Exploring and Visualizing the "alma" of XML Documents

Daniela da Cruz[1], Pedro Rangel Henriques[1], and Maria João Varanda[2]

[1] Universidade do Minho
Departamento de Informática, Campus de Gualtar
Braga, Portugal
[2] Instituto Politécnico de Bragança
Campus de Santa Apolónia
Bragança, Portugal

Abstract. In this paper we introduce eXVisXML, a visual tool to explore documents annotated with the mark-up language XML, in order to easily perform over them tasks as *knowledge extraction* or *document engineering*.
eXVisXML was designed mainly for two kind of users. Those who want to analyze an annotated document to explore the information contained—for them a visual inspection tool can be of great help, and a slicing functionality can be an effective complement.
The other target group is composed by document engineers who might be interested in assessing the quality of the annotation created. This can be achieved through the measurements of some parameters that will allow to compare the elements and attributes of the DTD/Schema against those effectively used in the document instances.
Both functionalities and the way they were delineated and implemented will be discussed along the paper.

1 Introduction

Our recent research on program comprehension using slicing and visual inspection, as well as the work on grammar metrics led us to investigate how those approaches could be adapted to the field of document engineering. As a consequence we have conceived a tool, denominated eXVisXML, to aid in the inspection and analysis of XML documents.

By analogy with another tool (ALMA) we have developed in the past for program visualization and comprehension, we say that eXVisXML allows us to capture the "*alma*" (in English, the "*soul*") of structured documents, i.e., the intrinsic characteristics of XML documents. eXVisXML allows us to visualize the structure of the document (the hierarchy of XML elements), and provides a set of quality metrics, which enable us to reason out the document properties.

On one hand, our tool shows, in a graphical form, the document tree with the content associated to the leaves, providing means to navigate over it; moreover

it displays, in a tabular form, all the element occurrences associated with the respective attribute/value pairs. Using forward slicing techniques, eXVisXML allows the user to select parts of the document to focus his analysis just on some aspect; namely one can regenerate the original document restricted to some elements. These features are aimed at the comprehension of the document and its exploration (in the sense of knowledge extraction). Inspired in ALMA system, this functionality is displayed in two windows, one for the tree, and the other for the table of elements. We argue that the graphical representation of the abstract syntax tree complemented by the table of elements provides an easy to read and effective way to grasp the sense of the document.

On the other hand, eXVisXML allows the document engineer to assess the quality of his annotation schema (the DTD/XML-Schema he has designed) when applied to real cases. eXVisXML computes automatically a set of syntactic and semantic parameters (according to the standard metrics for XML documents) and shows them in a separate window. Those parameters are evaluated over the actual document and the respective schema in order to be possible, for instance, to compare the total number of elements available against the actual number of different elements used.

Before introducing our tool, eXVisXML, in section 5—describing its architecture and discussing the implementation strategies—we will write about the visualization of XML documents (section 2) and related work, i.e., other tools also developed with a purpose similar to eXVisXML; then we discuss, in section 3, the concept of document slicing and how it can complement the visualization and navigation, making easier the comprehension of the document; at last, we dedicate section 4 to the introduction of metrics to assess XML documents. The paper ends in section 6 with concluding remarks

2 XML Documents Visualization

The ability to retrieve information from plain documents, in a simple and efficient way, is one of the objectives that has motivated the search for markup languages. Concerning machine manipulation, the annotation systems like XML, so far developed, were completely successful; XSL and other production-systems can easily extract information from annotated documents and transform them. However for human beings, this task is not as easy as desirable, mainly if the annotation is complex or the document too big.

To help in finding the document fragments corresponding to some kind of element/attribute, or even located in some sub-document, document engineers developed specific query languages. In the last few years, appeared among many other, XPath [CD99,OMFB02] and XQuery [Cha02] languages specially designed to query collections of XML data. XPath or XQuery stand for XML like SQL for databases, making possible to find and extract elements and attributes from structured documents.

Moreover, the research for tools to visualize XML documents, is not a new issue. People recognized a long time ago that the existence of visual editors was crucial to create or read structured documents.

Nowadays there are many tools which merge the XPath querying facilities with the visualization of XML documents. Some of this tools are:

- XPath Analyzer by Altova [Alt07];
- XPath Visualizer [Top07];
- XPath Viewer by Microsoft [Mic07];
- XPath Query Editor by Stylus Studio [Stu07];

Although these tools offer a (textual) hierarchical view with highlighted syntax, as illustrated in figure 1, and make easier the manipulation of documents, allowing to expand and collapse sets of elements, they are not always powerful enough for the exploration of the document's constituents (elements and attributes) and the relationships among them.

