Towards the unification of formats for overlapping markup

Paolo Marinelli* — pmarinel@cs.unibo.it
Fabio Vitali* — fabio@cs.unibo.it
Stefano Zacchiroli* — zacchiro@cs.unibo.it

Abstract

Overlapping markup refers to the issue of how to represent data structures more expressive than trees—for example direct acyclic graphs—using markup (meta-)languages which have been designed with trees in mind—for example XML. In this paper we observe that the state of the art in overlapping markup is far from being the widespread and consistent stack of standards and technologies readily available for XML and develop a roadmap for closing the gap.

In particular we present in the paper the design and implementation of what we believe to be the first needed step, namely: a syntactic conversion framework among the plethora of overlapping markup serialization formats. The algorithms needed to perform the various conversions are presented in pseudo-code, they are meant to be used as blueprints for researchers and practitioners which need to write batch translation programs from one format to the other.

1 Introduction

This paper is about *markup*, one of the key technological ingredient of hypertext. The particular aspects of markup we are concerned with are the *limits* of its expressivity. XML-based markup requires that the identified features of a document are organized hierarchically as a single tree, whereby each fragment of the content of the document is contained in one and only one XML element, each of which is contained within one and only one parent element all the way up to the single root element at the top.

This restriction has much less to do with the purpose and characteristics of the markup of elements, and more with the ease of processing of trees as opposed to more general abstract structures such as directed graphs. In fact, there are plenty of situations in which the same fragment of content needs to be associated to different, and possibly hierarchically incompatible markup descriptors. This is referred to, in the literature [13, 14], as the *overlapping problem*, as expressed by the following snippet.

Example 1.1. `<doc><b>John <i>likes</b> Mary</i></doc>`

Such a fragment is not well-formed XML markup: it is at most XML-like because neither the `<i>` element nests properly with the `<b>` element, nor vice-versa. Intuitively, a slice of both elements *overlap* with the others corresponding to the word "likes", hence the name. □

In digital publishing a scenario like Example 1.1 can be perfectly meaningful; although solutions can be found and implemented to work around the situation and obtain the expected typographical result, they are just workarounds, and overlapping markup can be seen as an intrinsic feature of the document structures of texts in specific domains. Some more examples are given below.

[1]Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7—40127 Bologna, Italy

Computational linguistics and (computer assisted) literary analysis are two interdisciplinary fields of some importance. Markup is used in both fields to encode orthogonal text facets, such as grammatical structure, phonetic structure, and typographical structure (see [4] for other examples). Not only it is perfectly reasonable for encoders to try to encode all such different structures—which overlap each other in all but trivial cases—in the very same document, but also in many cases it is precisely the relationship among structures which is the subject of study. For example, *enjambment* [1] in poetry can be expressed as a particular overlapping situation among the verse and the grammatical hierarchies; similarly, in verse dramas the metrical and dramatic structures may arrange the same text differently, possibly giving rise to overlapping components.

Text change tracking. Documents do change and change management systems are still both an active research topic and a vibrant software development area, due to the relevance of change tracking in business workflows. Wherever highly specialized technologies for change tracking (e.g. client-server systems or hard to use task-specific software) are not feasible choices, simpler solutions that keep track of changes together with the documents which are subject to change have been developed (e.g. OpenOffice.org and Microsoft Office built-in change management). Given that document formats are more and more frequently represented on disk as XML documents, it is becoming a common need to represent multiple versions as a single physical document, preserving sharing wherever possible; of course version markups overlap each other and XML is not up to the task. Consider for instance the situation in which two paragraphs are merged into one: in a containment hierarchy such as the ones used in the above mentioned office applications, both the markup for deletions and the markup for paragraph contain their content, so that the deletion element would have to contain the end of the first paragraph as well as the start of the following one. The issue of overlaps between change tracking annotation and structure annotation can be traced back in the old hypertext literature, with some seriously opposed to actually using any embedded markup [24], and others proposing a possible way out, based on an SGML CONCUR-like syntax [29].

As a consequence of the intrinsic limitation of XML with regard to overlapping markup, researchers and practitioners interested in markup have studied several aspects of the issue and the state of the art now roughly sports:

1. several (incompatible) serialization formats for encoding documents with overlapping issues in XML syntax (e.g. [3, 5, 12]);
2. some other (still incompatible) serialization formats based on *non*-XML syntaxes and some related data models (e.g. [16, 19, 28]);
3. software tools for processing the documents at points (1) and (2) (e.g. [7, 31]);
4. a family of data structures (called *GODDAG* [26]) to reason about documents with overlapping issues in the graph theory setting;
5. some embryonic proposals to assert validity requirements for documents with overlapping issues [25, 27] and for querying documents with overlapping issues [22, 2, 23].

What is missing is a complete framework for dealing with overlapping situations, a framework that is as complete and rich as the one available today for XML: this diversity in formats and models is based both on different approaches to overlapping situations as well as different features needed in specific projects. The rooting in history of such diversity of approaches makes it quite hard now to converge towards a single solution for all situations.

Yet, clearly, a unification would be highly desirable. Every different serialization format requires its specific sets of tools, in terms of visualization stylesheets, validation schemas, data mining query languages, etc. Non-XML syntaxes, require ad hoc tools even for editing and visualizing documents. Multiplication of efforts in all fields even disconnected from the actual capture of overlapping meaning is inevitable.

On the other hand, it would be a bonus to decide on a standard syntax for representing documents (similar to XML, or based on XML itself), a standard mechanism for expressing constraints on vocabularies (similar to DTD, XML Schema, or Relax NG), a standard approach to query the content of a document and identify relevant features within it (similar to XQuery and XPath), a standard mechanism for translating and displaying documents (similar to XSLT), and a standard event-based or in-memory representation of the document features (similar to SAX and DOM APIs). But, as it is, the current scenario is not particularly encouraging for people needing to explicitly encode overlapping hierarchies in their documents (which format to choose? which software tool?).

A different approach would be creating a *conversion framework* allowing existing collections[1] to pass from one format to any of the other, allowing scholars and practitioners to select the tool that best suits their needs regardless of the overlapping approach required by the tool. We aim at creating an unifying framework for reasoning about and processing documents with overlapping issues, de facto combining the efforts of theoretical studies and software tools for dealing with such documents. Where possible, the framework should enable reusing as much as possible legacy XML technologies, to exploit the already reached critical mass of XML users and tools.

Some of the efforts required to achieve our result predate ours, for example the availability of an agreed upon family of data structures such as GODDAG is undoubtfully a relevant milestone, some other efforts are beyond the scope of this paper (e.g. validation or semantic integration), and therefore a specific effort is the topic of this paper: the unification of *serialization* formats. To the best of our knowledge, such an attempt is new: even though some case by case conversion algorithms have been presented in the past (and are referred to in the algorithmic sections later on), no comprehensive treatment of the most widespread serialization formats for overlapping markup has been attempted before.

We have designed and implemented a translation framework among several different serialization formats which have been proposed in the overlapping markup literature and which have been already implemented within various software tools. Our claim is that such a framework will enable sharing document and tools among the communities which are behind the formats and will also enable, passing through XML formats, to exploit where semantically feasible legacy, overlapping-agnostic, XML software tools.

Paper structure. Section 2 reviews the state of the art in overlapping markup serialization formats, focusing on the main actors, which are also those supported by our translation framework; it also describes the *restricted GODDAG* data structure, which embodies the structural semantics of the supported documents, and is at the core of our framework. Section 3 presents the translation framework first by describing its architecture, then describing the algorithms developed for performing the various conversions; some algorithm properties are discussed as well. Section 4 concludes the paper mentioning our implementation and sketching future work.

2 State of the art in overlapping markup

2.1 Features of complex text documents

When marking up documents in certain domain it might be necessary to represent document features that do not fit into a single tree structure as that conveyed by an XML document. There are different kinds of such features, referred to using different terminology in the literature, with

[1]in fact, there are already quite a lot of document corpora freely available, of varying size, which are encoded using different overlapping markup formats. Just to mention a few of them: the Cambridge Wittgenstein Archive at `http://www.wittgen-cam.ac.uk/cgi-bin/forms/home.cgi` (using techniques developed by the MLCD project); the Electronic Beowulf, Digital Atheneum, and Electronic Boethius archives (using achievements of the ARCHway project: `http://beowulf.engl.uky.edu/~kiernan/ARCHway/entrance.htm`); several corpora based on the NITE XML toolkit (`http://groups.inf.ed.ac.uk/nxt/inuse.shtml`); a multilingual corpus based on EXMARaLDA (`http://www.exmaralda.org/en_korpora.html`).

varying degree of support in state of the art serialization formats. The universe of such features can be summarized as follows:

classic overlap With "classic" we refer to those overlap cases where two document fragments, to be annotated with different general identifiers (that would be tag names for XML), overlaps each other. A simple example was given in Example 1.1. Scenarios exhibiting several occurrences of classic overlap are those where there is the need to identify multiple concurrent hierarchies on the very same document, e.g., the grammatical, phonetic, and typographical structures. When merging such hierarchies in a single document, it is likely that a component of one structure overlaps with a component of another structure, e.g., a paragraph starts within a page and ends within the next page.

self overlap There are also cases in which two components of the same structure, and with the same name, do overlap. For instance consider a text document that should be commented by two distinct reviewers. Suppose those reviewers want to annotate two overlapping text regions. It is reasonable to consider those two overlapping comments as belonging to the same structure (the comment structure). The term "self" overlap is used to refer to these situations, as (assuming an XML syntax) they lead to elements with the same names overlapping each other. Naive handling of such cases make impossible distinguish between overlapping situations and proper nesting, as in: `<comment>John <comment>likes</comment>Mary</comment>`

virtual elements TEI describes *virtual elements* as elements not explicitly present in a text, but whose presence may be inferred by an application from the encoding supplied (Chapter 16 of [6]). Virtual elements are used to define elements whose content is a reordering of material present elsewhere in the document. They are also used to define elements whose content is not a contiguous text region (in such cases they are often referred to as *discontinuous elements*)

containment/dominance decoupling Intuitively, *dominance* is a relation between document parts where one is said to dominate another if it is one of its ancestor in the document structure. *Containment* is rather a look at two document parts from the point of view of which slices of the actual character content of the document they enclose; a document part contains another one if it encloses all the character content of that other part. Tree-based markup languages as XML tend to have the property that containment implies dominance, while this is not necessarily the case. If you think again at multiple hierarchies sharing the same character content, it is not evident at all why structural parts belonging to one hierarchy should be related via dominance to structural parts belonging to another hierarchy.

The XML data model is simply not able to cover such cases. Indeed, the structure of an XML document is inherently a tree. So there is no way to represent within it two overlapping elements, nor virtual elements, nor that a given text region is contained by two elements with no hierarchical relation between them.

2.2 Alternatives to the XML data model

The limitations of tree-like structures in representing complex documents is well-known, and dates back before the XML birth. Indeed, SGML sported an optional (and rarely implemented) feature then removed in XML: CONCUR [16]. SGML CONCUR allows for the representation of multiple (and possibly overlapping) hierarchies within the same document. CONCUR actually extends the classic SGML (and XML) data model, allowing for the co-existence of multiple tree structures within a single document.

