Towards markup support for full GODDAGs and beyond: the EARMARK approach

Silvio Peroni[1], Fabio Vitali[1], Angelo Di Iorio[1]

[1] Department of Computer Science, University of Bologna
speroni@cs.unibo.it, fabio@cs.unibo.it, diiorio@cs.unibo.it

Abstract. One of the most evident tenets of the literature on overlapping markup is that the philosophy of documents as trees (as dictated by meta-markup languages such as SGML and XML) is a simplification that sometimes fails and requires corrections. These corrections have been proposed at the markup level (e.g., milestones, segmentation), at the meta-markup level (e.g., LMNL, TexMecs, XCONCUR, etc.) or at level of the abstract model (e.g., GODDAG). Unfortunately full GODDAGs do not allow linearizations in general, and as such a restricted version of GODDAG, r-GODDAG, has been proposed that is guaranteed to be linearizable (in TexMecs) and still allows many nice features beyond trees.

In this paper we discuss that the problem of linearizing more-than-hierarchical structures lies basically in the embedding of markup within content and that no such problem arises with an appropriate standoff approach, that is able to represent full GODDAGs without restrictions. This gives ample opportunities to deal with interesting markup features that are describable with GODDAGs but not with r-GODDAGs, such as non-contiguous elements and virtual elements.

Besides, we discuss whether a specific constraint of full GODDAGs is really necessary once all residual hopes of embeddability are given up, and we further propose a minimal extension to GODDAG, genially called "extended GODDAG" (e-GODDAG) that, by removing the requirement for names in non-terminal nodes, adds support for additional interesting markup features such as content repetitions. In truth, e-GODDAGs are even less embeddable than full GODDAGs, but they are just as easily dealt with by using stand-off markup.

We further propose a meta-syntax for non-embedded markup, called EARMARK, that can be used for stand-off annotations of textual content, and that naturally represents e-GODDAGs with fully W3C-compliant technologies. EARMARK is based on an ontologically precise definition of markup that instantiates the markup of a text document as an OWL document, and through appropriate OWL and SWRL characterizations it can define structures such as trees, r-GODDAGs, full GODDAGs and e-GODDAGs, and can be used to generate validity constraints (including co-constraints), and to verify adherence to content model patterns.

As mentioned, in general the embedding of a full EARMARK document is not straightforward, but approaches can be taken in that direction: just like segmentation and fragmentation are strategies to embed in a strictly-hierarchical language a r-GODDAG-specific feature such as overlapping elements, similarly a number of strategies exist to provide embedding of GODDAG and e-GODDAG features in less expressive syntaxes. In the final part of the paper we discuss our wish to provide at the metalanguage level a series of embedding strategies of the

non-hierarchical features of EARMARK, i.e. a number of language-independent mechanisms to express e-GODDAGs structures into XML (as well as in TexMecs and in LMNL) and that can be recognized as such (i.e., as strategies, as tricks) by tools and readers alike, especially for further uses of such documents.

Keywords: EARMARK, GODDAG, Markup, OWL, Overlapping Markup, RDF, RDFa, e-GODDAG

1 Introduction

Not everybody working with markup languages needs support for overlaps and multiple hierarchies. But those that need it, usually need it badly. So badly, in fact, that a robust slice of markup literature is devoted to it, specialized scientific events have taken place, and a number of extremely varied approaches have been proposed in the last years for this issue.

Some of these approaches were proposed at the language level: many XML languages (TEI [17] being the most evident) took the decision to add specific language-dependent markup structures devoted to supporting overlapping. Some of such solutions, such as milestones and fragmentation, are so general and widely applicable that have been proposed even outside of the specificity of just one markup language (e.g., see [6]), as architectural forms available in general in XML languages. Further approaches have been to create new meta-markup languages, inspired by but independent from XML such as TexMecs, LMNL, XCONCUR, that provide at the metalanguage itself support for more than a single hierarchy. This makes it possible for any language defined within the syntax to make use of the features for overlaps, without the need for further special tools to make sense of the annotations.

At the conceptual level, what has shown its limits is the idea of forcing tree-like structures over documents. While some may be fully described by trees, some just are not, and we need more powerful abstract data structures to describe them. GODDAGs [19] have been proposed exactly for this purpose: direct acyclic ddgraphs with ordered children relax exactly the kind of constraints of trees that were in the way for sophisticated markup features such as overlap. Unfortunately, generalized GODDAGs do not allow an immediate linearization in form of an XML-like syntax (even if extended in some way), but a variant, restricted GODDAGs ([19] and then [12]), does allow a linearization in TexMECS. In general, though, the linearization of full generalized GODDAGs does not allow to keep all information expressed in the original graph. Many additional useful features of GODDAGs (e.g., virtual elements) can only be converted in XML structures by recurring to procedural tricks.

In this paper we discuss whether the problem of generalized GODDAGs lies in the embedding nature of meta-markup languages such as XML, TexMecs and LMNL, and whether by getting rid of embeddability altogether we can exploit the full potentiality of GODDAGs. Furthermore, we propose a minimal extension to GODDAGs to provide full support of repeated content in GODDAGs (currently only allowed if appearing in different substructures), thereby generalizing the idea of repeatability of markup structures.

Additionally, in this paper we propose a meta language for GODDAGs and extended GODDAGs that relies 100% on well-known and widely available W3C

technologies: EARMARK (*Extreme Annotational RDF Markup*) is a language for standoff annotations over documents that is based on an OWL ontology and uses RDF annotations as its linearization approach. EARMARK annotations are facts expressed about OWL classes such as ranges and markup elements whose properties are fully and explicitly described in the OWL ontology itself, and minimally depend on syntactic constraint that are inherent of embedded languages. Thus all standard hierarchy assertions as usually expressed in XML are available in EARMARK, but the language also supports overlapping structures, virtual elements, anonymous elements and structured attributes, that are available with more sophisticated non-XML languages such as TexMECS or LMNL, as well as unsupported features such as repeated structures, content variants and partially overlapping multiple hierarchies[1] etc. are trivially expressed in EARMARK, and contribute to generate a language that is suited to fully support generalized GODDAGs and extended GODDAGs.

EARMARK documents are therefore OWL documents that can be expressed as RDF assertions, and using plain and standard W3C technologies a number of Semantic Web tools can be used for generating, converting, querying and displaying EARMARK documents. Particularly relevant here is the process of embedding EARMARK documents in traditional embedded languages, such as XML or TexMecs. Of course, not all EARMARK assertions can be directly transformed into XML markup structures. The specific subset of the EARMARK document that can be expressed in the destination syntax (e.g., any of the possible tree substructures for XML, or of the r-GODDAG substructures for TexMecs) can be directly generated, and the remaining ones need to be either left out or forcedly embedded using any of a number of well-known or newly-introduced syntactic tricks (up to, of course, leaving part of the EARMARK markup directly as RDF fragments within the destination document).

Providing a recognizable and repeatable two-way process for generating EARMARK documents out of embedded documents, and vice versa for generating embedded documents out of EARMARK document with embedding tricks represents also a chance to collect and generalize all such embedding tricks, and providing an additional conversion model between different syntaxes. All in all, we propose EARMARK as the most natural syntactical rendering of GODDAGs (and e-GODDAGs, of course) and as the intermediate representation of any conversion path for documents, XML or otherwise, that use overlapping features of any form, i.e., as a generalization of the conversion algorithms for overlapping structures proposed in [13].

2 Embedding multiple hierarchies

There comes a time, in marking up documents, where different types of annotations need to be placed upon the same content, and different markup needs to be used. Sometimes these different annotations nest easily, and sometimes they do not. Trying to express these different annotations using a hierarchical metamarkup language such as XML is, *per se*, unfeasible: each structure needs to be described by its own hierarchy, and the

---

[1] Defined as the "set of partial or independent overlapping hierarchies in which the textual content between the tags is visible in some hierarchies but not in others" [16].

overlapping situations pose a big problem, since, as we know, XML is not naturally equipped to deal with them.

Some approaches to deal with overlapping structures in markup languages were proposed in past years. Each approach tries mediating between the support for overlapping and the hierarchical organization of XML documents, as illustrated in [6], as well as in [17], [20] and [13]. The five main overlap-handling techniques described in literature can be summarized in the following:

- *milestones*, through which one hierarchy is expressed using the standard hierarchical XML markup and the elements belonging to the other ones are represented through a pair of empty elements representing the start and the end tags, and connected to each other by special attributes.
- *flat milestones*, that represents each of the hierarchy elements as a milestone, i.e., an empty element placed where the start or the end tag should be, all of them contained as children of the same root element.
- *fragmentation*, in which one hierarchy (the primary) is expressed though the standard hierarchical XML markup, and the elements of the secondary hierarchies are fragmented within the primary elements as needed to suit the primary hierarchy and are connected to each other by special attributes.
- *twin documents*, in which each hierarchy is represented by a different document, which contains the same textual content but marks up the elements according to the individual hierarchy.
- *stand-off markup*, which puts all the textual content in a single structure with the possible specification of the shared hierarchy, and puts the remaining elements in other structures (e.g., files) with the positional association of each starting and ending location to the main structure, using, for instance, XPointer [5] locations.