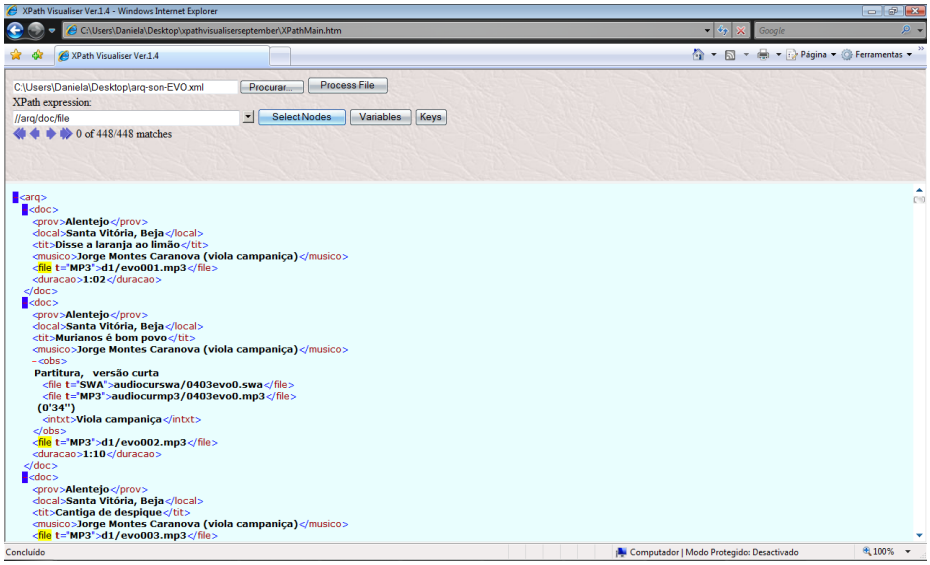

Fig. 1. Visualization of a XML document and selection of `file` nodes

The tool closest to our proposal is XML Schema Designer [Mic08]; however, that tool just deals with XML schemas. XML Designer provides a visual representation of the elements, attributes, types, and so on, that make up XML schemas. With XML Designer we can: construct new or modify existing XML schemas; create and edit relationships between tables; create and edit keys.

Actually, the kind of visualization that we propose is similar to the one provided by XML Schema Designer, but also applicable to XML documents. This is, we

propose a graphical representation of the internal abstract tree associated with the XML document, where intermediate nodes are XML elements and the text fragments (`#PCDATA`) are the leaves.

Edges describe the direct inclusion of document parts. So, we can distinguish two kinds of nodes: *text nodes* and *structure nodes*. The labels of *structure nodes* correspond to XML element types and *text nodes* (always leaves) are labeled with `#PCDATA` components (the actual text of the document). The visual representation used to show this information is lighter than the usual XML tag representation. It is well known the advantage of the use of graphical features to expose and explain structural and behavioral information.

3 XML Documents Slicing

A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest. Such a point of interest is referred to as a *slicing criterion*, and is typically specified by a pair (program point, set of variables). The parts of a program that have a direct or indirect effect on the values computed at a slicing criterion C constitute the *program slice with respect to criterion C*. The task of computing program slices is called *program slicing* [Tip95].

As referred in [Sil05], the slicing technique can also be applied to XML documents. Essentially, given an XML document, it is produced a new XML document (a slice) that contains the relevant information in the original XML document according to some criterion (the *slicing criterion*). Furthermore, it is also possible to slice a DTD, where the output is a new DTD such that the computed slice is valid according to the original DTD.

This technique was implemented in a Haskell prototype tool called XMLSlicer [Sil06], using the HaXML library [Mer01]. In this approach, XML documents and DTD's are seen as trees; and the slicing criterion consist of a set of nodes in the tree. In both types of slicing—DTD slicing and XML slicing—given a set of elements, it will be extracted those elements which are strictly necessary to maintain the tree structure, i.e., all the elements that are in the path from the root to any of the elements in the slicing criterion. The difference between them is that while a slicing criterion in a DTD selects a type of elements, a slicing criterion in an XML document can select only some particular instances of this type.

Both slicing techniques produce valid XML and DTD slices with respect to the slicing criterion, if both the original are valid.