In order to overcome the limitations of XML, other data models have been proposed. Sperberg-McQueen and Huitfeldt proposed a directed acyclic graph structure named GODDAG for the representation of text documents [26]. Hilbert and Witt resorted to the CONCUR data model in order to represent concurrent hierarchies all built on the very same sequence of text frontier ("PCDATA" in the XML lingo) [18]. Tennison and Piez presented LMNL, a language for marking up and annotating documents [28]. The LMNL data model is completely unrelated to XML: rather than of element nodes, a LMNL document consists of ranges which may vary either on text characters or on previously defined ranges. LMNL ranges are allowed to overlap. Dekhtyar and Iacob have formally defined the concept of Concurrent Markup Hierarchies (CMH), i.e., a collection of XML documents built on the same text content and sharing the same root element [11]. In another work, the authors made use of GODDAG as the data structure for the representation of their distributed documents [21]. Durusau and O'Donnell illustrated a technique where the text content of a document is divided into atoms (e.g., words) each annotated with assertions about its membership to one or more hierarchies [15].

2.3 Alternatives to the XML notations

A lot of work has also been done on the serialization formats used to encode text documents, and which are also the main topic of the present paper. Some formats relies on an XML notation, while others do not. Documents in an XML-based format have the advantage that any existing XML tool and technology can be used to process them. However the hierarchical structure constructed by an XML parser is forcibly a tree, in order to encode different document structures they need to be coerced into trees using special conventions, and they need post-parsing processing in order to be reconstructed a posteriori.

Moreover, documents with overlapping issues expressed in XML-based formats are neither easy to read, nor to write by humans, as the special elements considerably increase the complexity of the resulting XML structure. The TEI guidelines describes several XML-based encoding schemes to markup complex features of text documents: fragmentation, milestones, stand-off markup, twin documents just to mention the most widespread [6]. Only for LMNL, two XML-based notations do exist, namely CLIX [12] and ECLIX [8]. Durusau proposed to use separate XML documents (related by XPath expressions) to encode the different views of the same document [15]. Similarly, Dekhtyar and Iacob describes an algorithm for the merge of concurrent XML documents into a single XML document, where possible overlapping issues are solved using the fragmentation technique [11].

Documents using non-XML notations cannot use any of the existing XML tools and technologies. Special languages and tools are required to do even the most common actions, such as querying about the structure, presenting the document on a screen, etc. Still, as long as the notation is well designed, the overlapping issues are straightforwardly represented by the notation. In 2001, Huitfeldt and Sperberg-McQueen proposed TexMECS, a meta-markup language whose data-model is defined in terms of GODDAGs [19]. LMNL also defines its own syntax [28]. In 2005, Hilbert and Witt proposed MuLaX a CONCUR-like notation for the encoding of concurrent hierarchies [18].

2.4 Sacred and profane hierarchies

Our conversion framework is not meant to cover all the complex features described in Section 2.1. The main reason is pragmatic, as not all the considered serialization formats support all the features of Section 2.1 we chose to initially concentrate the efforts in providing complete transformations for the largest shared feature subset: classic overlapping markup. This means that for the purpose of this paper we are not interested in dealing with self-overlap, virtual elements, and containment/dominance decoupling. In some particular cases, where translations are between formats both supporting more than classic overlap, the presented algorithms will preserve some additional feature (most notably containment/dominance decoupling), but this is not a general property to expect.

In particular, we consider documents consisting of multiple hierarchies each consisting of *sacred* and *profane* nodes:[2] sacred nodes are shared among all overlapping hierarchies while profane nodes are specific of some hierarchy. Consider the most common cause of overlapping issues: the use

[2]terminology borrowed from Huitfeldt and Sperberg-McQueen after private communications, stretching its original meaning

of two or more independent vocabularies to structure the same content. In such a scenario the markup of the shared content is intended to be sacred, while the markup of each vocabulary is profane. No overlap issues are allowed to arise within sacred markup, nor within a single profane vocabulary, nor within the union of sacred markup and a single profane vocabulary. On the contrary, overlap issues can arise when joining more than one profane vocabulary in the same document.

2.5 Syntax: XML formats

In the remainder of this section, we are going to describe in details the serialization formats (both XML-based and non XML-based) supported by our conversion framework. In order to make clear differences and similarities among them, we will use a running example of a document with overlapping issues, and we show how each format handles them. The example is from the tragedy *La Mort d'Agrippine* by Cyrano de Bergerac [10], that provides two independent structures, the verse structure of lines with a given metre and rhyme, and the performance structure of speeches uttered by the play characters.

Example 2.1. Both structures can be easily expressed in TEI, but they frequently overlap, as shown by the following fragment:

Tibère
Poursuivez...
Agrippine
Quoi, Seigneur?
Tibère
Le propos détestable

Où je vous ai surprise.
Agrippine
Ah! Ce propos damnable
d'une si grande horreur tous mes sens travailla

□

Example 2.2. A (non well-formed) encoding using TEI markup would be as follows:

```
   <TEI>
    <teiHeader>...</teiHeader>
     <text>
      <body>
5      <l>
        <sp><speaker>Tibère</speaker>Poursuivez...</sp>
        <sp><speaker>Agrippine</speaker>Quoi, Seigneur?</sp>
        <sp><speaker>Tibère</speaker>Le propos détestable
       </l>
10     <l>
        où je vous ai surprise.</sp>
        <sp><speaker>Agrippine</speaker>Ah! Ce propos damnable
       </l>
       <l>
15      d'une si grande horreur tous mes sens travailla</sp>
       </l>
      </body>
     </text>
   </TEI>
```

The above example shows two overlapping issues: a speech starting within a line (at line 8) and ending within a different line (at line 11), and a similar issue at lines 12,15. Because of such issues, the above markup is not a well-formed XML document and as such it would be plainly rejected by any XML conformant tool. The Example also shows that although some elements do overlap, others never do. For instance, the elements at the top of the trees are shared among both the verse and the performance substructures, and thus never overlap. For the sake of the following examples, the following elements are assumed to be sacred: `TEI`, `teiHeader`, `text`, `body`, `speaker`; two profane hierarchies are assumed, one containing `l`, the other containing `sp`.

2.5.1 TEI milestones (ECLIX)

In order to encode overlapping structures in XML, the milestone approach is to represent one vocabulary as *primary* with a standard XML hierarchy and to use empty elements to delimit both the start and the end tags of the *secondary* vocabularies. This technique is described in details in the TEI guidelines [5], which also sets the terminology of calling *milestones* the special-purpose empty elements. Note that the primary/secondary distinction never concerns sacred markup parts, as they can never exhibit overlapping issues with neither themselves, nor any profane markup part.

When adopting this approach it is important to distinguish between actual empty elements and empty elements used as milestones. The ECLIX approach [8], a serialization format for LMNL (see Section 2.6.2), proposes one such distinction by inserting special attributes to identify the milestones, as well as to co-index start and end milestones. Other syntactic variants are described in the TEI guidelines.

Example 2.3. An ECLIX serialization of our running example is as follows:

```
1  <TEI>
    ...
    <l>
      <sp>
        <speaker>Tibère</speaker>
6       Poursuivez...
      </sp>
      <sp>
        <speaker>Agrippine</speaker>
        Quoi, Seigneur?
11    </sp>
      <sp clix:role="start-range" clix:sID="sp1" />
        <speaker>Tibère</speaker>
        Le propos détestable
    </l>
16  <l>
        où je vous ai surprise.
      <sp clix:role="end-range" clix:eID="sp1" />
      <sp clix:role="start-range" clix:sID="sp2" />
        <speaker>Agrippine</speaker>
21      Ah ! Ce propos damnable
    </l>
    <l>
        d'une si grande horreur tous mes sens travailla
      <sp clix:role="end-range" clix:eID="sp2" />
26  </l>
    ...
   </TEI>
```

Milestones can be observed at lines 12, 18, 19, 25. Here the verse vocabulary was chosen as the primary, but the performance vocabulary could have been chosen on that role without changing the expressed structure. □

When an element is represented as a pair of milestones its structure is flattened in the XML tree. Thus, although delimiting an element with milestones solves well-formedness issues, it rises processing issues, since special software is now needed to recognize the element and to reconstruct its structure.

ECLIX allows to encode overlapping elements as well as self-overlap (via co-indexing). However it fails to encode other features such as virtual elements. Moreover, as all hierarchies are merged in a single XML document, the lexicographic order between tags and the XML parsing rules might create dominance relationship among profane nodes of different hierarchies, also if such hierarchies are not really meant by the document author. For instance, this is the case of the first `<l>` and the first `<sp>` elements. The only way to decouple dominance relations from containment, is to have external information available in the parsing software.

2.5.2 Flat milestones (CLIX)

In ECLIX milestones are only used to delimit elements of the secondary hierarchy and only when they induce overlapping issues. On the contrary, the CLIX approach [12] expresses *every* element of the document via milestones, primary as well as secondary ones, profane as well as sacred ones, no matter whether milestones were required to solve actual overlapping issues or not.

Example 2.4. Our example can be rewritten in CLIX as follows:

```
<clix:clix>
 <TEI clix:role="start-range" clix:sID="tei01" />
   <teiHeader clix:role="start-range" clix:sID="h01" />
     ...
   <teiHeader clix:role="end-range" clix:eID="h01" />
   <text clix:role="start-range" clix:sID="t01" />
     <body clix:role="start-range" clix:sID="b01" />
       <l clix:role="start-range" clix:sID="l01" />
         <sp clix:role="start-range" clix:sID="sp01" />
           <speaker clix:role="start-range" clix:sID="spk01" />
             Tibère
           <speaker clix:role="end-range" clix:eID="spk01" />
           Poursuivez...
         <sp clix:role="end-range" clix:eID="sp01" />
         <sp clix:role="start-range" clix:sID="sp02" />
           <speaker clix:role="start-range" clix:sID="spk02" />
             Agrippine
           <speaker clix:role="end-range" clix:eID="spk02" />
           Quoi, Seigneur?
         <sp clix:role="end-range" clix:sID="sp02" />
         <sp clix:role="start-range" clix:sID="sp1" />
           <speaker clix:role="start-range" clix:sID="spk03" />
             Tibère
           <speaker clix:role="end-range" clix:eID="spk03" />
           Le propos détestable
       <l clix:role="end-range" clix:eID="l01" />
       <l clix:role="start-range" clix:sID="l02" />
           où je vous ai surprise.
         <sp clix:role="end-range" clix:eID="sp1" />
         <sp clix:role="start-range" clix:sID="sp2" />
           <speaker clix:role="start-range" clix:sID="spk04" />
             Agrippine
           <speaker clix:role="end-range" clix:eID="spk04" />
           Ah ! Ce propos damnable
       <l clix:role="end-range" clix:eID="l02" />
       <l clix:role="start-range" clix:sID="l03" />
           d'une si grande horreur tous mes sens travailla
         <sp clix:role="end-range" clix:eID="sp2" />
       <l clix:role="end-range" clix:eID="l03" />
     <body clix:role="end-range" clix:eID="b01" />
   <text clix:role="end-range" clix:eID="t01" />
 <TEI clix:role="end-range" clix:eID="tei01" />
</clix:clix>
```

□

A CLIX document interpreted as an XML tree is thus completely flat (with the exception of the root node of course): the document is just a sequence of characters interleaved by empty elements. No hierarchy survives in the document and no vocabulary classification or overlapping issue can be determined without specific software.