A separate approach is to give up the XML requirements of a single hierarchy, and try new approaches where multiple hierarchies can be specified in the same text flow. The data structure itself, of course, is not a tree anymore, and needs to become something more general. The *General Ordered-Descendant Directed Acyclic Graph* [19], or *GODDAG*, is the most relevant data structure that has been used to specify complex markup hierarchies, such as overlapping between elements and fragmentation.

Although GODDAG is not able to handle directly other features such as anonymous elements and structured attributes, yet another different non-XML approach for these and other well-known overlapping scenarios is given by *Layered Markup anNotation Language* [21], or *LMNL*. Contrarily to GODDAG, that expresses the many hierarchies with a graph, LMNL uses a XML-like syntax where named or anonymous elements can overlap with other ones in one or more element *layers*.

A similar approach is used by XConcur [15]. An XConcur document is made of multiple layers coexisting in the same multi-root structure, written in a XML-like syntax: each layer represents an independent hierarchy that can be extracted as a single unit and validated against a DTD, XML-Schema or RelaxNG schema. Relationships and constraints between multiple hierarchies are ruled by a related constraint language called XConcur-CL. XConcur documents end up being very complex and few tools to manipulate them are available.

3 Could singing songs be such a big deal?

To illustrate some of the difficulties in handling complex structures, let us examine a fictitious karaoke application in which lyrics are displayed on a screen in sync with a recording of the instrumental parts of the corresponding song; in order to make the example even more interesting, let us consider the situation whereby, beside the screenfuls of lyrics, the application would also show the chords of the song for any additional instrument playing along, and a few fun facts popups here and there to keep the attention of the readers.

We will use as an example for our discussion the song "And I love her" by the Beatles, one of the most famous and sung songs of the history of modern music. The lyrics of the original version appear in Table 1.

Table 1. Lyrics and structure of “And I love her” by The Beatles

Title	And I love her
1	I give her all my love / That's all I do / And if you saw my love / You'd love her too
Chorus	I love her
2	She gives me ev'rything / And tenderly / The kiss my lover brings / She brings to me
Chorus	And I love her
3	A love like ours / Could never die / As long as I / Have you near me
4	Bright are the stars that shine / Dark is the sky / I know this love of mine / Will never die
Chorus	And I love her
4	Bright are the stars that shine / Dark is the sky / I know this love of mine / Will never die
Chorus	And I love her

The first difficulty for our karaoke application is to handle more than one structure at the same time. We may be interesting in building multiple structures over the same content:

- the lyrics organized in stanzas and verses
- the notation for the time-driven excerpts of lyrics as shown on screen during the playback
- additional time-driven visualization of the chords, with different time intervals
- the (either time-driven or content-driven) visualization of pop-ups with fun fact sentences

Furthermore, we may need to deal with small difference in lyrics if the gender of the loved one is female, as in Beatles' original (“and I love her”) or male, as in several covers (“and I love him”).

3.1 Dark is the overlapping sky

Let us concentrate on a single stanza of the song, the fourth, and its refrain:

```
Bright are the stars that shine
Dark is the sky
I know this love of mine
Will never die
And I love her
```

The first hierarchy represents the lyrics. We may employ an XHTML vocabulary, using the class attribute for characterizing containers (e.g. “stanza” and “refrain”), obtaining a clear and straightforward structure.

```
<body>
    <div class="stanza" title="4">
        <p>Bright are the stars that shine</p>
        <p>Dark is the sky</p>
        <p>I know this love of mine</p>
        <p>Will never die</p>
    </div>
    <div class="refrain">
        <p>And I love her</p>
    </div>
</body>
```

The harmony of the song uses two chords, Em and Bm, for each of the first three lines, then moving to the bridge G for the last line and then the refrain in A and D. The A chord starts while the melody is still singing the second part of the last line of the stanza. A possible, trivial hierarchy for chords would then be:

```
<chords>
    <Em>Bright are the</Em>
    <Bm>stars that shine</Bm>
    <Em>Dark is the</Em>
    <Bm>sky</Bm>
    <Em>I know this</Em>
    <Bm>love of mine</Bm>
    <G>Will never</G>
    <A>die And I</A>
    <D>love her</D>
</chords>
```

We have a different issue with the timings for the lyrics. We want each line to appear exactly when the music calls for it to be sung, but at the same time we want that the

next line is shown, too, so that the singer gets ready to sing it aftwerward. Thus each line has to appear twice in each screenful, as in the following XML fragment:

```
<timing>
    <screenful starts=″68″>
        <main>Bright are the stars that shine</main>
        <next>Dark is the sky</next>
    </screenful>
    <screenful starts=″72″>
        <main>Dark is the sky</main>
        <next>I know this love of mine</next>
    </screenful>
    <screenful starts=″76″>
        <main>I know this love of mine</main>
        <next>Will never die</next>
    </screenful>
    <screenful starts=″80″>
        <main>Will never die</main>
        <next>And I love her</next>
    </screenful>
    <screenful starts=″84″>
        <main>And I love her</main>
    </screenful>
</timing>
```

Only this is not nice: each line appears twice in the screen, and therefore twice in the XML document, and forcing them to appear only once in the XML structure would either require overlapping, or forcing some structural semantics into procedural attributes, that would imply implementing ad hoc visualization tools, as in:

```
<p main=″68″>Bright are the stars that shine</p>
<p main=″72″ next=″68″>Dark is the sky</p>
<p main=″76″ next=″72″>I know this love of mine</p>
<p main=″80″ next=″76″>Will never die</p>
<p main=″84″ next=″80″>And I love her</p>
```

We do not like this approach and will not consider it further. Yet the multiplicity of the lines is worrisome, as it creates a redundancy that has to be carefully considered.

As for the pop-ups, we want to show some additional text exactly at the right time – i.e. when the song gets to the precise point of the lyrics associated to these paragraphs. It is not even given that pop-ups are aligned with whole lines, indeed it could very well happen that the association transcends line boundaries, as in the following:

```
<funfacts>
    <popup>
        <lyrics>this love of mine Will never die</lyrics>
        <fact>
            <p>
               Paul McCartney wrote this about his girlfriend,
                 an actress named Jane Asher.
```

```
            </p>
        </fact>
    </popup>
</funfacts>
```

As we have seen, each of these structures, taken individually, is a single hierarchy and could be easily managed with a traditional XML document. There is a final issue related to text variants: depending on the preferences of the singer, we may want to decline the lyrics in the masculine or feminine gender. As such, we end up with two variants of the refrain, and no syntactically evident way to point out which variant to show and which to ignore in each given run of the application:

```
<div class="refrain">
    <p>
        And I love <span class=”feminine” >her</span>
        <span class=”masculine”>him</span>
    </p>
</div>
```

This is not declarative at all: it is the application's job to know that when the feminine version is chosen, elements of class feminine are present (i.e., shown) and elements of class masculine are absent (i.e., hidden), and vice-versa: the class attribute suddenly impacts not only on the presentation of the lines, but on their presence and content, too.

Finally, the issue of repetitions has a further and subtler issue, that impacts on the difference between the content of a document and its *structured* content.

In the Beatles' song there are some repetitions of quite large structures, such as the refrain and the fourth stanza. It could be considered a pointless exercise in futility to decide whether the refrains of a song are to be considered as one instance of content to be repeated as needed after each stanza, or many different instances whose content happens to be identical. Yet, the praxis of transcription of song lyrics is usually to qualify the refrain lyrics as such the first time they are sung, and then refer back to them all other times without actually repeating the content, and as such we will treat them in our example. Yet the refrains are necessary handled, in an XML document, by repeating the entire structure, duplicating the markup code and the text. We could introduce it the first time only[2], and refer to it in some way the other times – for example, through an “href” attribute – in the other parts of the lyrics. Similarly we could handle the repetition of the fourth stanza, i.e., as follows:

```
<body>
    <h1>And I love her</h1>
    <div id=”first” class="stanza" title="1">[...]</div>
    <div id=”refrain” class="refrain">
        <p>And I love her</p>
    </div>
    <div id=”second” class="stanza" title="2">[...]</div>
    <div href=”#refrain” />
```

[2] We will ignore, for the time being, that in the lyrics that are actually sung by the Beatles the first refrain is slightly different from the other ones, since they sing “I love her” instead of “and I love her”

```
    <div id="third" class="stanza" title="3">[...]</div>
    <div id="fourth" class="stanza" title="4">[...]</div>
    <div href="#refrain" />
    <div href="#fourth" />
    <div href="#refrain" />
</body>
```

Unfortunately, we believe again that this approach is not declarative enough: it is the application's job to understand that the last three div elements are not empty, but refer to the previous-declared elements and repeat their content.