As a conclusion, we can say that this slicing technique can be seen as an easier way to query an XML document, simpler than an XPath/XQuery statement; it does not require to write the complete path to locate some information (or elements) in document.

4 XML Documents Metrics

Effective management of any process requires quantification, measurement, and modeling. Software metrics provide a quantitative basis for the development and validation of models of the software development process. Metrics can be used to improve software productivity and quality.

In the last years, a wide set of software metrics was defined and can be classified as follow: product metrics (to evaluate a software product); process metrics (to evaluate the design process); and resources metrics (to appraise the required resources).

In the field of XML, the quality assessment is also relevant because the approach followed by engineers, or end-users, to design the annotation-schema (the type of a family of documents), or even to markup existing texts, is many times improvised and naif. Concepts like *well-formedness* or *validity* are not sufficient to appraise XML documents; they are only prerequisites to achieve quality.

Some of the software metrics (briefly referred above) have been adopted to measure the quality of XML documents [KSH02], being applied both to DTDs and XML-schemas (XSDs).

A tool dealing with XSD metrics is XsdMetz [Vis06,LKR05]. The tool was implemented in the functional programming language Haskell, using functional graph representations and algorithms. The tool is related with SdfMetz, which computes metrics on SDF grammar representations [AV05]. XsdMetz tool exports successor graphs in dot format so that they could be drawn by GraphViz [KN02].

However, in this paper we will only focus on the metrics defined over DTDs.

As a consequence of that research effort, a set of XML metrics was defined—*size, structure complexity, structure depth, fan-in and fan-out, instability, tree impurity.* Below and after our own contribution (*attributes per element, non-used components and text length*), we introduce them, as they form the basis of the quality measurement that will be implemented by the proposed tool.

Before presenting those metrics, we should define the notion of a *sucessor graph* (SG), now applied to DTDs [Vis06,LKR05], in order to measure the dependence between components. Given a DTD, we say that a new *component* (in this case, an *element* or an *attribute*) is an *immediate successor* of the *element under definition*, i.e., the component in the context of which the new one appears; then, we introduce an arrow (an oriented edge) from the element to the component. Based on this relation, the result is a graph representation of the structure of the XSD/DTD.

Size

$Size(DTD) = n_{EL} + n_A$

where n_{EL} — number of elements in the DTD, and n_A — number of attributes in the DTD.

Given a DTD, its *size* (i.e. the value for this metric) is the total number of nodes in the SG, i.e., the number of DTD components.

Structure complexity

To determine the complexity of a DTD, the McCabe metrics, developed to evaluate the control flow of software, was adopted. There exist slight variations of McCabe Complexity measure (MCC), but in essence MCC counts the number of linearly independent paths through the control flow graph of a program module.

MCC for grammars may simply count all *decisions* in a grammar, this is, operators for *alternative*, *optional* and *iteration*. Because DTDs are equivalent to context-free grammars, *Lammel et al*, in [LKR05], argue that in the same way, the MCC for DTDs correspond to the addition of edges to SG if quantifiers $+$ and $*$ occur and if mixed content elements (but not #PCDATA) exist.

So, the formula to measure the complexity of a DTD is:

$Compl(DTD) = e - n + 1 + n_{IDREF}$,

where e is the number of edges in the SG, n is the number of nodes in the SG and n_{IDREF} is the number of *IDREF* attributes. Note that actually the number of references to other identifiers increases the complexity.

In fact, if the DTD corresponds to a pure tree (which always has n nodes and $n-1$ edges) without internal references, then we get as structural complexity the value $Compl(DTD) = 0$. On the other side, every recursion, all iterators $+$ and $*$, and all $IDREF$ attributes increase the complexity.

Structure Depth

This metric, which computes the depth of the SG, also provides information about the complexity of the schema.

To compute the depth of the SG, we have to eliminate recursion, otherwise the result would be infinite. Then, the depth of each node is computed as follows:

$$Depth(n) = \begin{cases} 0 & n \text{ is leaf} \\ max(Depth(n_i)) + 1 \; for \; each \; n_i \text{ (child node of n)} \end{cases}$$

According to [KSH02], an SG with a depth much higher than seven is complex and reveals a bad DTD design.

Fan-in and Fan-out

These two new metrics are defined as follows:

$Fan-in(n) = \#\{n_i | n_i \text{ is parent node of } n\}$.

$Fan-in$ gives the number of incoming edges in the node.