For what concerns the representational power of CLIX documents, while self-overlapping can be represented thanks to the co-indexing scheme, there is no support for virtual elements. Dominance can be decoupled from containment only by the means of external information, as discussed for ECLIX (though now containment should be obviously only defined in terms of milestone pairs, rather than also in terms of start/end tag pairs).

2.5.3 Fragmentation (or partial elements)

Fragmentation is yet another technique to deal with overlapping described in the TEI guidelines [5]: whenever an element belonging to a secondary vocabulary overlaps with elements belonging to the primary one, it is broken into as many smaller fragments (called *partial elements*) as necessary to solve the overlapping issue.

Syntactic conventions are used to distinguish between partial and non-partial elements, and to recognize the actual fragments of the same element. The technique called *aggregation* is adopted in this paper, whereby each fragment (beside the last one) links to the next fragment via a `@next` attribute.

Example 2.5. Our example can be represented using partial elements as follows:

```
<TEI>
 ...
 <l>
  <sp>
   <speaker>Tibère</speaker>
   Poursuivez...
  </sp>
  <sp>
   <speaker>Agrippine</speaker>
   Quoi, Seigneur?
  </sp>
  <sp xml:id="sp1.1" next="sp1.2">
   <speaker>Tibère</speaker>
   Le propos détestable
  </sp>
 </l>
 <l>
  <sp xml:id="sp1.2">
   où je vous ai surprise.
  </sp>
  <sp xml:id="sp2.1" next="sp2.2">
   <speaker>Agrippine</speaker>
   Ah ! Ce propos damnable
  </sp>
 </l>
 <l>
  <sp xml:id="sp2.2">
   d'une si grande horreur tous mes sens travailla
  </sp>
 </l>
 ...
</TEI>
```

□

Partial elements and aggregation allow to represent complex features for text documents. Self-overlap can be implemented thanks to the pointing scheme. Discontinuous elements can easily be implemented, as the region of content between two partial elements is not part of the fragmented element. There are even more expressive techniques to implement aggregation (see the `<join>` element of the TEI guidelines [5]), which allow to create virtual elements whose content is a reordering of elements defined elsewhere within the document. As seen for ECLIX, the use of a single XML document to encode the different hierarchies might lead to the construction of dominance relationships among profane elements, not meant by the document author.

2.5.4 Twin documents

By *twin documents* we mean a collection of XML documents sharing the same sacred markup content interspersed with distinct pieces of profane markup. This approach has been presented in the past using different names, such as distributed documents [11], multiple encodings of the same information [5], and redundant encoding in multiple forms [30].

Example 2.6. Our example can be serialized into two twin documents: one for the verse vocabulary, one for the performance vocabulary. The result is as follows:

```
<TEI>    <!-- verse vocabulary -->
 <teiHeader>...</teiHeader>
 <text>
  <body>
   <l>
    <speaker>Tibère</speaker>
    Poursuivez...
    <speaker>Agrippine</speaker>
    Quoi, Seigneur?
    <speaker>Tibère</speaker>
    Le propos détestable
   </l>
   <l>
    où je vous ai surprise.
    <speaker>Agrippine</speaker>
    Ah ! Ce propos damnable
   </l>
   <l>
    d'une si grande horreur tous mes sens travailla
   </l>
  </body>
 </text>
</TEI>

<TEI>    <!-- performance vocabulary -->
 <teiHeader>...</teiHeader>
 <text>
  <body>
   <sp>
    <speaker>Tibère</speaker>
    Poursuivez...
   </sp>
   <sp>
    <speaker>Agrippine</speaker>
    Quoi, Seigneur?
   </sp>
   <sp>
    <speaker>Tibère</speaker>
    Le propos détestable
    où je vous ai surprise.
   </sp>
   <sp>
    <speaker>Agrippine</speaker>
    Ah ! Ce propos damnable
    d'une si grande horreur tous mes sens travailla
   </sp>
  </body>
 </text>
</TEI>
```

□

In twin documents each hierarchy is a distinct XML document, denoting its own tree structure. No distinction therefore is necessary between primary and secondary hierarchy, and no special tool is required to examine each hierarchy separately. The disadvantages of the approach is that multiple copies of the same sacred markup have to be maintained and that relationships among profane hierarchies cannot be observed looking at a single document.

It is worth observing that as the profane elements are stored in distinct documents, it is not always possible to establish a total order among their start and end tags. For instance the first `<l>` element (stored within the first document above) and the first `<sp>` element (stored within the second document) both occurs between the `<body>` and `<speaker>` sacred elements, and even at the very same byte count from the beginning of the respective documents. In this case it is not possible to assert which start tag precedes the other. Hence if we would want to merge the two documents into a single document, we have at least two possible ways to do that. However it also means that no unwanted dominance hierarchies among profane elements of different hierarchies are implicitly encoded in the serialization.

From a representational power standpoint, twin documents do not allow to represent virtual elements. Moreover, assuming that the profane vocabularies are disjoint, it is not possible to encode self-overlap situations.

2.5.5 Stand-off markup

The stand-off markup was initially described in the TEI guidelines [6]; later on it has been adopted in various forms by other projects and corpora [7, 31]. The idea is to have a *source* document (which is either an XML or a plain text document) and one or more *external* XML documents with their own markup and which reference portions of the source document using some pointing mechanism. Each external document may be seen as an independent (profane) hierarchy over the source document which contains all the sacred markup. Actually TEI allows to store the stand-off markup within the source document itself into special sections. As suggested by TEI it is possible to transform (*externalize*) a single XML document of such a kind into two or more documents, one acting as the source document and the others as external documents. So we will assume without loss of generality to always work with source and external documents.

As the source document is not aware of the external documents, in stand-off markup a sacred element is not allowed to contain profane markup. With our example, this means that we cannot store the sacred elements `<TEI>`, `<teiHeader>`, `<text>`, and `<body>` within the source document.

Example 2.7. A sample source document for our example is:

```
<root>
 <speaker xml:id="spk01">Tibère</speaker>
 Poursuivez...
 <speaker xml:id="spk02">Agrippine</speaker>
 Quoi, Seigneur?
 <speaker xml:id="spk03">Tibère</speaker>
 Le propos détestable
 où je vous ai surprise.
 <speaker xml:id="spk04">Agrippine</speaker>
 Ah ! Ce propos damnable
 d'une si grande horreur tous mes sens travailla
</root>
```

Given the above source, sample external documents are:

```
<TEI>    <!-- verse vocabulary -->
 ...
 <l>
  <xinclude href="source.xml"
    xpointer="xpath1(id('spk01'))" />
  <xinclude href="source.xml"
    xpointer="string-range(xpath1(id('spk01')), 0, 13)" />
  <xinclude href="source.xml"
    xpointer="xpath1(id('spk02'))" />
  <xinclude href="source.xml"
    xpointer="string-range(xpath1(id('spk02')), 0, 15)" />
  <xinclude href="source.xml"
    xpointer="xpath1(id('spk03'))" />
  <xinclude href="source.xml"
    xpointer="string-range(xpath1(id('spk03')), 0, 19)" />
 </l>
 <l>
  <xinclude href="source.xml"
    xpointer="string-range(xpath1(id('spk03')), 20, 23)" />
  <xinclude href="source.xml"
    xpointer="xpath1(id('spk04'))" />
  <xinclude href="source.xml"
    xpointer="string-range(xpath1(id('spk04')), 0, 23)" />
 </l>
 <l>
  <xinclude href="source.xml"
    xpointer="string-range(xpath1(id('spk04')), 24, 47)" />
 </l>
 ...
</TEI>

<TEI>    <!-- performance vocabulary -->
 ...
 <sp>
  <xinclude href="source.xml"
    xpointer="xpath1(id('spk01'))" />
  <xinclude href="source.xml"
    xpointer="string-range(xpath1(id('spk03')), 0, 13)" />
 </sp>
 <sp>
  <xinclude href="source.xml"
    xpointer="xpath1(id('spk02'))" />
  <xinclude href="source.xml"
    xpointer="string-range(xpath1(id('spk02')), 0, 15)" />
 </sp>
 <sp>
  <xinclude href="source.xml"
    xpointer="xpath1(xpath1(id('spk03'))" />
  <xinclude href="source.xml"
    xpointer="string-range(xpath1(id('spk03'))), 0, 42)" />
 </sp>
 <sp>
  <xinclude href="source.xml"
    xpointer="xpath1(id('spk04'))" />
  <xinclude href="source.xml"
    xpointer="string-range(xpath1(id('spk04')), 0, 70)" />
 </sp>
 ...
</TEI>
```

□

As suggested by TEI, we used XPointer to reference the nodes and characters of the source. It is worth noting that when using a powerful pointing mechanism—as XPointer is—it is possible to create discontinuous elements within the external documents. Assuming, that sacred and profane vocabularies are disjoint and that external documents are allowed to point to the source document only, stand-off markup does not allow to encode self-overlap situations. Similarly to what happen with twin documents, the use of external documents avoids the creation of unwanted dominance relationships among profane nodes of different hierarchies.

2.6 Syntax: non-XML formats

Non XML-based syntaxes aim at using a faithful encoding of competing hierarchies even if this means abandoning standard technologies and the associated tools. This paper addresses two such syntaxes: TexMECS and LMNL.

2.6.1 TexMECS

TexMECS is a markup meta-language designed to encode complex text documents [19]. As XML, TexMECS defines the concept of *elements*, which are delimited by *start-* and *end-tags*. However, TexMECS representational power goes well beyond XML. Indeed, besides allowing to represent tree structures, TexMECS allows to represent graph structures, where start- and end-tags are not required to properly nest. In fact, TexMECS specs explicitly references GODDAG as the appropriate data structure to represent a (parsed) TexMECS document. TexMECS supports:

self overlap Using a simple co-indexing scheme, it is possible to encode situations where two elements with the same generic identifier overlap.

virtual elements Using a reference mechanism and special delimiters, it is possible to make an element point to another element: the pointing element (called the virtual element) inherits the pointed element's content. Through virtual elements it is possible to define two elements sharing the same content but without any dominance relationship between them. Also *discontinuous elements* can be encoded using special delimiters. With them one can interrupt an element, and resume it later on.

unordered content It is possible to mark the content of an element as unordered. On such elements children reordering is not an operation which affect document semantics.

This paper focuses on a subset of TexMECS, in particular the advanced features of the above list are not considered. When such a restriction is in effect, TexMECS syntax has some similarities with XML. Indeed, start- and end-tags delimiters are `<`, `>` (both used in XML), and `|`. For instance, in order to represent the document in Example 1.1, we might write the TexMECS document shown in Example 2.8.