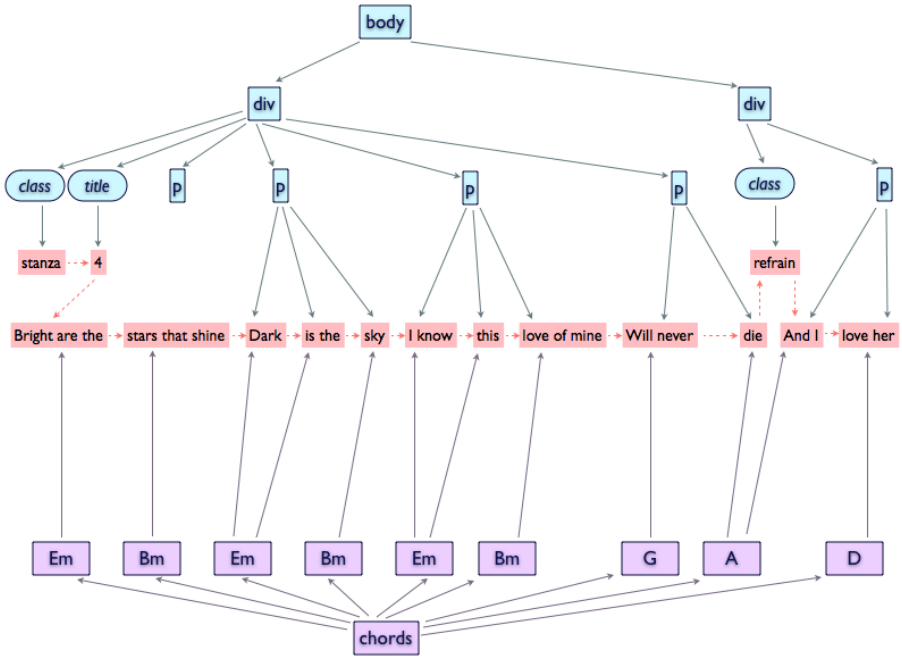

Figure 1. The rGODDAG structure to handle lyrics and chords overlapping. The red dashed line represents, here and in the following figures, the document order.

XML entity references could be used to express repetitions too: content can be declared as an entity to be resolved when users view the document. From a merely presentation perspective, such an approach would be enough as all the repeated content is retrieved and merged into the final XML file. On the other hand, processing entities in a more sophisticated way still require entangled and application-dependant operations. For instance, it is rather complex to add metadata about entities, to extract information about that content, to process those fragments via XSLT or to validate entity fragments.

Joining the above mentioned different hierarchies in a single document and dealing with the issues mentioned so far presents issues that are not manageable with the plain XML armamentarium, and requires special approaches:

- the timing of the A chord overlaps two lines of the lyrics;
- the timed display of the lyrics requires each line to be shown multiple times;
- the popups introduce additional text content to the document, and do so independently of the stanza and line structure of the lyrics;
- text variants require elements that exist or do not exist depending on context;
- repetitions require that content is specified once, and referred to many times.

Some of these structural issues can be handled by standard overlapping approaches, and other can be dealt with by introducing ad hoc, non-declarative markup that is procedurally interpreted by specialized tools. But it is at the level of the data structure model that we prefer to study the problem.

3.2 Restricted GODDAG

Handling overlapping elements requires a more expressive data structure than trees, such as directed graphs. *Restricted GODDAG* ([19] and [12]) are able to deal with the lyrics/chord overlap, as shown in Figure 1.

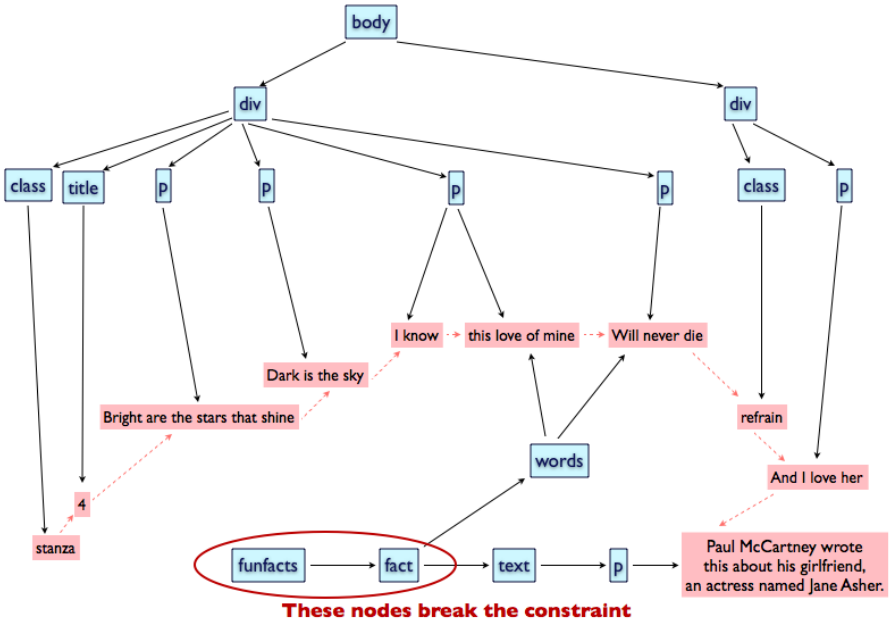

Figure 2. The first tentative to make an r-GODDAG for describing both lyrics and fun fact.

Restricted GODDAGS gives strong support for overlapping structures and guarantees their full linearizability into TexMECS documents [9].

Restricted GODDAGs, on the other hand, will not help us with the management of popups. Restricted GODDAGS have some strong constraints that prevent this:

- each r-GODDAG node dominates a contiguous sequence of leaf nodes (i.e., nodes that contain text);
- no two r-GODDAG nodes that are not connected by a dominance relation, dominate the same subsequence of leaf nodes.

For our popups, these constraints appear quite strong, and particularly the first one. Basically, the requirement of contiguity prevents two hierarchies to overlap on some leaf nodes whenever the content of other leaf nodes is different.

In our example, inserting in the same document both the lyrics structure and the one related to the popups implies breaking the contiguousness of one of the two hierarchies, because there is always a node that breaks the constraint.

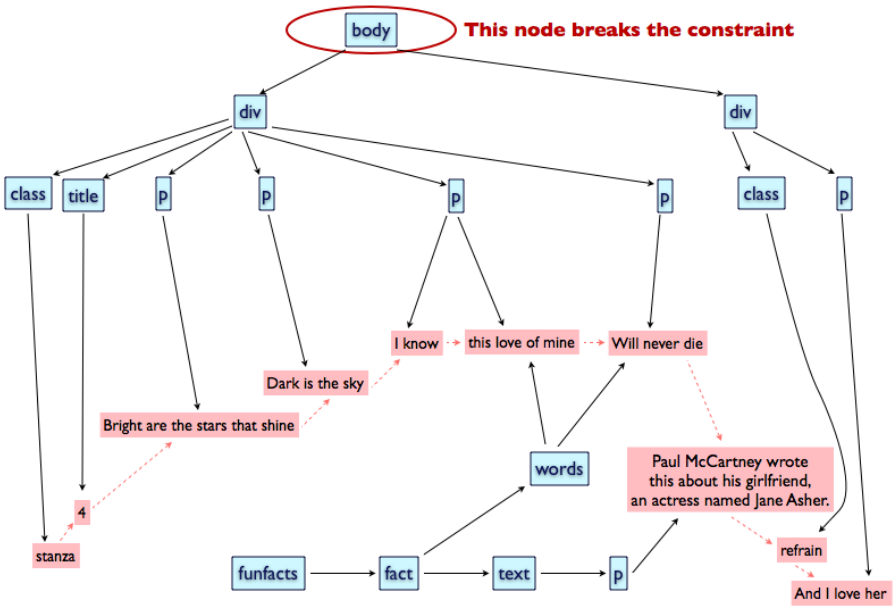

Figure 3. The second tentative to make an r-GODDAG for describing both lyrics and fun facts.