$Fan-out(n) = \#\{n_i | n_i \text{ is child node of } n\}$.
$Fan-out$ gives the number of outgoing edges in the node.
Both metrics are directly applicable to the nodes of SG. For the graph as a whole, the average and the maximum values for those parameters can be useful to spot unusual nodes, which can be inspected to detect the anomaly and fix the problem. Elements with a high *Fan-in/Fan-out* value are more complex than other elements with a lower value.

Instability

Based on *Fan-in/Fan-out* metrics, a measure related with the *instability* of a node can be computed as follows:

$Instability(SG) = \frac{Fan-out}{Fan-in+Fan-out} \times 100\%$
A node with a low instability allows us to conclude that it is less dependent of other nodes, while many nodes are depend on it. This is, *instability* can be interpreted as resistance to change, hence a node with low instability corresponds to a situation where changes that occur over the node will affect relatively many other nodes.

Tree Impurity

$TI(SG) = \frac{n*(e-n+1)}{(n-1)*(n-2)} * 100\%$
where n is the number of nodes in the SG and e is the number of edges.
This metric is clearly inspired in Fenton's impurity concept used in the context of software or grammar quality assessment.
A tree impurity of 0% means that a graph is a tree and a tree impurity of 100% means that it is a fully connected graph.

Now we introduce the set of complementary new metrics, which we have defined.

Attributes per Element

To complement the *Size metric*, we define
$AttrsEle(DTD) = \frac{\sum n_A}{n_{EL}}$
where n_{EL} — is the number of the elements in the DTD, and n_A — is the number of attributes.
This metric allows us to figure out the average number of attributes defined per element in the DTD.
A similar metric could be defined over the XML document.
$AttrsEle(XML) = \frac{\sum n_{Au}}{n_{ELu}}$

where n_{ELu} — is the number of the elements used in the document, and n_{Au} — is the number of attributes actually used.

This metric, applied directly to the XML document, allows us to figure out the average number of attributes actually used per effective elements present in the XML document.

Non-used Components

In order to detect the non-used components (elements and attributes) in an actual XML document, we define:

$NonAttr(XML) = Attr(DTD) - Attr(XML)$

$NonElem(XML) = Elem(DTD) - Elem(XML)$

if $Attr(DTD)$ represents the set of attributes defined in the DTD, and $Attr(XML)$ represents the set of actual attributes (the attributes used in the XML document instance), then $NonAttr(XML)$ is the set of non-used attributes.

The set of non-used elements, $NonElem(XML)$, is defined precisely in the same way; once again, it gives an idea of the elements in the DTD that are not used in XML instances (it is similar to the notion of *dead-code* in a class—this is, *a set of methods that are never called*).

Then we define two metrics:

$NAttr(XML) = \#NonAttr(XML)$

$NElem(XML) = \#NonElem(XML)$

that measure the size (number of elements) of those two sets.

Text Length

$$TxtLen(XML) = \frac{\sum length(PCDATA)}{n_{PCDATA}}$$

where, length(PCDATA) computes the total length of the document's text (the sum of the length of all text fragments, i.e., text associated with element tags, or untagged text), and n_{PCDATA} is the number of text fragments (the number of $PCDATA$ leaves that appear in the XML document tree).

In a similar way,

$$AttTxtLen(XML) = \frac{\sum length(AttPCDATA)}{n_{Au}}$$

measures the average attribute text length.

Usually, the choice between the use of an *element* or an *attribute*, in a XML document type, is an ambiguous matter; in practice, some document engineers consider some particularities as elements, while others consider them as attributes. That metrics is precisely useful to study that phenomena; in fact, when we write a XML instance that duality/ambiguity becomes clear. We have the perception that an attribute should be used when its content is not too large, while an element should be used when we do not know how much large will be its content.

Over XML-schemas, the metrics applied are similar to the referred above, but with a slight difference: usually, the successor graph is built in the same way but the set of nodes that are strongly connected are grouped into the same node (a *module*). However, as said previously, we will not consider them in this paper; to learn more about the common metrics defined over XML-schemas, we suggest the reading of [Vis06].