Example 2.8. `<doc|<b|John <i|likes|b> Mary|i>|doc>`

This is a well-formed TexMECS document. Elements `b` and `i` do not properly nest, and it is allowed by TexMECS. □

In TexMECS, when two elements properly nest a dominance relationship between them is inferred, exactly as it happens for XML. In the considered TexMECS subset (where virtual elements are excluded), this means that unwanted hierarchical relationships can be built.

Example 2.9. TexMECS encoding of our *La Mort d'Agrippine* excerpt.

```
<TEI|
 <teiHeader|...|teiHeader>
 <text|
  <body|
   <l|
    <sp|<speaker|Tibère|speaker>Poursuivez...|sp>
    <sp|<speaker|Agrippine|speaker>Quoi, Seigneur?|sp>
    <sp|<speaker|Tibère|speaker>Le propos détestable
   |l>
   <l|
    où je vous ai surprise.|sp>
    <sp|<speaker|Agrippine|speaker>Ah! Ce propos damnable
   |l>
   <l|
    d'une si grande horreur tous mes sens travailla|sp>
   |l>
  |body>
 |text>
|TEI>
```

According to TexMECS semantics, there is a father-child relationship among the first `<l>` element and the first two `<sp>` elements. We already observed that such relationships is not necessarily meaningful, as `<l>` and `<sp>` elements belong to distinct profane hierarchies. However, ignoring the TexMECS features described early in this section (and in particular, ignoring virtual elements), in TexMECS there is no way to eliminate such dominance relationships.[3] □

2.6.2 LMNL

LMNL (the *Layered Markup and Annotation Language*) is an approach to text encoding based on layered ranges which can overlap each other. LMNL main contribution is a data model [28], rather than a syntax. However, at least three syntaxes have been proposed: two XML-based syntaxes which we have already touched—CLIX and ECLIX—and a non-XML syntax known as the LMNL syntax.

Briefly, a LMNL document is a set of layers where each layer is made of either a sequence of characters (the *text layer*) or a sequence of ranges. Ranges mimic the homonymous mathematic intuition and are defined on top of a previously defined layer which should be listed as the base of the ranges' layer. A layer can be the base for several other layers, but should be based on a single other layer; the text layer has no base. LMNL also introduces other concepts such as *annotations*, which are outside the scope of this paper. For an exhaustive discussion on LMNL, the best resource is the official LMNL website `http://lmnl.net/`.[4]

The LMNL data model captures classic overlapping cases, as ranges are allowed to overlap. Also self-overlap cases are covered, as each range has a label, and nothing prohibits two (possibly overlapping) ranges to share the same label. The separation into layers allows for the definition of ranges entirely containing other ranges, but without any dominance relation between them. A LMNL range varies over continuous sequence of characters (or other ranges). Thus, it is not

[3]To be more precise, it is possible to avoid the dominance relationship between the first `<l>` element and the first `<sp>` element, moving the start-tag of `<l>` after the start-tag of `<sp>`, thus creating an overlapping issue. However, the same trick cannot be adopted also for the `<sp>` element, as there is meaningful text content between those two elements.

[4]Actually, the data model described in `http://lmnl.net/` was defined as "partially obsolete" by Cowan in 2006 [9]. However, at the time of writing there is no official definition of an up to date data model. So we are taking into account only the data model currently described by the LMNL website.

possible to represent discontinuous text components. Moreover, a range is currently not allowed to dominate both characters and other ranges (mixed content).

Example 2.10. The running example can be serialized into LMNL syntax as follows:

```
[!layer name="l1" base="l2"]
[!layer name="l2" base="l3"]
[!layer name="l3" base="#default"]
[TEI~l1}
 [teiHeader~l2}...{teiHeader~l2]
 [text~l2}
  [body~l3}
   [l}
    [sp}[speaker}Tibère{speaker]Poursuivez...{sp]
    [sp}[speaker}Agrippine{speaker]Quoi,Seigneur?{sp]
    [sp}[speaker}Tibère{speaker]Le propos détestable
   {l]
   [l}
    où je vous ai surprise.{sp]
    [sp}[speaker}Agrippine{speaker]Ah! Ce propos damnable
   {l]
   [l}
    d'une si grande horreur tous mes sens travailla{sp]
   {l]
  {body~l3]
 {text~l2]
{TEI~l1]
```

The *TEI* range dominates both the *teiHeader* and *text* ranges; the *text* range dominates the *body* range only; the *body* range varies over the *l*, *sp*, and *speaker* ranges, which in turn all vary over the default text layer. Note that, as LMNL does not allow to define ranges varying over both other ranges and characters, in writing a LMNL document of the running example, we had two possibilities: either make *l* and *sp* ranges varying over *speaker* ranges, or make them varying over characters. The document above implements the latter possibility: though such a choice means that no dominance relationship does exist between *sp* and *speaker* ranges (and nor between *l* and *speaker* ranges), it at least allows to recognize the text content as dominated by *l* and *sp* ranges. □

2.7 Semantics: GODDAG

The semantics of documents with overlapping issues we want to grasp is the hierarchical structures of the involved vocabularies and how element boundaries intersperse with textual content. GODDAG is an agreed upon family of data structures for representing overlapping hierarchies [26] which fit our needs.

A GODDAG is a DAG (Direct Acyclic Graph) with no transitive arcs, where both XML elements and textual content are represented as nodes. Arc denotes father-child relationships. Multi-parentage is allowed and is exploited to represent overlapping issues. Text content can hence be shared among nodes and is split as needed at element boundaries: in a GODDAG representation of Example 1.1 we will have three text nodes (or *leaf nodes*), one (containing "John") child of `b`, one (containing "likes") child of both `i` and `b`, and one (containing "Mary") child of `i`.

Several kind of GODDAG has been defined, in [26] authors define: generalized, restricted, and clean GODDAGs. The same authors discuss about normalized and colored GODDAGs in [20]. The form we are concerned with for the conversions is that of *restricted* GODDAGs; for the sake of readability we repeat its definition here.

Definition 2.1 (restricted GODDAG)**.** A GODDAG is restricted if the following conditions hold:

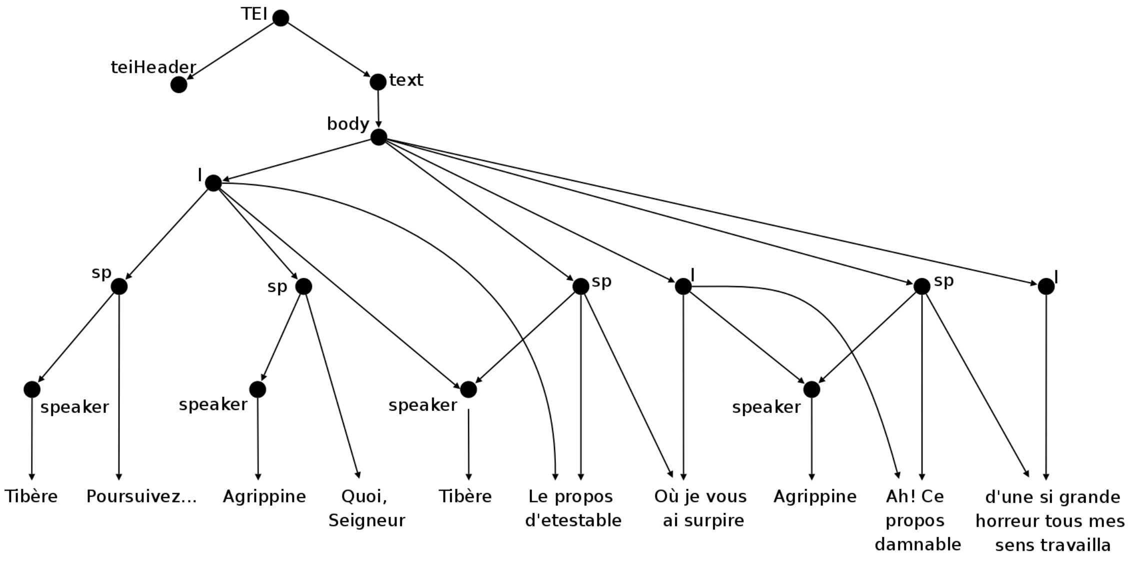

Figure 1: A restricted GODDAG corresponding to Example 2.2. Empty leaf nodes, which guarantee that the GODDAG is restricted, are not shown to keep the figure terse.

1. leaf nodes form a sequence (called *frontier*) and are totally ordered by the $\prec$ relation (document order); $\prec$ is trichotomus on leaf nodes, that is: $\forall l_1, l_2$ leaf nodes: either $l_1 \prec l_2$, or $l_2 \prec l_1$, or $l_1 = l_2$;

2. each node n dominates a contiguous sub-sequence of the frontier (called n's *leafset*), that is: if n dominates leaf nodes l_1, l_2 then $\forall l$ leaf node such that $l_1 \prec l \prec l_2$, n dominates l;

3. no two nodes dominate the same frontier sub-sequence, that is: $\forall n_1, n_2$ nodes, there exists a leaf node l such that either n_1 dominates l and n_2 does not dominate it, or vice versa.

Clearly the second constraint (relaxed in *generalized GODDAG*) rules out the possibility of representing discontinuous elements. This is not an invalidating restriction, as we are not interested in dealing with discontinuities. The third constraints avoids to represent the situation of two elements in distinct hierarchies and dominating exactly the same text region. It also prohibits to have a node with a single outgoing arc, pointing to a non-leaf node. As suggested by the proposers, such cases can be handled creating a leaf node labeled with the empty string and adding it to one of the two nodes.

For what concerns the first constraint, we do not perceive it as a limitation, but rather as a property. It allows us to define an order on the outgoing arcs of a node as follows.

Definition 2.2 (Arc ordering)**.** Let n be a non-leaf node and let $n_1, \ldots, n_k$ be its children. Let l_i be the first leaf node dominated by n_i, for $1 \leq i \leq k$. Then $n \rightarrow n_i \leq n \rightarrow n_j$ (or equivalently $n_i \leq n_j$) iff $l_i \prec l_j$.

Though not explicitly stated in the original definition, we assume that every restricted GODDAG has a single root element, assumption shared by other works in the literature that have made use of GODDAGs [21].

The choice of restricted GODDAGs for the conversion framework does not allows to represent some features of the participating serialization formats, such as TexMECS virtual and discontinuous elements. However, for the features we are interested to cover in our conversion framework, we believe restricted GODDAGs represent an appropriate data structure.