Consider the situation in which a popup is associated to the string “this love of mine Will never die”. If the content of the fun fact is put at the end of the lyrics, as shown in Figure 2, the elements “funfacts” and “fact” both dominate non-contiguous leaf nodes, as “this love of mine” is non-contiguous with “Paul McCartney...”, thus breaking the constraint.

If, on the other hand, the content of the fun fact is put before or after the lines it refers to (Figure 3), the element “body” will dominate non-contiguous leaf nodes (two

of them will be interrupted by the “Paul McCartney...” node that does not belong to that hierarchy) and therefore violate the constraint.

So, even if a restricted GODDAG is a more expressive data structure than a tree, it is still not sufficient to handle complex scenarios such as the ones described. The overall point of the contiguity constraint is to allow for embedding markup within text; thus r-GODDAG structures are indeed representable with milestones or fragmentation in XML, or with TexMECS documents, but more complex structures are still unavailable, such as those involving non-contiguous leaf nodes.

3.3 A more general data structure: the GODDAG

If we give up the feature of embedding, we already have a data structure for handling complex overlapping scenarios: the full GODDAG, which does not require the two constraints mentioned above: it does not require leaf nodes to follow document order, and it does not require that any two different nodes dominate different sets of leaf nodes.

Without these restrictions, we can describe all four hierarchies in a GODDAG, as shown in Figure 4. The obvious disadvantage of this data structure is that embedding everything in a linear structure such as an XML document implies either losing some information or recurring to procedural tricks that would subject the structural meaning of the document to specific tools.

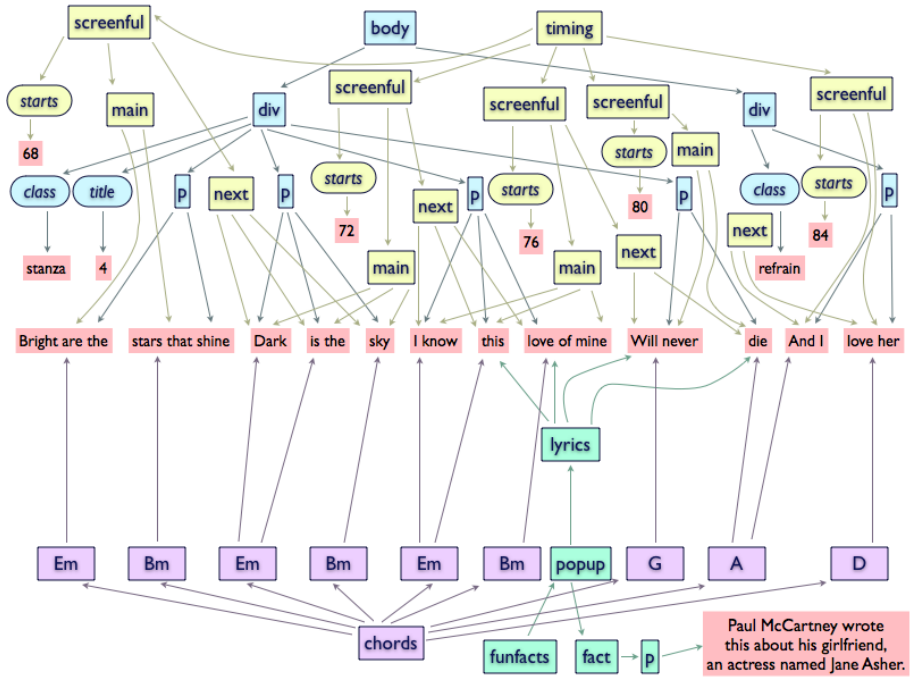

Figure 4. A GODDAG for the four hierarchies defined for the song. Document order is not shown since where it is not obvious (e.g., in the lyrics lines it is completely arbitrary).

In Figure 4 the full GODDAG structure of the three data hierarchies is shown: lyrics (in blue), the time in which the lyrics are shown (in yellow), the chords (in violet) and the fun facts popups (in green). Non-bordered nodes are content, bordered nodes represent markup: rectangles are XML elements, and rounded rectangles are XML attributes.

Moreover, through GODDAG we can handle cases of textual variants and some simple cases of repetitions (for instance, specifying the presence of the class attribute in multiple div elements, and even specifying that the refrain text appears in multiple places, but is really only defined once).

GODDAG can also be employed for textual variants: as shown in Figure 5, since the refrain uses “her” if the lyrics are feminine and “him” otherwise, we actually generate two different and almost identical lyrics hierarchies that point to the text in different manners depending on the chosen gender.

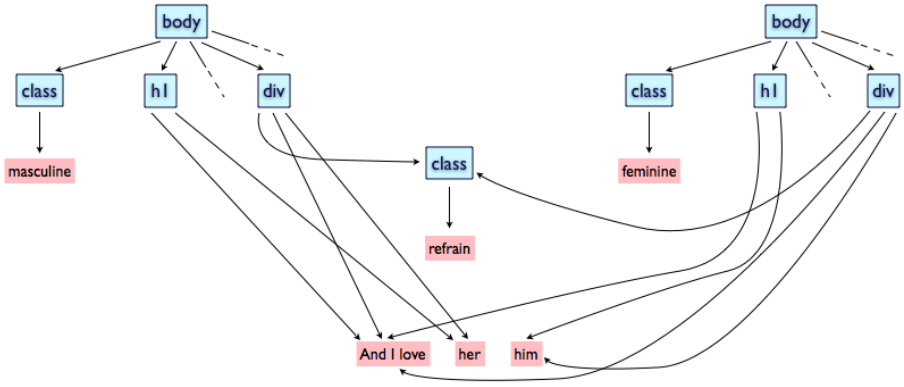

Figure 5. A GODDAG with repetitions and textual variants depending on the gender of the lyrics.

Clearly, the best thing we can do for linearizing all these kinds of structures presented in this section is to use stand-off markup or twin documents techniques in order to embed all the elements in a rationally unique document. Expressing all information in a single XML tree requires some procedural tricks: elements with procedural values, for instance, are an acceptable trade-off between the structure and the relative document representation.

3.4 Beyond GODDAG: extensions for repeatability

Even if the GODDAG is able to handle perfectly all the above-mentioned scenarios, there are more things that are interesting to represent, such as a different type of repetition. The screenful of lines of the karaoke example is interesting in that sense.

In Figure 6 we show a plausible graph for describing the entire structure of “And I Love Her” that avoids the explicit repetitions of the refrain and the fourth stanza. Note

that the body element has many arcs going to the div of the refrain, and two going to the div of the fourth stanza, and that we had to specify the order of the arcs themselves.

Unfortunately, GODDAGs do not support this kind of repetitions. In fact, for any non-terminal node *n*, the sets of arcs from *n* is ordered and, if two arcs *n#a* and *n#b* exist and if *a* is equal to *b*, then *n#a* and *n#b* are the same arc. This prevents us from creating multiple arcs from body to the refrain divs, which is exactly what we are trying to do.

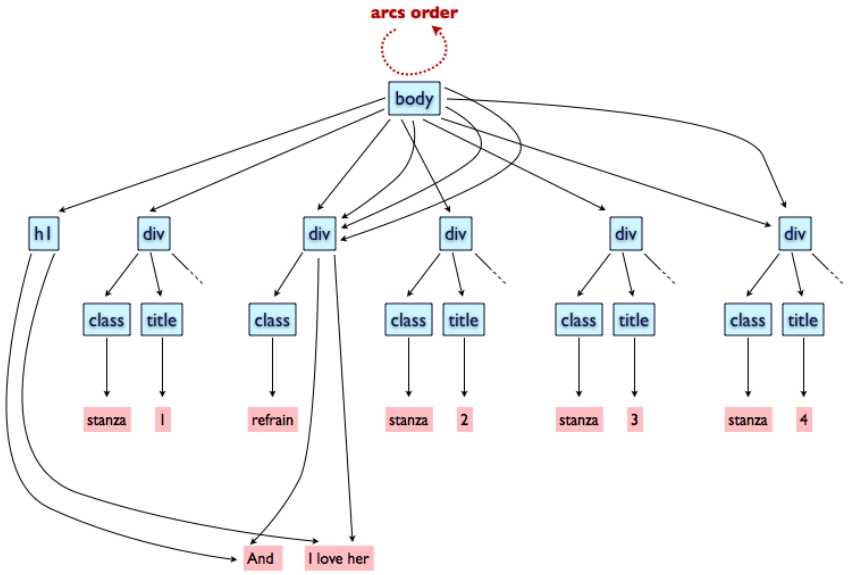

Figure 6. This graph describes the structure of "And I Love Her" avoiding the explicit repetitions (refrain and fourth stanza).

