must then refer to some tag(s) somewhere else in the LeAG that delimit either an element or a sequence of elements.

**Quote example** The LEAG representing the above eAG is:

```
[Extract][Comment: Att in Extract; The mention of the Army of the Republic of Vietnam ([Quote; ID = 1 \rightarrow 2]) refers to the Vietnam War.] An [Acronym in Extract, ID = 1] ARVN (Acronym in Extract, ID = 2) officer [...] hours. [Extract]
```

This LeAG does correspond to the eAG above, since it states that the Extract contains a Comment:Att, made out of some not annotated text (which translates into an epsilon edge), followed by a Quote containing an annotation graph whose root and leaf have the identifiers ‘1’ and ‘2’ respectively; Extract further contains an Acronym, whose root and leaf identifiers are ‘1’ and ‘2’ respectively.

**Links.** An eAG link is an element whose root is a node from an element and whose leaf is a node from another element. First, to represent such a graph in LeAG, we need to be able to identify a node inside any element. Consider the link in figure 3. It connects the internal nodes of two AC:Att elements that contain nothing but those nodes. Yet, the ID fields of an insert with no LeAG field, suit to represent those AC:Att elements, only identifies the root and leaf of the matching element, not an internal node. To fill this gap, we define void inserts:

```
VoidInsert := [in, context, ..ID ]
```

\[
\text{ID} := \text{ID}_{=1} \rightarrow \text{N}
\]

Such an insert neither has a name nor a LeAG field, but it does have a context (the element it is included in) and an ID field. Placed immediately after an opening tag, e.g. \([A]\), a void insert \([in \ A, ID = 1]\) enables to give the identifier ‘1’ to a node that, in the corresponding eAG, is the node ending the A:In edge.
Second, we need a means to express that an element may start inside an element and end inside another one. For such a special element, we defined link insert:

\[
\text{Link} := [ \text{name : LinkTo, in, context } , \text{ID( } \ldots \text{, LeAG)? } ]
\]

\[
\text{ID} := \text{ID} = M \Rightarrow N \text{ OR ID} = N \Rightarrow N
\]

The LeAG field defines the content of the link; if empty, the link is an edge. The leaf of the link, identified by the value of the variable \( N \) above, must be an internal node of some element, represented elsewhere by a void insert.

**Link example.** Figure 3 illustrates how to annotate different, overlapping AC in an extract, and how links could reify an order relation between them. The following LeAG expresses the same annotation, extended to three ACs (the prisoner, the beating, the American observer) as in the eAG on figure 3:

\[
\begin{aligned}
\{ \text{Extract} \} &\text{An ARVN officer asked AC in Extract}\{ \text{AC:Att of AC} ; [\text{Longer:LinkTo, ID} = \Rightarrow 2]\text{[Longer:LinkTo, ID} = \Rightarrow 1]\} \text{[Exp in AC] a young prisoner[Exp in AC] questions, and when [Exp in AC] the[Exp in AC] failed to answer, the[Exp in AC] in [AC:Att of AC\#Blue] ; [in AC:Att\#Blue, ID} = 1]\} \text{[Exp in AC\#Blue] \text{An American observer who saw [\#Red over Extract] [\#Red over Extract]} [\text{AC:Att of AC\#Red} ; [\text{in AC:Att\#Red, ID} = 2]\text{[Longer:LinkTo, ID} = \Rightarrow 1]\} \text{[Exp in AC\#Red] the beating[\#Blue over Extract]} [\text{Exp in AC\#Blue}] \text{[in AC\#Blue] that happened then[Exp in AC\#Red] reported that the officer 'really worked [Exp in AC] in AC\#Red'] over Extract]. [Exp in AC] the prisoner[Exp in AC] in Extract] was forced to remain standing for hours. [Extract]
\end{aligned}
\]