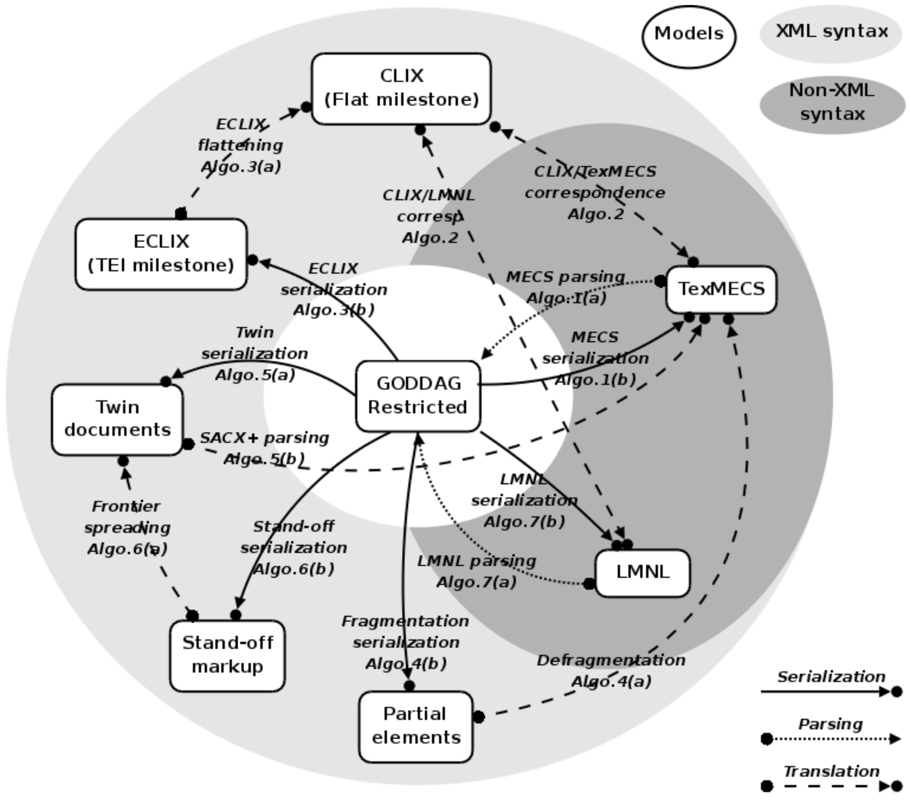
Models
XML syntax
Non-XML syntax
CLIX (Flat milestone)
ECLIX flattening Algo.3(a)
ECLIX (TEI milestone)
CLIX/LMNL corresp. Algo.2
CLIX/TexMECS correspondence Algo.2
ECLIX serialization Algo.3(b)
MECS parsing Algo.1(a)
TexMECS
Twin serialization Algo.5(a)
MECS serialization Algo.1(b)
GODDAG Restricted
Twin documents
SACX+ parsing Algo.5(b)
Frontier spreading Algo.6(a)
Stand-off serialization Algo.6(b)
LMNL serialization Algo.7(b)
LMNL parsing Algo.7(a)
LMNL
Stand-off markup
Fragmentation serialization Algo.4(b)
Defragmentation Algo.4(a)
Partial elements
Serialization
Parsing
Translation

Figure 2: Architecture of the conversion framework and conversion paths.

In Figure 1 a restricted GODDAG of the running example is shown.[5] It has a speaker node containing "Tibère"; that element, which sits in the middle of an overlapping issue, has two parents: a speech node and a line node. Note also that GODDAG has dominance relationship between some line and speech nodes. Though it is possible to show a restricted GODDAG with no such relationships, we preferred to show the GODDAG of Figure 1 to remark that in the conversion framework there are formats, such as TexMECS, with no ability to fully control the dominance relationships between nodes.

3 A framework for format translations

The framework we have designed enables mutual conversions among seven serialization formats for documents with overlapping markup issues, namely: TexMECS (non-XML), CLIX, LMNL (non-XML), TEI milestones (or ECLIX), partial elements (or fragmentation), twin documents, stand-off markup. As always when having to deal with all-pairs issues, we have to choose among developing $7 \cdot (7 - 1)$ all-pairs conversion algorithms and using a "star"-architecture where all format instances are first converted to a central format (not necessarily concrete, it might well be an abstract model or a data structure) and then serialized to target format instances relying only on the information available in the central format representation.

The proposed translation framework, whose architecture is depicted in Figure 2, implements a star-architecture; the natural choice for its center is the restricted GODDAG[6] data structure;

[5]To be more precise, that GODDAG is not actually restricted as there are nodes (i.e., *text* and *body*) dominating the same frontier subsequence. However, the restrict-ness of the GODDAG can simply be achieved assuming the presence of leaf nodes labeled within the empty string, they are not shown for the sake of readability.

[6]From now on, unless otherwise specified we will use the term "GODDAG" to refer to *restricted* GODDAGs

Section 2.7 discusses how it is a proper model to grasp the details of the largest subset of features shared by the serialization formats we are concerned with.

Figure 2 also shows the conversion paths among serialization formats with references to all the algorithms presented in the present paper. In many cases there are multiple possible paths to implement a given conversion and different properties might be preserved by different paths. They are detailed in the algorithm descriptions later on. Of course, implementation of conversion paths as single algorithms is possible and in some cases can lead to better performances, but we have refrained to do so in the paper for the sake of clarity and code modularity.

We do not present a formal API for working directly on the GODDAG data structure and as such we only assume a basic set of graph APIs; an early proposal for a DOM-like API for GODDAG can be found in [21] and can be exploited in the implementation of the proposed algorithms.

Translation algorithms. In the remainder of this section we present the conversion algorithms which constitute the core of the translation framework. The presentation is format-by-format. Usually for each format we will present an algorithm for parsing the given format into a GODDAG (directly or not) and one for serializing back from GODDAG to the given format.

Syntactic conventions. Algorithms are presented using an object-oriented-like pseudo code, the syntax of which we believe to be mostly self-explanatory: properties are accessed as "o.property" and set as "o.property $\leftarrow$ *value*", variables are set similarly and accessed by their names alone; methods are invoked as "o.method($\text{arg}_1, \ldots, \text{arg}_n$)". Auxiliary sub-routines not strictly relevant for the understanding of algorithms have been postponed until Appendix A, while domain-specific abstract routines which need to be externally defined to actually implement a given algorithm are "underlined()" and described in the prose.

3.1 Translation via TexMECS

Parsing. The first serialization format we consider is TexMECS (see Section 2.6.1).

Algorithm 1(a) parses a stream of TexMECS events and returns a GODDAG instance as output. The TexMECS stream is extracted from an event parser for TexMECS documents; possible events are:

start_tag generated when the opening tag of an element is encountered; equipped with the element name;

end_tag generated when the closing tag of an element is encountered; equipped with the element name;

cdata generated when some character chunk is encountered; equipped with the actual character content (a string). Chunks are delivered as large as possible using other kind of events as boundaries. No two non-*cdata* events can be generated in a row: they will always be intermixed with a *cdata* event, possibly containing the empty string.

For the sake of conciseness, for TexMECS event streams, and later on also for XML event streams, we do not treat specially empty elements (which are assumed to correspond to three-event bursts: *start_tag*(n), *cdata*(`""`), *end_tag*(n) for some element name n) and do not consider attributes (other than those used for linking purposes, as needed by some serialization format) at all. Each of the presented algorithms can be trivially extended to support general purpose attributes and empty elements. Technically, we also assume that events are equipped with unique identifiers which will be recorded in GODDAG nodes and which can be used to find correspondences between matching start/end tag (or range) events (a feature commonly implemented by event parsers).

Algorithm 1(a) keeps the usual stack of currently open tags (O), but of course admits popping tags other than the topmost: that situation will occur (and will be detected at line 17) exactly once for each overlapping issue occurring in the input document D. The intuition behind the parsing algorithm is to add arcs between nodes *eagerly* (line 7), as soon as *potential* ancestor/descendant

Algorithm 1 TexMECS $\leftrightarrow$ GODDAG

(a)

```
    TEXMECS2GODDAG($D$)
      $E \leftarrow$ event_parser($D$)
      $G \leftarrow$ goddag(), $O \leftarrow \emptyset$
      while $e \leftarrow E$.next_event() do
        if $e = start\_tag$ then
 5:       $n \leftarrow G$.add_element($e$)
          for all $p \in O$ do
            $G$.add_arc($p \rightarrow n$)
          end for
          $O$.push($n$)
10:     else if $e = cdata$ then
          $n \leftarrow G$.add_text($e$)
          for all $p \in O$ do
            $G$.add_arc($p \rightarrow n$)
          end for
15:     else if $e = end\_tag$ then
          $n \leftarrow O$.rm_peer($e$)
          for all $p \in O, p > n$ do
            $G$.rm_arc($p \rightarrow n$)
          end for
20:     end if
      end while
      $G \leftarrow$ TRED$^t$($G$) /* transitive reduction */
      return G
```

(b)

```
    GODDAG2TEXMECS(G)
      $E \leftarrow$ event_parser($G$.EVENTS())
      while $e \leftarrow$ E.next_event() do
        $n \leftarrow e$.node
        if $e = start\_tag$ then
          print "<$n$.tag|"
        else if $e = cdata$ then
          print $n$.content
        else if $e = end\_tag$ then
          print "|$n$.tag>"
        end if
      end while
```

relationships are spotted. When it becomes clear that a previously added arc is bogus, and it will be the case each time an overlapping issue is found, the corresponding arc will be removed (line 18).

However, to ensure a proper GODDAG is returned, transitive arcs should be also removed (line 22); this can be done at once before returning the GODDAG, by efficient well-known algorithms performing transitive closure [17] (not reported in the paper). It is worth noting that before transitive arc removal, the amount of non-bogus arcs in the GODDAG is maximal and that those arcs fully embody GODDAG dominance relation.

In fact, Algorithm 1(a) is a reformulation of an algorithm developed for the very same purpose by the initial proposers of TexMECS and GODDAG in [26], it is reported here for ease of reference. As a minor improvement over the original formulation, ours is more efficient thanks to the postponing of the transitive reduction phase.

Serialization. Algorithm 1(b) does the converse transformation: from GODDAG to TexMECS. Its core is actually implemented by the EVENTS routine (see Appendix A) which returns the stream of TexMECS events matching a GODDAG; having that, it's trivial to serialize them using the appropriate syntax. EVENTS is a generic routine we will be using in several other algorithms later on and it is implemented in 2 phases. The first one, implemented by ADDLEAFTAGS, annotates each leaf node of the frontier with the set of nodes that start just before ("left" property) and end just after ("right" property) the leaf. The second phase, implemented by EVENTS itself, just traverses the frontier outputting[7] tags and textual content as needed.

Both EVENTS and ADDLEAFTAGS relies on the assumption that GODDAG *internal* nodes are totally ordered (on such an order functions like "min" and "sort" are in turn supposed to rely).

[7]we use **print** to output markup chunk, you can think at it either as print of text to be parsed later on, or as direct event emission

This property is not directly granted by the definition of restricted GODDAG, but is inherited from it (see Definition 2.2).

3.2 Translation via CLIX

We then consider the CLIX serialization format (see Section 2.5.2). We observe that a CLIX subset that can be expressed as a (restricted) GODDAG is isomorphic to the TexMECS obeying the same requirements. Indeed CLIX start milestones play the same role of TexMECS start tag, end milestones those of end start tags, and character data are represented as themselves in both formats. Similar considerations can be done to relate CLIX milestones to LMNL syntax, as CLIX has been designed precisely as an XML syntax to canonically represent instances of the LMNL data model.