In order to avoid this constraint, we need to extend the definition of GODDAGs. In particular, we believe that we can solve our problem by simply relaxing the rule that requires non-terminal nodes in GODDAGs to have a general identifier (a label) associated to them. We call *anonymous* all non-terminal nodes that do not have such general identifier and we refer to this new GODDAG as *extended GODDAG* (or *e-GODDAG*). Anonymous e-GODDAG nodes allow the definition of anonymous elements *a la* LMNL [21], and at the same time provide the necessary infrastructure for our repeating refrains.

Through e-GODDAG, in fact, we are now able to allow the previous repetitions: we have to add as many anonymous nodes as needed for any repetition we need. Since anonymous nodes do not introduce markup or content, they can be used to disambiguate

multiple arcs going from and to the same nodes: each repeated arc from *body* to *div* is therefore interrupted by a different intermediate anonymous node[3].

4 EARMARKing cats and docs

The problems described in the previous section derive mostly from the very act of embedding annotations: multiple overlapping annotations, especially when referring to the same text multiple times and reordering the document order, do not naturally fit in a linear structure of an XML document, and analogously there is no natural position for embedding annotations to the whole document.

The opposite approach – full externalization of annotations *a la* RDF – does not satisfy our requirements, for different motivations. RDF annotations do not change the annotated resource in any way, but refer to it via URIs. The problem we face in this case is that there exists no URI referring to a fragment of text that is not wrapped within an XML or XHTML element provided with an ID. And since XHTML or XML elements need to follow a nice, hierarchical, document-order-compatible structure, we are back to the beginning with the problem of overlapping hierarchies that play with multiplicities and reshuffling of the document order.

An approach has been recently proposed in [10] in order to try to offer a way to identify precise document locations, called *pointers*, through different means (character positions, string indexing, etc.) and languages (XPath [3], XPointer [5], etc.). Unfortunately some languages mentioned, such as XPointer, were never standardized by the W3C and there is no sign that they will ever be in the foreseeable future. Furthermore, from the RDF point of view all URIs are opaque strings referring to different resources, and as such it would be difficult to create ontologies and make inferences that differentiate assertions on text fragments from assertions on elements or other structures, the required infrastructure to verify overlapping or superimposition of assertions.

There is another (less important) consideration that comes down against a fully externalized approach: the fact that assertions are disjoint from the original document and require a more articulated process for storing and transfer (this is known as the so-called fragility of standoff markup). Consider the case of textual variations in our karaoke example: it would be useful to handle all variations (and any other overlapping hierarchy) within a unique document, easier to move and manipulate. The *variant graph* approach, introduced in [16], goes in that direction and allows users to express these differences and to extract multiple text linearizations, depending on the particular context.

Our approach takes inspiration from this work and from the GODDAG-related theories. The goal is to introduce a new syntactic approach for overlapping markup that combines advantages of embedded and external annotations into a unified framework. In this section we define an ontology-based model for expressing such complex

---

[3] An issue to consider relates to another GODDAG constraint: no node can dominate another node both directly and indirectly. That simply means that we need to add an anonymous node for each repeated arc of a node, and just the ones after the first one, so that the we only have indirect dominance in all of them.

overlapping structures, similar but more general of existing research efforts such as [23] and [24].

A very central point of our proposal is the reliance on Semantic Web technologies. The reason is that we want to create tools that can exploit existing modules, that can be integrated with other applications and that can be extended by other researchers too.

As expected, RDF and OWL are the candidates for our proposal. Actually, we propose an intermediate language built on the top of RDF and OWL data model, that can be straightforwardly translated into these standards. We called this language *EARMARK* (*Extreme Annotational RDF Markup*). EARMARK allows us to build e-GODDAG-equivalent data structures that encode all the aforementioned scenarios. High-level data structures can be then instantiated into W3C standard documents, easy to integrate in legacy tools and environments.

Basically, EARMARK allows us create assertions on text fragments by using an intermediate ontology that subsumes the XPointer schemas in a manageable way and builds from there the concepts of markup structures and generic identifiers useful for the specification of elements and attributes.

4.1 General model

This section describes the model behind EARMARK, *Extreme Annotational RDF Markup*. The model itself is defined through an OWL document specifying classes and relationships. Through these classes we can produce EARMARK documents with assertions about individuals.

We introduce four concepts: *docuverses*, *locations*, *ranges* and *markup items*. Each of them is represented in EARMARK with a different (and disjoint) OWL class. The following code snippets are written using Turtle [2][4].

The textual content of a EARMARK document is conceptually separated from the annotations, and is referred to by means of assertions on the specific class called "Docuverse". This class (and its name) is based on the concept introduced by Ted Nelson in his Xanadu Project [14] to refer to the collection of text fragments that can be interconnected to each other and transcluded into new documents.

The individuals of this class represent the object of discourse, i.e. all the text containers related to a particular EARMARK document.

```
:Docuverse
    a       owl:Class ;
    rdfs:subClassOf owl:Thing .
```

[4] In all code examples we will also be implying the following prefixes:

```
@prefix :        <http://www.essepuntato.it/2008/12/earmark#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
@prefix swrl:    <http://www.w3.org/2003/11/swrl#> .
@prefix swrlb:   <http://www.w3.org/2003/11/swrlb#> .
```

```
:has-text
    a       owl:FunctionalProperty , owl:DatatypeProperty ;
    rdfs:domain :Docuverse ;
    rdfs:range xsd:string .

:has-uri
    a       owl:FunctionalProperty , owl:DatatypeProperty ;
    rdfs:domain :Docuverse ;
    rdfs:range xsd:anyURI .
```

Any individual of the *Docuverse* class – commonly called a *docuverse* (lowercase to distinguish it from the class) – might *contain* or *refer to* the text fragments representing the actual content of the document. That is expressed through two properties: *has-uri* if the content is stored at a particular URI and *has-text* if the content lies in the document itself.

A *location* is the expression of a position in a particular docuverse. It is aninstance of the class "*Location*". The property *at* defines a precise point in the docuverse, while the property *refers-to* indicates the docuverse the location refers to.

```
:Location
      a       owl:Class ;
      rdfs:subClassOf owl:Thing .

:refers-to
    a       owl:FunctionalProperty , owl:ObjectProperty ;
    rdfs:domain :Location ;
    rdfs:range :Docuverse .

:at
    a       owl:FunctionalProperty , owl:DatatypeProperty ;
    rdfs:domain :Location ;
    rdfs:range xsd:string .
```

The value for the property *at* is a string. The overall ontology is then independent from the actual addressing mechanism. In fact, we expect several syntaxes to be used there, including XPointers.

We then define the class "*Range*" for any text lying between two locations:

```
:Range
    a       owl:Class ;
    rdfs:subClassOf owl:Thing .

:begins
    a       owl:FunctionalProperty , owl:ObjectProperty ;
    rdfs:domain :Range ;
    rdfs:range :Location .

:ends
    a       owl:FunctionalProperty , owl:ObjectProperty ;
```

```
    rdfs:domain :Range ;
    rdfs:range :Location .
```

A range, i.e, an individual of the class Range, is defined by a starting and an ending location through the properties *begins* and *ends* respectively. These locations must refer to the same docuverse. Since this restriction cannot be directly expressed in OWL, we add the following SWRL [8] rules to enforce that constraint[5]:

$$\text{begins}(?r,?loc1), \text{ends}(?r,?loc2), \text{refers-to}(?loc2,?d) \Rightarrow \text{refers-to}(?loc1,?d) \quad \textbf{(1)}$$

$$\text{begins}(?r,?loc1), \text{ends}(?r,?loc2), \text{refers-to}(?loc1,?d) \Rightarrow \text{refers-to}(?loc2,?d) \quad \textbf{(2)}$$

There is no restriction on locations used for the *begins* and *ends* properties. That is very useful: it allows us to define ranges that "follow" or "reverse" the text order of the docuverse they refer to. For instance, the string "desserts" can be considered both in document order, with the *begins* location lower than the *ends* location or in the opposite one, forming "stressed" [6]. Thus, the properties "begins" and "ends" define the way a range must be read.

The class "*MarkupItem*" is the superclass defining artefacts to be interpreted as markup (such as elements and attributes).

```
:MarkupItem
    a       owl:Class ;
    rdfs:subClassOf owl:Thing .

:has-general-identifier
    a       owl:FunctionalProperty , owl:DatatypeProperty ;
    rdfs:domain :MarkupItem ;
    rdfs:range xsd:string .
```

A *markupitem* individual is a sequence (rdf:Bag or rdf:Seq) of individuals belonging to the classes MarkupItem and Range. Is it then possible to define elements containing nested elements or text, or attributes containing values, as well as overlapped and complex structures.