### 4.3 Parsing LeAG

Let us consider that two eAG are isomorphic iff there is a bijective morphism \( \phi \) between them so that a node and its image by \( \phi \) share the same reference value. LeAG is designed as a markup syntax for eAG. Ideally, there should have been a bijection between LeAGs and the classes of isomorphic eAGs. Yet, this is not the case: first, because two equivalent LeAG documents shall translate into the same eAG, and second, because several non-isomorphic eAGs could match a given LeAG – which is clearly problematic when considering parsing LeAG documents into eAG. For instance, the elementary LeAG \([ A, \ldots [ A] A, \ldots [ A] \) may reasonably translate into either of the following:

\[ [1] \begin{array}{c}
    A
    \end{array}
\]

\[ [2] \begin{array}{c}
    A
    \end{array}
\]

or any eAG made out of a sequence of two edges labelled \( A \) with the right reference values, separated by any number of epsilon edges.

The problem is we cannot, in the absolute, prefer one eAG over the others, since all of them do represent the fact the LeAG document contains two \( A \) elements in a row – and also, and most importantly, because the different eAG will not be validated against the same schemas. Indeed, considering the three SeAGs below, the above eAG \([1]\) is validated by the schema \([S_1]\) only, \([2]\) by both \([S_2]\) and \([S_3]\), and \([3]\) by \([S_3]\) only.

Choosing one solution against the others thus cannot be done but by considering a predefined schema. Hence parsing a LeAG means: given a SeAG, yielding a valid eAG that ’represents well’ the LeAG – if such an eAG exists.

In the following, we discuss how to design a deterministic schema-aware LeAG parser. First, we propose to restrict SeAGs to non-ambiguous (N-A) ones \([5]\), ambiguous schemas resulting in non-determinism. We then show that associating to the initial LeAG an eAG, validated by a N-A SeAG, containing the same sequences of elements, whose elements’ span is the same, and whose inclusion relations are the same as in the LeAG, is deterministic in general – but not in some cases. We identify those cases and we show that there is a notion of a minimal eAG, that enables to deterministically single out one valid eAG among the others (up to isomorphism)\(^{13}\).

**Non-ambiguous SeAG.** A SeAG is non-ambiguous (N-A) iff given any label sequence \( w \), there is at most one path connecting the root of the SeAG to its leaf that, epsilon edges set apart, spells \( w \) \([5]\). Non-N-A SeAG will result in non-deterministic parsing. See the following schema, that matches the LeAG \([ A, \ldots [ A], A, \ldots [ A] \) in two different ways – see graphs \([1]\) and \([2]\) below:

\[ [S_3] \begin{array}{c}
    A
    \end{array}
\]

\[ [1] \begin{array}{c}
    A
    \end{array}
\]

\[ [2] \begin{array}{c}
    A
    \end{array}
\]

#### LeAG-eAG label sequences.

First, we want to stress the fact the sequence of the tags in a LeAG implies the sequences of labels along the different paths the corresponding eAG is made out of, provided the eAG is so that each element of the LeAG has one and only one corresponding element in the eAG.

First let us consider hierarchical LeAGs. The rule 4.2.3 implies there is only one way to read the tags from a given train of tags, regardless of the order in which they are written: this ensures that each of the set of tags of the same colour, in a LeAG, defines one and only one hierarchy of elements. Since a given hierarchy of elements translates into one and only one sequence of labels (epsilon edges set apart) in the eAG model\(^{14}\), a hierarchical LeAG can be associated only one label sequence in the eAG model – epsilon edges set apart. Now given grafts are hierarchies of elements that are included in a given element of the LeAG\(^{15}\) and given links and quotes are also hierarchical structures whose connection with the rest of the eAG is determined by the identifier of their root and leaf; the previous discussion extends to LeAGs in general: the sequence of the labels along the paths of the eAG representing a graft, link, quote, is deterministically implied by the LeAG.

**EAG equivalence.** Two non-isomorphic eAGs are equivalent iff their elements form bijective pairs, so that: 1) the elements from each pair share the same name and 2) their previous, following and father and son elements, if they exist, form pairs, and 3) so that the reference values and identifiers at their root/leaf are identical.

**Finding a valid eAG.** Hierarchies form paths in an eAG. The label sequence of each hierarchical structure of an eAG matching a given LeAG is, as shown above, uniquely defined by the LeAG.