Algorithm 2 CLIX $\leftrightarrow$ TexMECS and CLIX $\leftrightarrow$ LMNL correspondences

TexMECS	$\leftrightarrow$	CLIX	$\leftrightarrow$	LMNL	
`<`n`|`	$\leftrightarrow$	`<`n` clix:role="start-range" />`	$\leftrightarrow$	`[`n`}`	start tag/range
`|`n`>`	$\leftrightarrow$	`<`n` clix:role="end-range" />`	$\leftrightarrow$	`{`n`]`	end tag/range
t	$\leftrightarrow$	t	$\leftrightarrow$	t	character data

As a consequence, translations from/to CLIX can be performed indirectly passing from either TexMECS or LMNL syntax. The correspondences relating CLIX markup to the markup of the former two formats are given in Algorithm 2 which can indeed be seen as an algorithm to convert events streams of one format to the others. Note that since the correspondences are on markup, rather then between markup and data model, range ordering issues for LMNL [12] do not matter and relative order of markup atoms will be preserved by the conversions.

Of course we are not supporting LMNL annotations in CLIX (nor we will in ECLIX), but we are able to support LMNL layers with direct conversions from/to GODDAG (see Section 3.7).

3.3 Translation via ECLIX

We now consider the translations involving ECLIX (see Section 2.5.1), or TEI milestones, which is roughly speaking an extended version of CLIX where milestones are not necessarily used for all elements, and legacy XML tags can be used as long as they do not introduce overlapping issues.

Parsing. Developing an ad-hoc algorithm for parsing ECLIX to GODDAG is moot, as we already have a parsing algorithm for CLIX and that ECLIX can be converted to CLIX expanding non-milestoned elements to milestone pairs. The latter is precisely what Algorithm 3(a) does. Some care need to be used when processing end tags, since empty XML elements are represented as start/end tag pairs but we do *not* want to generate bogus end milestones when a start milestone is represented as a pair of start/end tags. To do so at line 11 our algorithm is careful about avoiding to output bogus end milestones, it avoids outputting them if the CLIX role attributed to the peer start tag was `start-range`.

Serialization ideally should not be any harder that TexMECS serialization, except for the syntactical differences. However for a given document several alternative ECLIX serialization do exists that when parsed back with Algorithm 3(a) return the starting GODDAG. In particular, for documents with at least one *overlapping issue* (i.e. a pair of elements whose leafsets L_1, L_2 are such that $L_1 \neq L_2$, $L_1 \not\subset L_2$, and $L_2 \not\subset L_1$), each overlapping issue induces a choice between two possible serialization of involved elements.

Example 3.1. For instance, in Example 1.1 a single overlapping issue is present and it is among `b` and `i`. Both element cannot be serialized as legacy XML elements at once without introducing a single point of non well-formedness in the resulting document. To ensure well-formedness, one

Algorithm 3 ECLIX ↔ GODDAG

(a)

```
ECLIX2CLIX($D$)
  $E \leftarrow$ event_parser($D$)
  $O \leftarrow \emptyset$
  while $e \leftarrow E$.next_event() do
    if $e$ = start_tag then
5:    print "<$e$.tag sID='$e$.id'/>"
      $O$.push($e$)
    else if $e$ = cdata then
      print $e$.content
    else if $e$ = end_tag then
10:   $e' \leftarrow O$.pop()
      $r \leftarrow e'$.attr("clix:role")
      if $r \neq$ "start-range" then
        print "<$e$.tag eID='$e$.id'/>"
      end if
15: end if
  end while
```

(b)

```
GODDAG2ECLIX(G)
  $G$.MARKMST()
  $E \leftarrow$ event_parser($G$.EVENTS())
  while $e \leftarrow$ E.next_event() do
    $n \leftarrow e$.node
    if $e$ = start_tag then
      if $n$.is_mst then
        print "<$n$.tag sID='$n$.id'/>"
      else
        print "<$n$.tag>"
      end if
    else if $e$ = cdata then
      print $n$.content
    else if $e$ = end_tag then
      if $n$.is_mst then
        print "<$n$.tag eID='$n$.id'/>"
      else
        print "</$n$.tag>"
      end if
    end if
  end while
```

of the two should be output as a pair of milestones and the other as a legacy XML element; the choice discriminates among two alternative ECLIX serializations:

1. `<doc><b sID="1"/>John <i>likes<b eID="1"/>Mary</i></doc>`

2. `<doc><b>John <i sID="1"/>likes</b>Mary<i eID="1"/></doc>`

Actually even both elements can be serialized as milestone pairs, but the whole point of ECLIX is to preserve "real" XML hierarchies as long as possible in order to exploit the tree model with legacy XML software tools. □

In order to serialize to TexMECS all such choices need to be made. Our serialization algorithm (Algorithm 3(b)) is hence abstracted over a function called choose_xml(n_1, n_2) which should return a pair containing first the node chosen as to be represented by a legacy XML element and then the other. Technically, Algorithm 3(b) delegate choose_xml invocation to MARKMST (see Appendix A) which uses a stack to spot all overlapping issues, make a choice for each of them, and mark as milestones (using the "is_mst" property) the nodes which have *not* been chosen. Later on a fully determined ECLIX serialization can be output querying the nodes' is_mst property during a traversal similar to what we used in Algorithm 3(b).

We believe that the overlapping issue resolution logics should not be implemented making an arbitrary choice (e.g. "always the rightmost element" as was done in [12]) but rather exploiting some domain specific knowledge in order to preserve as much as possible of the hierarchy that benefit most of processing with legacy XML tools. However, it is of course possible to implement choose_xml as always returning the same node if that is the desired behaviour.

Finally, it is worth observing that with milestones, choosing that an element should not be output as legacy XML solves all future overlapping issues involving that element. Since past choices are recorded into nodes, choose_xml can exploit this property implementing policies which check the value of is_mst before deciding, possibly avoiding to output unneeded milestones.

3.4 Translation via partial elements

We now turn our attention to partial elements (see Section 2.5.3), also known as the *fragmentation* technique, where an overlapping issue among elements e_1 and e_2 is solved by splitting into several partial elements one of the elements involved in the issue, say e_1, using the tag of e_2 contained in e_1 as splitting point (exactly one of the tag of e_2 should be contained in e_1, otherwise it could not have been an overlapping issue in the first place).

Algorithm 4 Partial elements $\leftrightarrow$ GODDAG

(a)

```
PARTIAL2TEXMECS($D$)
  $E \leftarrow$ event_parser($D$)
  $O \leftarrow \emptyset$ ; $O' \leftarrow \emptyset$
  while $e \leftarrow E$.next_event() do
    if $e = start\_tag$ then
      $O$.push($e$)
      if $\neg$REFERRED($e, O'$) then
        print "<$e$.tag|"
        $O'$.push($e$)
      else if $e$.has_attr("next") then
        $id \leftarrow e$.attr("next")
        $e' \leftarrow O'$.get_peer($e$)
        $e'$.set_attr("next", $id$)
      end if
    else if $e = cdata$ then
      print $e$.content
    else if $e = end\_tag$ then
      $e' \leftarrow O$.pop()
      if $\neg e'$.has_attr("next") then
        print "|$e$.tag>"
        $O'$.rm_peer($e$)
      end if
    end if
  end while

CLOSEFRAGS($F$)
  for all $f \in F$ do
    print "</$f$.tag>"
  end for
```

(b)

```
GODDAG2PARTIAL(G)
  $G$.MARKFRAGS()
  $E \leftarrow$ event_parser($G$.EVENTS())
  while $e \leftarrow$ E.next_event() do
    $n \leftarrow e$.node
    if $e = start\_tag$ then
      CLOSEFRAGS(rev(sort($n$.s_brk)))
      if $n$.is_frag then
        $nid \leftarrow$ new_id($n$)
        print "<$n$.tag next='$nid$'>"
      else
        print "<$n$.tag>"
      end if
      REOPENFRAGS(sort($n$.s_brk)
    else if $e = cdata$ then
      print $n$.content
    else if $e = end\_tag$ then
      CLOSEFRAGS(rev(sort($n$.e_brk)))
      print "</$n$.tag>"
      REOPENFRAGS(sort($n$.e_brk)
    end if
  end while

REOPENFRAGS($F$)
  for all $f \in F$ do
    $id \leftarrow$ cur_id($f$)
    $nid \leftarrow$ new_id($f$)
    print "<$f$.tag id='$id$' next='$nid$'>"
  end for
```

Parsing. We observe that the continuity property of partial elements matches the property of restricted GODDAGs that each node dominates a contiguous frontier sub-sequence, hence it would be impossible to translate non contiguous partial elements to a restricted GODDAG. For this reason our parsing algorithm, presented as Algorithm 4(a), assumes that the input document satisfies continuity (though it can be easily patched to *ensure* that). Similarly, this serialization algorithm will only generate documents preserving continuity.

Parsing is performed generating a TexMECS event stream out of a partial element document (i.e. a document using the fragmentation technique). The only tricky point is detecting when a tag is not to be output since it stands for some inner machinery of partial elements rather than for an actual element boundary. This is the case for a start tag when a previous start tag references it via the `next` attribute, and for an end tag when its peer start tag has a `next` (i.e. this is not the last fragment). To do the housekeeping of past fragments referencing future ones the usual open tag stack it is not enough, since partial element documents are well-formed XML documents and therefore past fragments get popped out of the stack, hence we use an extra stack (line 3) which deals with the conceptual TexMECS stack and always keep a fresh reference to the next fragment identifier which has to be encountered.

Serialization to partial element documents is implemented by Algorithm 4(b). It is similar to Algorithm 3(b) for what concerns the choices of how to solve overlapping issues, with markFrags here being the counterpart of markMst there. A noteworthy difference, inherited from the serialization format, is that while the choice of "milestoning" a logic element globally solves all the overlapping issues involving that element, with partial elements it solves a single overlapping issue. Consider again Example 3.1, the choice of using partial elements for, say, `<b>` will only have as a consequence that the start tag `<i>` behaves as a *breakpoint* for `b`, but other overlapping issues can still involve either of its two fragments split by `<i>`. Generalizing, a single overlapping issue between elements e_1 and e_2 can be solved by fragmenting e_1, assigning the role of breakpoint for e_1 either to <e_2> (if $e_1 \prec e_2$) or to </e_2> (if $e_2 \prec e_1$).

The above principle is implemented in the algorithm by extending the phase making overlapping issue resolution choices (see markFrags in Appendix A) so that tags playing the role of breakpoints with the list of elements they are "breaking" using the "s_brk" (start breakpoint for) and "e_brk" (end breakpoint for) properties. Once such annotations are available, the usual traversal of the TexMECS event stream can output elements as physical partial elements breaking them at breakpoints: each time one is encountered all the element it is breaking are closed (closeFrage at lines 6, 17) and just after reopened (reopenFrags at lines 13, 19). The order in which partial elements are to be closed/opened is forced by the XML well-formedness requirement and is a function of document order.

3.5 Translation via twin documents

We now consider twin documents (see Section 2.5.4), each of which contains a replica of the sacred hierarchy and a single profane hierarchy.

Parsing is peculiar in which it consumes as input a set of document rather then a single one; serialization will be dual to parsing in this respect. The algorithm we propose—Algorithm 5(a)—is similar to the previously proposed SACX algorithm [21]. We do not want to use SACX itself for several reasons: it relies on a definition of restricted GODDAG other than the usual one, which has a sharp distinction between hierarchies (which in turn cannot be represented in formats other than twin documents and stand-off markup); can violate the constraint that no two nodes dominate the same sub-sequence of the frontier; handles leaf nodes as extra nodes in addition to the frontier; it is not capable of representing a sacred hierarchy as shared, but only a shared frontier. All these shortcomings are addressed by Algorithm 5(a).