A markupitem might have a name, specified in the property "*has-general-identifier*" (recalling the SGML term to refer to the name of elements [7]). Note that we can classify markup items as *anonymous* – as possible in LMNL[21] and e-GODDAG – by simply not asserting a general identifier for the items.

All the concepts represented by an EARMARK document are expressed using these four disjoint classes and their relative properties:

```
[]
    a       owl:AllDisjointClasses ;
    owl:members (:Docuverse :Location :MarkupItem :Range) .
```

---

[5] Because of the functional property declarations of "begins" and "end" and the SWRL rules illustrated, an EARMARK document will be consistent if and only if the constraint is valid. Otherwise, there will be a range with two locations that refer to two different documents.

[6] An interesting example of *semordnilap*, http://en.wikipedia.org/wiki/Palindrome#Semordnilaps

4.2 Detailed model

The model discussed so far gives us a general picture of the EARMARK framework and, as expected, is not enough to describe all the scenarios we are interested in. We then need to refine our model. Such a refinement is actually a specialization of three classes – all except "Range" – in subclasses that apply specific restrictions.

First of all, the class Docuverse is specified into a "StringDocuverse" (the content is specified as value of *has-text* and no value is associated to *has-uri*) or an "URIDocuverse" (the actual content is located at the URL specified in *has-uri* and no value is given to *has-text*).

```
:StringDocuverse
    a       owl:Class ;
    rdfs:subClassOf :Docuverse ;
    owl:equivalentClass
        [
            a       owl:Class ;
            owl:intersectionOf (
                :Docuverse [
                    a       owl:Restriction ;
                 owl:cardinality "1"^^xsd:nonNegativeInteger ;
                    owl:onProperty :has-text ]
                [
                    a       owl:Restriction ;
                    owl:cardinality "0"^^xsd:nonNegativeInteger ;
                    owl:onProperty :has-uri ] ) ] .

:URIDocuverse
    a       owl:Class ;
    rdfs:subClassOf :Docuverse ;
    owl:equivalentClass
        [
            a       owl:Class ;
            owl:intersectionOf (
                :Docuverse [
                    a       owl:Restriction ;
                 owl:cardinality "0"^^xsd:nonNegativeInteger ;
                    owl:onProperty :has-text ]
                [
                    a       owl:Restriction ;
                    owl:cardinality "1"^^xsd:nonNegativeInteger ;
                    owl:onProperty :has-uri ] ) ] .

[]
    a       owl:AllDisjointClasses ;
```

```
    owl:members (:StringDocuverse :URIDocuverse) .
```

Depending on particular scenarios or on the kind of docuverse we are dealing with – it could be plain-text, XML, LaTeX, a picture, etc. – we need to be able to use different kinds of locations. Therefore, the class “Location” has at least three different disjoint subclasses:

```
:CharNumberLocation
    a       owl:Class ;
    rdfs:subClassOf :Location .

:XPathLocation
    a       owl:Class ;
    rdfs:subClassOf :Location .

:XPointerLocation
    a       owl:Class ;
    rdfs:subClassOf :Location .

[]
    a       owl:AllDisjointClasses ;
    owl:members (
        :CharNumberLocation :XPathLocation :XPointerLocation
    ) .
```

- “CharNumberLocation” defines a location by counting characters. In that case, the string value of the “at” property must be an integer – it is a positive integer (including zero) if we are counting from the begin of the document to the end, and a negative integer for vice versa – that identifies an unambiguous position in the character stream;
- “XPathLocation” defines a location as a node of an XML docuverse. In this case, the property “at” will be an XPath expression [3];
- “XPointerLocation” defines a precise point in a docuverse. In that case, the expression “xpointer(point(.42))”, for instance, indicates the cursor in-between the 42nd and the 43rd character; with “xpointer(point(/1/9.3))” we mean the cursor between the 3rd and the 4th character of the ninth node of the root, and so on.

MarkupItem is specialized in three disjointed sub-classes: “Element”, “Attribute” and “Comment”, that allow a more precise characterization of markup items.

```
:Element
    a       owl:Class ;
    rdfs:subClassOf :MarkupItem .

:Attribute
    a       owl:Class ;
    rdfs:subClassOf :MarkupItem .
```

```
:Comment
    a       owl:Class ;
    rdfs:subClassOf :MarkupItem .

[]
    a       owl:AllDisjointClasses ;
    owl:members (:Attribute :Comment :Element) .
```

Through this classification, shown also in Figure 7, we can describe all the concepts introduced by XML, LMNL or TexMecs, including virtual elements [18] [9], structured attributes [21] and so on.

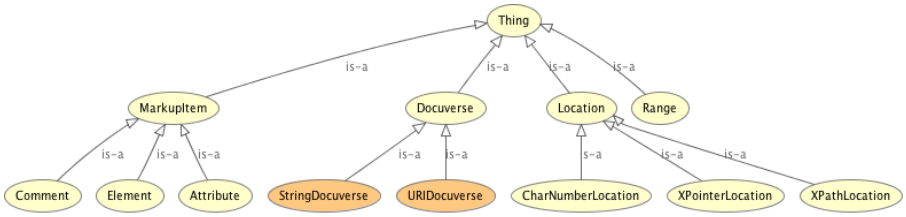

Figure 7. The class hierarchy of the EARMARK ontology.

In order to discuss such potentialities, the next section analyses in detail the EARMARK encoding of the aforementioned karaoke example.

4.3 "And I Love Her" in EARMARK

The approach to mark up a complex document with EARMARK is composed of the following steps:

- creation of one or more docuverses depending on the number of data streams we must handle;
- identification of the ranges within the docuverses;
- identification of the *leaf* markup items, i.e. those containing attributes and ranges only;
- identification of the *internal* markup items, i.e. those containing markup items or a mixed content of markup items and ranges.

Let us take into consideration the fragment of the lyrics of "And I Love Her" by The Beatles, introduced in Section 3.1.

In EARMARK strings are placed in one or more docuverses. As mentioned, there are two different types of docuverses: *autonomous resources* (i.e., independent files identified by a URIs, appropriate for the actual lyrics of the song and the content of the fun fact popups) and *local strings* (i.e. an internal data value, appropriate for strings that do not exist as independent units like attribute values, metadata, and so on).

For the XML version of “And I Love Her”[7], we will employ four docuverses:

- an independent text file with the lyrics;
- a local string containing strings for all attribute values;
- a local string containing the timings of the screenfuls of lyrics;
- an independent file with a selection of fun facts. This could just as well be an existing, independent HTML resource such as the one in http://www.songfacts.com/detail.php?id=43.

Note that we have immediately introduced the machinery for overlapping elements and shared text fragments. We can also add any additional annotation (such as spaces, separators, etc.) to each docuverse in order to make it more readable. We will be explicitly ignoring the non-relevant text within the docuverses.

The Turtle translation of the docuverses could be[8]:

```
e:lyrics
    a :URIDocuverse ;
    :has-uri
                          "http://www.essepuntato.it/2009/01/
andiloveher.txt"^^xsd:anyURI .

e:funfacts
    a :URIDocuverse ;
    :has-uri
                        "http://www.songfacts.com/detail.php?
id=43"^^xsd:anyURI .

e:attribute_values
    a :StringDocuverse ;
    :has-text "stanza - refrain - 4"^^xsd:string .

e:time_values
    a :StringDocuverse ;
    :has-text "68 - 72 - 76 – 80 - 84"^^xsd:string .
```

All the strings defining the actual text content of an EARMARK document are identified by ranges. Ranges refer to any of the docuverses, and can overlap and invert order. For example, the ranges for the refrain and the last chord overlap over the same range.

We next define a range for each text node of the song, encoded as element or attribute, e.g.:

```
e:r_refrain_1
    a :Range ;
    :begins e:location0-lyrics ;
```

---

[7] The complete Turtle example if “And I Love Her” is available at “http://www.essepuntato.it/2009/01/andiloveher.ttl”.

[8] The prefix “e” refers to “http://www.essepuntato.it/2009/01/andiloveher#”.

```
    :ends e:location6-lyrics .

e:r_refrain_2
    a :Range ;
    :begins e:location6-lyrics ;
    :ends e:location14-lyrics .

e:r_attribute_class_refrain
    a :Range ;
    :begins e:location9-attribute_values ;
    :ends e:location16-attribute_values .

e:location0-lyrics
    a :XPointerLocation ;
    :refers-to lyrics ;
    :at "xpointer(point(.0))"^^xsd:string .

e:location6-lyrics
    a :XPointerLocation ;
    :refers-to lyrics ;
    :at "xpointer(point(.6))"^^xsd:string .

e:location14-lyrics
    a :XPointerLocation ;
    :refers-to lyrics ;
    :at "xpointer(point(.14))"^^xsd:string .

e:location9-attribute_values
    a :XPointerLocation ;
    :refers-to attribute_values ;
    :at "xpointer(point(.9))"^^xsd:string .

e:location16-attribute_values
    a :XPointerLocation ;
    :refers-to attribute_values ;
    :at "xpointer(point(.16))"^^xsd:string .
```

Some ranges can be used more than once in the final EARMARK document. For instance, the “r_refrain_2” range is used both in the refrain of the song and in the last chord of the refrain.