\(^{13}\)The whole discussion that follows is ‘up to isomorphism’.

\(^{14}\)Eg. ‘A contains X’ translates into the label sequence: \([X/n / A / X/Out]\).

\(^{15}\)We consider grafts are included in their context element, while an element might belong to both a graft and an outer element: it is always possible to change a graft sharing an element with its context into two grafts strictly included in this context.
Validating a LeAG $L$ is then quite simple. If $L$ is hierarchical, be $L$ the eAG label sequence corresponding to $L$. If there is a path in the schema whose label sequence, $e$ edges apart, spells out $L$, the LeAG is valid and a valid corresponding eAG can be defined. If $L$ contains one graft: be $L$ the eAG label sequence matching the uncoloured hierarchy within $L$, and $L'$ the label sequence for the graft: $L$ is valid if the schema contains a path that spells out $L$ and another path, inside the element in that path that matches the element in which there is a graft, that spells out $L'$. Recursively, this principle applies for grafts on grafts; it can be adapted for links and quotes.

Conveniently, in a N-A SeAG, two paths cannot spell the same label sequence, so if there is a path in a schema, say, that matches a hierarchical label sequence, then it is unique. Identically, if, in the context of an element, there is a path that matches the label sequence of a graft, it is also unique. If there are more than one valid eAG, then we know they still all correspond to the same paths in the schema. This means that some schemas are non-deterministic, i.e. that they validate several non-isomorphic, equivalent eAGs.

N-A Schemas validating several equivalent eAGs. We provide the following result: non deterministic N-A SeAGs are the SeAGs that contain a sequence of at least two epsilon edges.

A sequence of (two) epsilon edges may happen in two different patterns: the edges either form a cycle or not. The first, general problem with a sequence of two epsilon edges is, since epsilon edges are not represented in LeAG, that the reference values of the nodes of the sequence that do not work as the root or the leaf of an element are undetermined. For instance, the reference of the orange node in $I^{S_a}_w$ on figure 6 can be given any value between $r_1$ and $r_2$. Hence, several eAGs, differing only by one reference value, can be associated to the left LeAG on that figure.

Additionally, non-determinism may result in structurally different graphs. Cyclic sequences of two epsilon edges may result in a valid, hierarchical eAG, in any even sequence of epsilon edges.

Then, single-layered LeAG annotations, that translate into single-pathed eAGs, will be associated an infinity of equivalent, valid eAGs: see $I^{S_a}_w$ in figure 6. N-A schemas containing only linear sequences of two epsilon edges will, on the contrary, be deterministic for the parsing of hierarchical LeAGs, but not in case of simulation-based multilayering (figure 6, right). Cyclic epsilon sequences will also be problematic in case of simulation-based multilayering.

One can check that the cases of non-determinism above necessitate a sequence of no less than two epsilon edges: the schemas that contain no or isolate epsilon edges are deterministic.

Minimal eAG. In order to make LeAG parsing with N-A schema deterministic, that is, to ensure that an equivalent class of LeAG documents be associated only isomorphic eAGs, we must single out one of the equivalent eAGs as a unique parsing solution.

Let $X_S$ be an equivalent class of eAGs validated by the same schema $S$. For $G \in X_S$, let us denote $|V_G|$ and $|E_G|$ the cardinal of its sets of nodes and edges respectively. We claim that, up to isomorphism, there is only one eAG $M \in X_S$ that minimises both the values of $size = |V_M| \times |E_M|$ and $Sref = \sum_{v \in V_M} ref(v)$. Then let $M$ be the right representative of that eAG class for parsing under $S$.

LeAG parsability. For any class of LeAG documents, there is at most one minimal eAG validated by a given N-A SeAG schema, that preserves the LeAG elements, elements’ span, identifiers and inclusion relations.