The basic algorithm intuition is the same of SACX, though we generate as output a stream of TexMECS events which can be in turn fed as input to Algorithm 1(a). Housekeeping is done for maintaining a global parsing position which is always clipped to the minimum parsing position of all document parsers (line 23). Only at that point we know that some parts of the frontier (some cdata, which is per assumption shared by all twin documents) has been overtaken by all parsers; hence we need to output the slice of text between the last global position and the new freshly clipped global position: "lstrip" (mnemonic for left strip) at line 34 does precisely that.

To identify sacred hierarchy tags we assume the existence of the domain-specific demux function, which given as input an element (or some of its representative, e.g.: a parser event or a GODDAG node) classifies it into either the sacred hierarchy (returning 0) or one of the profane hierarchies (returning $n > 0$). Practically such a function can often be implicitly defined, for example starting from a set of external schema documents, one for each hierarchy.

Housekeeping of the sacred hierarchy (lines 15 and 25) is similar to frontier's housekeeping, as we know that sacred tags occur in all twins and we can therefore use them as synchronization

Algorithm 5 Twin documents ↔ GODDAG

(a)

```
TWINS2TEXMECS($D_1, \dots D_n$)
  $E \leftarrow \emptyset$ /* parsers */
  for all $D_i$ do
    $E \leftarrow E \cup \{\text{event\_parser}(D_i)\}$
    $p_i \leftarrow 0$ /* per-doc. position */
  end for
  $p \leftarrow 0$ /* global position */
  $C \leftarrow \emptyset$ ; $s \leftarrow \text{nil}$ /* cdata/sacred buf. */
  $lst\_tag \leftarrow$ false
  while $\exists E_i \in E, E_i.\text{has\_event}()$ do
    for all $E_i \in \{E_j \in E \mid p_j = p\}$ do
      $e \leftarrow E_i.\text{next\_event}()$
      if $e = cdata$ then
        $C \leftarrow C \cup \{e\}$ /* buffer cdata */
      else
        if $\underline{\text{demux}}(e) = 0$ then
          $s \leftarrow e$ /* buffer tag */
        else /* profane hierarchy */
          EMITTAG($e, lst\_tag$)
          $lst\_tag \leftarrow$ true
        end if
      end if
    end for
    $p' \leftarrow p$ ; $p \leftarrow \min(p_i)$
    if $p > p'$ then
      if $s \neq \text{nil}$ then
        if $lst\_tag$ then
          print " "
        end if
        EMITTAG($s, lst\_tag$)
        $lst\_tag \leftarrow$ true
      else
        $cdata \leftarrow$ ""
        for all $c \in C$ do
          $cdata \leftarrow c.\text{lstrip}(p', p)$
        end for
        print $cdata$
        $lst\_tag \leftarrow$ false
      end if
      $s \leftarrow \text{nil}$
    end if
  end while
```

(b)

```
GODDAG2TWINS($G, D$)
  $E \leftarrow \text{event\_parser}(G.\text{EVENTS}())$
  while $e \leftarrow \text{E.next\_event}()$ do
    $n \leftarrow e.\text{node}$
    if $e = start\_tag$ then
      $i \leftarrow \underline{\text{demux}}(n)$
      PRINTTO$^{td}$($E, D, i,$"<$n$.tag>")
    else if $e = cdata$ then
      PRINTTO$^{td}$($E, D, 0, n.\text{content}$)
    else if $e = end\_tag$ then
      $i \leftarrow \underline{\text{demux}}(n)$
      PRINTTO$^{td}$($E, D, i,$"</$n$.tag>")
    end if
  end while

PRINTTO$^{td}$($E, D, i, markup$)

  if $i = 0$ then /* sacred */
    for all $d_j \in D$ do
      print $\gg d_j, markup$
    end for
  else
    $d_1, \dots, d_n \leftarrow D$
    print $\gg d_i, markup$
  end if

EMITTAG($e, need\_sep$)
  if $need\_sep$ then
    print " "
  end if
  assert($e \neq cdata$)
  if $e = start\_tag$ then
    print "<$e$.tag|"
  else if $e = end\_tag$ then
    print "|$e$.tag>"
  end if
```

points. However, since their similarity with the frontier, we need to use as the unit of measure for parsing positions both the frontier length and the number of tags of the sacred hierarchy elapsed since the beginning of the document. Interestingly enough, this seems to close the circle with the proposal made by De Rose in [12] of unifying textual data and range markers in the LMNL data model. This does not come as a surprise, since the document semantic we are grasping here is precisely tag placement.

Finally, two technical remarks: we are assuming, mimicking a restricted GODDAG assumption, that the root node is sacred and thus shared by all twins; the presented algorithm, differently than SACX, grant that no two nodes dominate the same part of the frontier thanks to the properties of Algorithm 1(a) and to the output of empty text events when needed (line 27).

Serialization. Without additional information, serializing a GODDAG to a set of twin documents sharing a sacred hierarchy is not possible, given that the information asserting to which hierarchy a node belongs to is not here. For this reason we rely on the domain-specific demux function also for serialization purposes. Practically, one can imagine that the needed information is already there if the GODDAG has been built out of a set of twin documents or stand-off markup, but can also add via a GODDAG API the information programmatically. Once such information is available, the serialization done by Algorithm 5(b) can traverse the TexMECS event stream and direct markup snippets to the appropriate output document using the PRINTTO^{td}; of course the sacred hierarchy and the frontier are directed to all twins ensuring that they play their synchronization role.

3.6 Translation via stand-off markup

Translations for stand-off markup documents (see Section 2.5.5) are implemented on the observation that twin documents are basically the same representation as stand-off markup with the exception that in twins the sacred hierarchy is replicated rather than factorized out.

Algorithm 6 Stand-off ↔ GODDAG

(a)

```
STANDOFF2TWINS($D_i$)
  $E \leftarrow$ event_parser($D_i$)
  while $e \leftarrow E$.next_event() do
    assert($e \neq cdata$)
    if $e = start\_tag$ then
      if is_so_href($e$) then /* stand-off reference */
        $markup$ ← so_resolve($e$) /* retrieve source part */
        print $markup$
      else
        print "<e.tag>"
      end if
    else if $e = end\_tag$ then
      print "</e.tag>"
    end if
  end while
```

(b)

```
GODDAG2STANDOFF(G)
  Same as Algorithm 5(b), but using PRINTTO$^{so}$ instead of PRINTTO$^{td}$.

PRINTTO$^{so}$($E, D, i, markup$)
  if $i = 0$ then /* sacred hierarchy */
    $e \leftarrow E$.last_event
    if $e = start\_tag$ then
      $id \leftarrow$ fresh_id()
      $e$.set_attr("xml:id", $id$)
      SKIPSUBTREE($D, 0, E, e$)
    else if $e = cdata$ then
      print $\gg D_0, markup$
    end if
    $ref \leftarrow$ ref_to($e$)
    for all $j = 1$ to $D$.length do
      print $\gg D_j, ref$
    end for
  else
    print $\gg D_i, markup$
  end if
```

Parsing is performed via an indirection on twin documents. Each twin should be used as input of Algorithm 6(a); the overall result can then be passed to Algorithm 5(a) to build a GODDAG. The former algorithm simply recognizes stand-off references and expands them to its output (line 5–7); everything else, that is: the profane hierarchy, is printed unchanged to its output.

Serialization can be done reusing Algorithm 5(b) simply plugging into it a different function for delivering markup part to each document; in Algorithm 6(b) we indeed use PRINTTO^{so} (where "so" stands for stand-off) instead of PRINTTO^{td} ("td" for twin documents). Nodes pertaining to the profane hierarchy are printed as they were for twin documents, the frontier and sacred stuff are handled differently. Indeed they are printed to a separate document, shared by all the profane documents, generating reference to the sacred markup as needed; the reference markup will be included in the profane document.[8]

Since with stand-off markup sacred sub-tree cannot recursively contain profane sub-trees, once we begin the traversal of a given sacred sub-tree we will not encounter a profane node before leaving the given sub-tree. Moreover, we know that the encountered markup needs only to be delivered to the sacred document during serialization. Doing so is the role of the SKIPSUBTREE routine whose code is given in Appendix A. The additional generation of a fresh XML identifier for the sacred markup provides for ease of reference for sacred XML elements.

3.7 Translation via LMNL

Finally, we detail the translations involving LMNL (see Section 2.6.2). We have already shown possible translation paths from/to LMNL exploiting the CLIX isomorphism, but here we show direct translations that can also benefit from LMNL layering information.

Algorithm 7 LMNL ↔ GODDAG

(a)

```
LMNL2GODDAG()
  Same as Algorithm 1(a), but using TRED$^l$
  instead of TRED$^t$.

TRED$^l$($G$)
  for all $n_1 \to n_2 \in G$.edges do
    if $n_1$.layer.base $\neq n_2$.layer then
      $G$.rm_edge($n_1 \to n_2$)
    end if
  end for
```

(b)

```
GODDAG2LMNL()
  $E \leftarrow$ event_parser($G$.EVENTS())
  while $e \leftarrow$ E.next_event() do
    $n \leftarrow e$.node
    if $e$ = start_tag then
      if $n$.has_layer() then
        $l \leftarrow n$.layer.name
        print "[$n$.tag~$l$}"
      else
        print "[$n$.tag}"
      end if
    else if $e$ = cdata then
      print "{$n$.tag]"
    else if $e$ = end_tag then
      print $n$.content
    end if
  end while
```

Parsing from LMNL to GODDAG is implemented in Algorithm 7(a) which is a variant of Algorithm 1(a) which processes a stream of SAL[9] events, and uses a different transitive reduction logics. Additionally we also assume that syntactic layering information associated to ranges are propagated to the GODDAG being built.

As we have observed in Section 3.1, just before transitive reduction the set of GODDAG arcs is maximal and they express containment, but not all of them are legitimate dominance arcs according to LMNL layering rule. Indeed, a LMNL range r_1 dominates a range r_2 only if it contains it and its own layer base is r_2's layer. So, what we have to do to express LMNL layer dominance relationship as father/child relationship in the resulting GODDAG is simply to get rid of the arcs violating this rule, that is what TRED^l does.[10]

[8]currently, our serialization algorithm generates references to single sacred nodes. It can be easily extended to generate range references as long as they do not violate continuity

[9]Simple API for LMNL, a SAX-like API for LMNL, `http://lmnl.net/prose/APIs/`

[10]At the time of writing it is not clear whether LMNL dominance will remain defined exploiting explicit layering

Serialization is similar to TexMECS', it is implemented by Algorithm 1(b). The only notable difference is that we should account for absence of layering information in GODDAG nodes (it will be there only if the GODDAG has been built by parsing a LMNL document or added later on programmatically), hence we use the LMNL layering annotation syntax only for nodes where such information is available.