Using these ranges we can now create the leaf markup items, i.e. all the attributes and all the *first-level* elements. The latter are all the elements that have a simple content, i.e., sequences of ranges and attributes only.

Given an e-GODDAG node *N*, an EARMARK markup item is made as follows:

- it has an identifier generated randomly;
- the name of *N*, if it exists, is the general identifier;

- all children non-terminal nodes of *N* are translated into individual markup items. They are recursively generated with these same rules;
- the ranges corresponding to the text content end up as the sequence of the new markup item.

In the next piece of code we take into consideration both the e-GODDAG structure and the implicitly given XML description for all the markup items, that defines the kind – *element* or *attribute* – of each of them. For instance, the Turtle translation of the attribute class and of the p element of the refrain, using the ranges previously defined, is:

```
e:attr_refrain_class
    a :Attribute ,
    [ a rdf:Bag ; rdf:_1 e:r_attribute_class_refrain ] ;
    :has-general-identifier "class"^^xsd:string .

e:refrain_div
    a :Element ,
    [   a rdf:Seq ;
        rdf:_1 e:attr_refrain_class ;
        rdf:_2 e:refrain_p ] ;
    :has-general-identifier "div"^^xsd:string .

e:refrain_p
    a :Element ,
    [   a rdf:Seq ;
        rdf:_1 e:r_refrain_1 ; rdf:_2 e:r_refrain_2 ] ;
    :has-general-identifier "p"^^xsd:string .
```

The difference between those leaf elements that are simply sequences of ranges and those that are sequences of attributes and ranges mirrors the difference between types in XML Schema [22], with the former resembling simple type elements with simple content, and the latter resembling complex type elements with simple content and attributes.

The expressiveness of e-GODDAG's is clearly within EARMARK's : through EARMARK we can express general digraphs with or without *repeatable edges* depending on the particular context we are taking into consideration.

Through such digraphs we can handle particular scenarios that involve overlapping – i.e. different elements partially dominate the same content, such as with chords and lines – as well as virtual elements – i.e. non-contiguous ranges are contained by a markup item, such as with the fun fact pop-up.

Finally, it is interesting to note that EARMARK is actually more expressive than e-GODDAGs. Consider the case of unordered items. Although e-GODDAGs always considers ordered markup items and ranges within a container, EARMARK allows us to specify whether the items are ordered or not, by simply using “rdf:Seq” and “rdf:Bag” container classes. The ordering of inner elements becomes a matter of explicit choice rather than implicitly given by the markup embedding.

Consequently, EARMARK even allows us to specify sequences of attributes, elements and ranges in any arbitrary order. Differently than XML, LMNL and

TexMECS, EARMARK makes possible sequences in which attributes, elements and ranges are freely mixed in any order, including elements followed by attributes followed by other elements and so on. Moreover, the same global identifier can be specified for multiple attributes in the sequence (i.e., EARMARK allows multiple attributes with the same name for the same element). These situations are not directly expressible in any embedded markup model.

5 Embedding EARMARK documents

The process of generating a linearized structure (such as an XML document) from a set of EARMARK annotations is not immediate, mostly because of the substantially greater expressive power of EARMARK annotations. Without loss of generality, we will be describing a conversion to XML, since converting to LMNL or TexMecs will constitute a much simpler exercise of stopping the linearization a few steps earlier.

Although the conversion of any EARMARK subset that already describes a tree is obviously immediate and fully automatic, several different options exist for any further EARMARK annotations that we wish to linearize. Since these additional annotations are at odds with a tree-like structure, we need to use a few embedding tricks to obtain a well-formed XML document, and of course the choice of tricks to use is wide and rich. In this section we will explore the task of linearizing a chosen tree-shaped subset of the EARMARK document, and in the following section we will describe a few options for the remaining assertions.

The construction of the tree we envision is bottom up:

- the first step is deciding which docuverses (or fragments thereof) will constitute the content of the document, which the content of the attributes, and which, if any, will be ignored;
- then a subset of the first-level elements needs to be chosen, as well as the ranges they contain. Of course, no overlapping or reverse order ranges can be accepted as such;
- there might well be the situation whereby multiple independent sets of first-level elements exist, each of which is by itself non-overlapping, but combined with others would. In this situation, of course, only one set can be selected as the main hierarchy, and all others will need to employ an embedding trick to be expressed in the final linearized document. One possible way to do so is to create independent sets of elements and hierarchy over elements, and then choose the largest set as composing the principal hierarchy, and all others as candidates for tricks;
- mixed content elements are sequences of ranges and first-level elements, and are generated once all contained elements are ready;
- similarly, structure elements (only containing other elements) are available for creation once their content is already generated;
- finally, attributes and their ranges are selected as well and converted into linearized form and associated to their elements;

- the final result of this linearization is possibly a selection of separate and disjoint trees, each linearizing a connected component of the EARMARK document. It is then a linearization choice either to generate several independent XML documents or to employ the *universal root* pattern[9] and include these structures within a single *new* elements that become their container.

Whatever is left out of this linearization process needs to be approached using one or more of the methods described in the next section.

6 Handling the remaining EARMARK structures

Some kinds of EARMARK structures are not directly linearizable by embedding. In order to allow a full representation of the EARMARK document we therefore need to apply some stratagem to force the hierarchical structure to accept these *remaining structures*.

Reasonably, frequent unmanaged structures would include:

- overlapping leaf elements referring to contiguous ranges;
- overlapping leaf elements referring to non-contiguous ranges;
- shared ranges;
- text variants;
- overlapping structural elements;
- structured attributes.

In Section 5 we listed the EARMARK assertions that could not be directly translated into an XML document. Let us examine a few potential approaches (which we call *embedding tricks*) for forcing the conversion. A few of such approaches, as well as algorithms for passing from one to the other, are described in [13]

6.1 Milestones

Plain overlapping leaf elements (i.e. elements that partially share the text content, but no lower structures) may be forced into an XML structure via *milestones* as proposed in CLIX [6].

The open and close tags of the unconverted elements are considered as individual empty elements placed in the positions where they should reside. The attribute role specifies whether the empty element corresponds to a start or end tag, and the *sID* and *eID* attributes connect the two elements in a single conceptual one.

```
<body>
    <div class=”stanza” title=”4”>
        <p>
            <chord name=”G”>Will never</chord>
         <chord name=”A” clix:role=”start-range” clix:sID=”A”/>
            die
```

```
        </p>
    </div>
    <div class=”refrain”>
        <p>
             And I
          <chord name=”A” clix:role=”end-range” clix:eID=”A”/>
            <chord name=”D”>love her</chord>
        </p>
    </div>
</body>
```

Although easy to implement and appreciate, milestones are nonetheless limited in that only frontier overlapping (i.e., overlapping on ranges) is expressible.

6.2 Fragmentation

Another approach is to use fragmentation as introduced by the TEI guidelines [17].

Overlapping elements are separated in many multiple fragments each of which properly nests within their container. Individual fragments are then connected via attributes such as *next* or *previous*.

```
<body>
    <div class=”stanza” title=”4”>
        <p>
            <chord name=”G”>Will never</chord>
            <chord name=”A” xml:id=”a1” next=”a2”>die</chord>
        </p>
    </div>
    <div class=”refrain”>
        <p>
             <chord name=”A” xml:id=”a2”>And I</chord>
             <chord name=”D”>love her</chord>
        </p>
    </div>
</body>
```

6.3 Repetitions

The easiest embedding trick for dealing with shared ranges is simply to multiply the instances of the corresponding text and possibly annotate that all instances except the first one is redundant.

```
<p>
    <span class=”repeat” title=”r_refrain_1”>And I </span>
    <span class=”repeat” title=”r_refrain_2”>love her</span>
</p>
```

6.4 Hidden variants

When we have multiple variants of the same text, we may want to hide in substructures (such as attributes or subelements) the alternative variants.

```
<p>And I love <span class=″alternative″ title=″him″>her</
span></p>
```

6.5 RDFa

RDFa [1] allows arbitrary assertions to be placed on existing elements. It is understood that if an assertion exists over a text fragment that is not wrapped within an existing element, a generic element (such as the HTML *span*) is added to allow for RDFa assertions to attach to the corresponding content.