5 RELATED WORK

Many schema-aware data models have been specifically proposed to overcome the limitations of XML by enabling, at the very least, the expression of not only one, but several hierarchies onto the same resources. Among the most notable such ‘multistructured’ data models, we may mention MulaX [13], XConcur [24], MSXD [7], Rabbit/Duck grammars [27]. Those are all built upon XML and make use of the same fundamental notions like elements, attributes, inclusion, etc. The general validation strategy for those models is to extract or isolate the different hierarchies of elements present in the documents, and validate each separately; they also investigate inter-hierarchy constraints.

LMNL [32] represents a more stripped-down vision of multilayer annotation. In many respects, LeAG borrows from LMNL. In LMNL, the user can identify ranges in a character stream and name them by means of pairs of opening and closing tags. Ranges themselves can be annotated by (meta)ranges, which inspired the attribute syntax in LeAG. Yet LMNL claims to be an annotation language solely, and not a structuring language: in particular, LMNL does not provide the user with means to represent inclusion or sibling relations. As we have seen in paragraph 4.1, indeed, inclusion is either represented by nesting, which limits the data model to trees, or by means of an explicit syntax; LMNL does not propose such syntax, and yet allows overlap and multilayering. By sweeping out the notion of inclusion, LMNL seemingly clears the paradox out; yet LMNL is not absolutely blind to the charms of hierarchies: it lies upon the notion of ‘layers’, that is, ranges that fully contain the ranges that start and end in their scope, which is reminiscent of XML hierarchies – but if such patterns cannot be interpreted in structural terms, can they be but fortuitous patterns? Still, because hierarchies are a classic and fundamental annotation structure [34], the LMNL model comes along with XML generators that can extract hierarchies from the data. Our point, on that matter, is that since hierarchies are so central, the best is to enable the editors to have direct control over their expression – which indeed demands additional syntax. Apart from those critical considerations, LMNL...
is an important annotation model, that goes beyond most others, in terms of expressivity; moreover, it benefits from a grammar-based validation language [31], able to embrace the multilayer documents as a whole, which can be compared only to RDF validators (or to the SeAG we propose [2]).

Indeed, several annotation models have originated from the RDF community. One may think of the pioneering RDFTef [33], the Open Annotation data model [22] or EARMARK [16]. The RDF data model, which imposes no restriction on the shape of the resulting graph, is very expressive; moreover, RDF annotation can be used as a complement to an existing TEI annotation [1], which is a way to ally the best of two worlds. One limitation though of RDF-based annotation languages is the current lack of a proper and computationally efficient validation mechanism. OWL is not natively suitable for validation [21, 25]; tweaks aiming at the expression of, say, integrity constraints, have been experimented, but results in huge execution time [30]. Nonetheless, RDF validation is a promising field of research, as illustrated by the SHEx [18] and SHAACL [15] projects. Time complexity still seems to be quite high, but cutting it down is being investigated [29].

6 CONCLUSION

In this paper, we introduced LeAG, an inline markup syntax for extended Annotation Graphs. eAG is a stand-off annotation model, based upon a general, cyclic graph formalism: as one may expect, the model is thus highly expressive, fit to express multilayer annotation, but also distant connections across the annotation. The LeAG syntax illustrates that a similarly expressive inline markup model can be defined, at least for textual annotation. To achieve that goal, we defined a limited number of necessary syntactical structures (grafts and inserts). This way, the LeAG syntax is kept as simple as possible, while opening wide prospects in terms of editorial enrichments.

We also established the parsability of LeAG into eAG, that is, the fact a LeAG document translates into one corresponding eAG. This aspect of the LeAG syntax is crucial, since it indirectly provides the LeAG documents with a validation language, that is, SeAG, that comes as the validation mechanism for eAG. SeAG does not rely upon the notion of grammars, like most validation languages do, but on the simulation relation. From a technical point of view, simulation-based validation shines by enabling the validation of cyclic data, while keeping the algorithmic costs low. Moreover, as illustrated throughout this article, the ability of SeAG schemas to validate anaphoric chain annotations, which is a canonical linguistic annotation, also evidences the editorial relevance of that particular kind of validation for annotation purposes.

REFERENCES


[29] Jiao Tao, Evren Sirin, Jie Bao, and Deborah L McGuinness. 2010. Integrity Constraints in OWL. In AALI.