4 Conclusion and future work

This paper sets a long term goal: the creation of an unifying framework to reason formally and programmatically about overlapping markup, a particular kind of markup used to encode data structure richer than those natively encodable in XML languages. The goal is ambitious and need to be pursued stepwise. For this reason the paper also details what we believe to be the first needed step: a set of conversion algorithms between 7 well-known serialization formats for documents with overlapping markup issues.

The algorithm presentations is meant to serve two purposes. On one hand they can be read as blueprints for real-life implementations. Our own implementation—codenamed `PalOLap`—is being developed at the University of Bologna; it is a C/C++ free (as in freedom) conversion library containing all the algorithms discussed in Section 3 and an implementation of restricted GODDAGs. On the other hand, the study of conversion algorithms sketches some lights on the equivalence between serialization formats; while developing them we have discovered several properties of the various formats which have been highlighted in the algorithm discussions, their formal proofs have been omitted for the sake of brevity and will be available in a forthcoming technical report.

Future work is manifold. As this is the first step in the closure of a big technological and standardization gap between overlapping markup and tree-based markup, several pieces are still missing: we hope studies like this one will foster discussions in the researcher and practitioner communities in the direction of that *unification* which has been neglected thus far. On our own we plan to extend the translation algorithms to support the advanced features of complex text documents such as self-overlap, virtual elements, and containment/dominance decoupling. This would imply abandoning restricted GODDAGs as the central model (as they can not represent some of those features), in favour of more expressive data structures such as generalized GODDAGs. This would also mean that some translation algorithms may cause information loss, as not all serialization formats are capable of encoding the aforementioned advanced features (e.g., virtual elements are not supported by CLIX, ECLIX, and twin documents).

Also, we observe that TEI, the most popular XML-based text encoding language, does not mandate the usage of a single technique for dealing with overlapping markup issues, but rather supports a mixture of them. On the contrary, the proposed translation framework assumes that each document is encoded using a single technique. Though practically this is not always an issue—given that large text libraries which have to deal with overlapping issues have always chosen a single technique—we plan to address this issue studying restricted contexts, within a single document, in which the uses of a single technique can be confined.

References

[1] M.H. Abrams and Geoffrey Harpham. *A Glossary of Literary Terms.* Heinle, 9th edition, March 2008.

[2] Wouter Alink, Raoul Bhoedjang, Arjen de Vries, and Peter Boncz. Efficient XQuery support for stand-off annotation. In *International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, 2006.

information or not: some static dominance declarations based on range names are being considered as a replacement. Algorithm 7(a) can be trivially ported to exploit them rather than layering, assuming the dominance declarations are available at GODDAG creation time.

[3] Syd Bauman. TEI HORSEing around. In *Extreme Markup Languages*, 2005.

[4] Steven Bird and Mark Liberman. A formal framework for linguistic annotation. *Speech Communication*, 33(1-2):23–60, 2001.

[5] Lou Burnard and Syd Bauman, editors. *TEI P5: Guidelines for Electronic Text Encoding and Interchange*, chapter 20: Non-hierarchical Structures. TEI Consortium, 2007.

[6] Lou Burnard and Syd Bauman, editors. *TEI P5: Guidelines for Electronic Text Encoding and Interchange.* TEI Consortium, 2007.

[7] Jean Carletta, Stefan Evert, Ulrich Heid, Jonathan Kilgour, Judy Robertson, and Holger Voormann. The NITE XML Toolkit: Flexible annotation for multimodal language data. *Behavior Research Methods, Instruments and Computers*, 35(3):353–363, 2003.

[8] John Cowan, Jeni Tennison, and Wendell Piez. ECLIX: reading XML as LMNL. LMNL wiki.

[9] John Cowan, Jeni Tennison, and Wendell Piez. LMNL update. In *Extreme Markup Languages*, 2006.

[10] Savinien Cyrano de Bergerac and Dominique Moncond'huy. *La mort d'Agrippine.* Table ronde, November 1995.

[11] Alex Dekhtyar and Ionut Emil Iacob. A framework for management of concurrent XML markup. *Data Knowledge Engineering*, 52(2):185–208, 2005.

[12] Steven J. DeRose. Markup overlap: A review and a horse. In *Extreme Markup Languages*, 2004.

[13] Patrick Durusau. Visualizing overlapping hierarchies in textual markup. In *Joint international conference ALLC/ACH*, 2002.

[14] Patrick Durusau and Matthew Brook O'Donnell. Coming down from the trees: Next step in the evolution of markup? In *Extreme Markup Languages*, 2002.

[15] Patrick Durusau and Matthew Brook O'Donnell. Concurrent markup for XML documents. In *XML Europe*, 2002.

[16] International Organization for Standardization (ISO), editor. *ISO 8879: Information processing—Text and office systems—Standard Generalized Markup Language (SGML)*, chapter C.3.1: CONCUR - Document Instances May Occur Concurrently. ISO, 1986.

[17] Alla Goralcikova and Vaclav Koubek. A reduct-and-closure algorithm for graphs. *Mathematical Foundations of Computer Science 1979*, 74:301–307, 1979.

[18] Mirco Hilbert, Oliver Schonefeld, and Andreas Witt. Making CONCUR work. In *Extreme Markup Languages*, 2005.

[19] Claus Huitfeldt and C. M. Sperberg-McQueen. Texmecs: An experimental markup meta-language for complex documents. http://decentius.aksis.uib.no/mlcd/2003/Papers/texmecs.html, 2001.

[20] Claus Huitfeldt and C. M. Sperberg-McQueen. Representation and processing of goddag structures: implementation strategies and progress report. In *Extreme Markup Languages*, 2006.

[21] Ionut Emil Iacob and Alex Dekhtyar. Parsing concurrent XML. In *WIDM: ACM international workshop on Web Information and Data Management*, pages 23–30, New York, NY, USA, 2004. ACM Press.

[22] Ionut Emil Iacob and Alex Dekhtyar. Towards a query language for multihierarchical XML: Revisiting XPath. In *WebDB: international workshop on the Web and DataBases*, pages 49–54, 2005.

[23] H. V. Jagadish, Laks V. S. Lakshmanan, Monica Scannapieco, Divesh Srivastava, and Nuwee Wiwatwattana. Colorful XML: One hierarchy isn't enough. In *SIGMOD Conference*, pages 251–262, 2004.

[24] Theodor Holm Nelson. Embedded markup considered harmful. *World Wide Web Journal*, 2(4):129–134, 1997.

[25] C. M. Sperberg-McQueen. Rabbit/duck grammars: a validation method for overlapping structures. In *Extreme Markup Languages*, 2006.

[26] C. M. Sperberg-McQueen and Claus Huitfeldt. GODDAG: A data structure for overlapping hierarchies. In *DDEP/PODDP*, pages 139–160, 2000.

[27] Jeni Tennison. Creole: Validating overlapping markup. In *XTech 2007: "The ubiquitous web" conference*, 2007.

[28] Jeni Tennison and Wendell Piez. The layered markup and annotation language (LMNL). In *Extreme Markup Languages*, 2002.

[29] Fabio Vitali and David Durand. Using versioning to provide collaboration on the WWW. *World Wide Web Journal*, 1(1):37–50, 1995.

[30] Andreas Witt. Multiple hierarchies: new aspects of an old solution. In *Extreme Markup Languages*, 2004.

[31] Andreas Witt, Oliver Schonefeld, Georg Rehm, Jonathan Khoo, and Kilian Evang. On the lossless transformation of single-file, multi-layer annotations into multi-rooted trees. In *Extreme Markup Languages*, 2007.

Acknowledgements

We would like to thank the anonymous referees for their comment and suggestions which have been truly helpful in improving the presentation of this work.

A Auxiliary routines

```
ADDLEAFTAGS($G, n, L, R$)
  $n$.left ← $n$.left ∪ $L$
  $n$.right ← $n$.right ∪ $R$
  for all $c \in G$.children($n$) do
    $L' \leftarrow \emptyset, R' \leftarrow \emptyset$
    if $c = \min(G.\text{children}(n))$ then
      $L' \leftarrow n.\text{left} \cup \{n\}$
    end if
    if $c = \max(G.\text{children}(n))$ then
      $R' \leftarrow n.\text{right} \cup \{n\}$
    end if
    $G$.ADDLEAFTAGS($c, L', R'$)
  end for
  $G$.has_leaftags ← true
```

```
EVENTS($G$)
  if $\neg G$.has_leaftags then
    $G$.ADDLEAFTAGS($G$.root, $\emptyset, \emptyset$)
  end if
  for all $l \in$ sort($G$.leaves()) do
    for all $n \in$ sort($l$.left) do
      print "<$n$.tag|"
    end for
    print $l$.content
    for all $n \in$ sort($l$.right) do
      print "|$n$.tag>"
    end for
  end for
```

```
MARKMST($G$)
  $O \leftarrow \emptyset$
  $E \leftarrow$ event_parser($G$.EVENTS())
  while $e \leftarrow$ E.next_event() do
    $n \leftarrow e$.node
    if $e = \mathit{start\_tag}$ then
      $O$.push($n$)
    else if $e = \mathit{end\_tag}$ then
      $O$.rm_peer($n$)
      for all $p \in O, p > n$ do
        $m, \bar{m} \leftarrow \underline{\text{choose\_xml}}(n, p)$
        $\bar{m}$.is_mst $\leftarrow$ true
      end for
    end if
  end while
```

```
MARKFRAGS($G$)
  $O \leftarrow \emptyset$
  $E \leftarrow$ event_parser($G$.EVENTS())
  while $e \leftarrow$ E.next_event() do
    $n \leftarrow e$.node
    if $e = \mathit{start\_tag}$ then
      $O$.push($n$)
    else if $e = \mathit{end\_tag}$ then
      $O$.rm_peer($n$)
      for all $p \in O, p > n$ do
        $m, \bar{m} \leftarrow \underline{\text{choose\_xml}}(n, p)$
        $\bar{m}$.is_frag $\leftarrow$ true
        if $m < \bar{m}$ then
          $m$.e_brk $\leftarrow m$.e_brk $\cup \bar{m}$
        else if $\bar{m} < m$ then
          $m$.s_brk $\leftarrow m$.s_brk $\cup \bar{m}$
        end if
      end for
    end if
  end while
```

```
REFERRED($e, O$)
  if $e$.has_attr("xml:id") then
    $id \leftarrow e$.attr("xml:id")
    for all $e' \in O$ do
      if $e'$.has_attr("next") then
        $next \leftarrow e'$.attr("next")
        if $next = id$ then
          return true
        end if
      end if
    end for
  end if
  return false
```

```
SKIPSUBTREE($D, i, E, e$)
  print $\gg D_i$,"<$e$.tag>"
  $depth \leftarrow 1$
  while $depth > 0$ do
    $e' \leftarrow E$.next_event()
    assert($e' = cdata \vee \underline{\text{demux}}(e') = 0$)
    if $e' = \mathit{start\_tag}$ then
      $depth \leftarrow depth + 1$
      print $\gg D_i$,"<$e'$.tag>"
    else if $e' = \mathit{end\_tag}$ then
      $depth \leftarrow depth - 1$
      print $\gg D_i$,"</$e'$.tag>"
    else if $e' = cdata$ then
      print $\gg D_i, e'$.content
    end if
  end while
```