For instance, support for overlapping inner structures are difficult to provide in either fragmentation or milestones, but become possible in RDFa. Consider for instance the sequence which contains individual chord elements and overlaps with the *div* element containing individual *p* elements.

RDFa thus supports the specification of a virtual instance of the class Chords, expressed as a sequence of three instances of the Chord class (in fact, one instance each of subclasses GChord, AChord and DChord of the Chord class) as follows:

```
<body about=″#Chs″ typeof=″#Chords″>
    <div typeof=″rdf:Seq″ property=″rdf:_1″ href=″#G″>
        <p property=″rdf:_2″ href=″#A″>
            <span about=″#G″ typeof=″#GChord″ property=″#has″>
                Will never
            </span>
            <span
                   about=″#A″ typeof=″#AChord″ property=″#has-
first-part″>
                die
            </span>
        </p>
    </div>
    <div property=″rdf:_3″ href=″#D″>
        <p>
             <span about=″#A″ property=″#has-second-part″>
                 And I
             </span>
             <span about=″#D″ typeof=″#DChord″ property=″has″>
                 love her
             </span>
        </p>
    </lg>
```

```
</body>
```

6.6 Embedded RDF

When all else fails, the fallback approach is simply to place the remaining assertions as an RDF/XML block in the XML structure, either in a block properly thought out for external vocabularies, or converted into some local vocabulary, or even as a lump of XML elements placed in a random position within the document.

This is useful, for instance, for dealing with structured attributes *a la* LMNL [21]. In the following example, a RDF block is inserted in the XML document to provide support for the attribute *name* of the *chord* element, which contains a structure of two different values wrapped by elements *normal* and *jazzy*. This allows the *name* of the *chord* to cater for both a pop and a jazz rendering of the tune, while at the same time remaining one attribute of one element.

```
<body>
    <rdf:RDF
       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns="http://www.essepuntato.it/2008/12/earmark#">
        <Attribute rdf:about="#attr_chord_structured">
            <has-general-identifier rdf:datatype="xsd:string">
                name
            </has-general-identifier>
            <rdf:type>
                <rdf:Seq>
                    <rdf:li rdf:resource="#normal"/>
                    <rdf:li rdf:resource="#jazzy"/>
                </rdf:Seq>
            </rdf:type>
        </Attribute>
        <Element rdf:about="#normal">
            <has-general-identifier rdf:datatype="xsd:string">
                normal
            </has-general-identifier>
            <rdf:type>
                <rdfs:Bag>
                    <rdf:li rdf:resource="#r_chord_D"/>
                </rdfs:Bag>
            </rdf:type>
        </Element>
        <Element rdf:about="#jazzy">
            <has-general-identifier rdf:datatype="xsd:string">
                jazzy
            </has-general-identifier>
            <rdf:type>
                <rdfs:Bag>
                    <rdf:li rdf:resource="#r_chord_Dmaj7"/>
```

```
                </rdfs:Bag>
            <rdf:type>
        </Element>
    <rdf:RDF>
    ...
</body>
```

7 Conclusions

In this paper we presented the Extreme Annotational RDF Markup (EARMARK), a proposal for expressing GODDAGs (and e-GODDAG) structures in a general metamarkup language that does not rely on embedding, and that integrates the advantages of standoff annotations and embedded markup into a single unifying framework.

Through EARMARK authors can express a large number of markup assertions and observations that would otherwise be non expressible, including overlapping elements, elements over non contiguous ranges, repeated structures, text variants, overlapping hierarchies, structured attributes, etc.

In further work we plan both to explore the application of the structural patterns defined in [4], trying to give a formal ontological demonstration if an EARMARK document follows them, and to explore, both formally and pragmatically, the expressive power of the EARMARK language and its applications.

8 Acknowledgements

The authors wish to thank all that have commented on this paper and on EARMARK in general. The anonymous reviewers of this and other EARMARK papers were incredibly useful in smoothing out both the basic concepts and the written explanation of the most oscure parts of the proposal. The participants to the Goddag Workshop in Amsterdam (December 2008) have in no small part provided thought fodder for what ended up becoming EARMARK. Finally, we wish to thank explicitly Michael Sperberg-McQueen and Federico Meschini for their help, comments and suggestions.

Bibliography

[1] Adida, B., Birbeck, M., McCarron, S., Pemberton, S. (2008). RDFa in XHTML: Syntax and processing. W3C Recommendation. World Wide Web Consortium. http://www.w3.org/TR/rdfa-syntax/.

[2] Becket, D., Berners-Lee, T. (2008). Turtle - Terse RDF Triple Language. W3C Team Submission. http://www.w3.org/TeamSubmission/turtle/.

[3] Berglund, A., Boag, S., Chamberlin, D., Fernández, M. F., Kay, M., Robie, J., Siméon, J. (2007). XML Path Language (XPath) 2.0. W3C Recommendation. http://www.w3.org/TR/xpath20/.

[4] Dattolo, A., Di Iorio, A., Duca, S., Feliziani, A.A., Vitali, F. (2007). Structural patterns for descriptive documents. In the Proceedings of the Seventh International Conference on Web Engineering 2007, Como, Italy, 2007.

[5] DeRose, S., Maler, E., Daniel, R. (2001). XML Pointer Language (XPointer) Version 1.0. W3C Candidate Recommendation.

[6] DeRose. S. (2004). Markup overlap: A review and a horse. In Extreme Markup Languages.

[7] Goldfarb, C. F. (1990). The SGML Handbook. Oxford University Press, USA.

[8] Horrocks, I., Patel-Schneider, P. F., Boley, H. Tabet, S., Grosof, B., Dean, M. (2004). SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission. http://www.w3.org/Submission/SWRL/.

[9] Huitfeldt, C., Sperberg-McQueen, C. M. (2001). TexMECS: An experimental markup meta-language for complex documents.

[10] Iglesias, C., Squillace, M. (2009). Pointer Methods in RDF. W3C Working Draft availables in http://www.w3.org/TR/Pointer-in-RDF.

[11] Manola, F., Miller, E. (2004). RDF Primer. W3C Recommendation. http:// www.w3.org/TR/rdf-primer/.

[12] Marcoux, Y. (2008). Graph characterization of overlap-only TexMECS and other overlapping markup formalisms. Paper presented at the Balisage: The Markup Conference.

[13] Marinelli, P., Vitali, F., Zacchiroli, S. (2008). Towards the unification of formats for overlapping markup. The New Review of Hypermedia and Multimedia.

[14] Nelson, T. (1980). Literary Machines: The report on, and of, Project Xanadu concerning word processing, electronic publishing, hypertext, thinkertoys, tomorrow's intellectual... including knowledge, education and freedom - Mindful Press, Sausalito, CA, USA.

[15] Oliver Schonefeld und Andreas Witt (2006). Towards validation of concurrent markup. In: Proceedings of the Extreme Markup 2006, Montréal, Canada.

[16] Schmidt, D., Colomb, R. (2009). A data structure for representing multi-version texts online. International Journal of Human-Computer Studies.

[17] Sperberg-McQueen, C. M., Burnard, L. (2005). TEI P5 Guidelines for Electronic Text Encoding and Interchange (revised). The Association for Computers and the Humanities.

[18] Sperberg-McQueen, C. M., Huitfeldt, C. (2008). Markup Discontinued: Discontinuity in TexMecs, Goddag structures, and rabbit/duck grammars.

[19] Sperberg-McQueen, C.M., Huitfeldt, C. (2004). GODDAG: A Data Structure for Overlapping Hierarchies. Lecture Notes In Computer Science. Springer.

[20] Tennison, J. (2008). Representing Overlap in XML. Article from “Jeni's Musings” blog, available in http://www.jenitennison.com/blog/node/97.

[21] Tennison, J., Piez, W. (2002). The Layered Markup and Annotation Language (LMNL). Paper presented at the Late breaking at Extreme Markup. Montreal, Canada.

[22] Thompson, H. S., Beech, D., Maloney, M., Mendelsohn, N. (2001). XML Schema Part 1: Structures. W3C Recommendation. http://www.w3.org/TR/xmlschema-1/.

[23] Tummarello, G., Morbidni, C., Pierazzo, E. (2005). Toward textual encoding based on RDF. 9th ICCC Conference on Electronic Publishing (ELPUB 2005). Leuven, Belgium.

[24] W3C OWL Working Group (2009). OWL 2 Web Ontology Language Document Overview. W3C Working Draft. http://www.w3.org/TR/owl2-overview/.