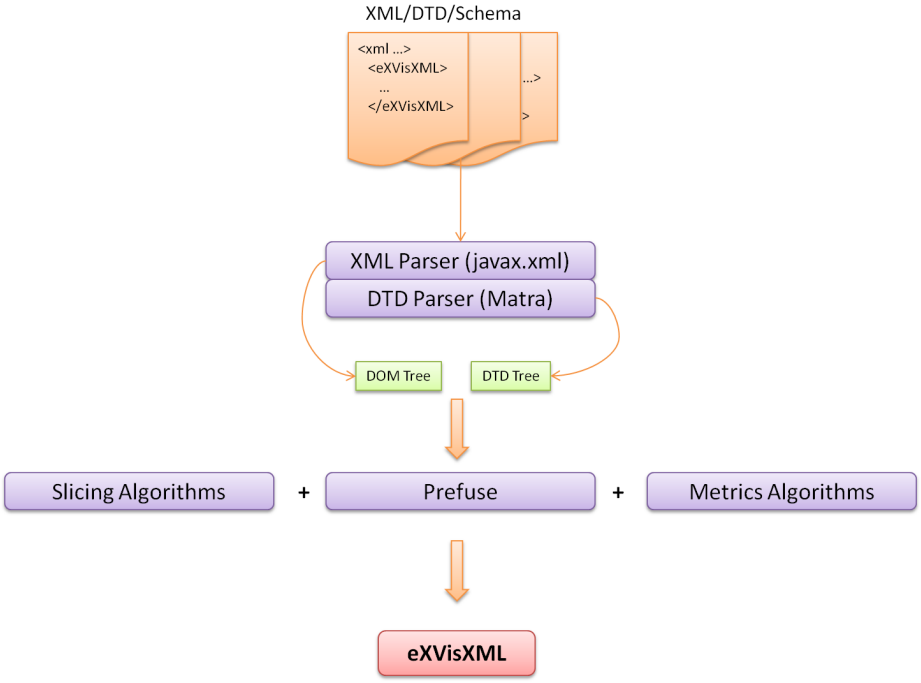

Fig. 2. Architecture of eXVisXML

5 eXVisXML, XML Document Visualization and Exploration

In this section, we concretize the ideas introduced along the previous sections, concerned with visualization, slicing and measuring of XML and DTD documents, discussing how they will be fully implemented in the proposed tool eXVisXML.

Nowadays, the development of a tool requires that the implementor searches for existing programming resources (libraries, design-patterns or program-templates, frameworks, generators, etc.), which can be used in his specific project.

So, in order to get advantage from other tools to build up our own, Java language and the Eclipse platform will be used as the programming environment.

The input for our tool are the the DTD and the XML document. From these 2 documents we can extract all the information needed. The information extraction process will be done by parsing the documents.

In this context, we will use the Java API for XML Processing (JAXP), which can be applied to parse and transform XML documents independent of a particular XML processing implementation. This API provides 2 basic interfaces: the DOM interface and the SAX interface.

The main difference between them arises from the fact that DOM interface builds a complete in-memory representation of the XML document, while SAX interface does not create an in-memory representation; instead, SAX parser informs clients of the XML document structure by invoking callbacks, and so it is faster and uses less memory.

For the purpose of our tool, we chose DOM because we need all the information stored in memory for visualization, slicing and measurement. DOM parser is called through the DocumentBuilder class, which creates an org.w3c.dom.Document instance, an abstract tree representing the structure of the XML Document.

Concerning the parsing of DTDs, a search over the Web led us to a tool called Matra, a Java-based XML/DTD parser utility, that parses the DTD and builds up a tree representation. Other tools of this kind were found and studied; although Matra proved to be the best, concerning its final representation.

Figure 2 depicts the architecture of eXVisXML, summarizing the technical decisions described above.

We discuss in the following subsections how to visualize, slice and measure the input documents. Those features will be illustrated by means of an working example (see appendix A for the XML document and its DTD)—*an excerpt of the well-known screenplay by William Shakespeare, The Romeo and Juliet Love Story [New01], (RJls)*—previewing the output that it will produce. Moreover, this will give a flavor of eXVisXML behavior.

5.1 Visualization

The role of the visualization technology, in fields like program comprehension and software engineering, is strongly recognized by the computer science community as a very fruitful one. The use of software visualization features allows us to get a high quantity of information in a faster way. Graphical representations have a positive impact in learning process because it engages the users in a more efficient comprehension process.

There are several kinds of views that can be produced: they can show *operational data* or *behavioral data* (more abstract view); they can be *static* or *dynamic*; they can be more *structural* or they can be more *quantitative* (based on metrics or other kind of statistical information).

These graphical or iconic representations must be carefully chosen because they usually depend on the problem domain. In our case, we want to visualize XML declarations or documents. Since structure/content visualization is used as a vehicle to make easier the comprehension of a document, it is necessary to care about the choice of visual paradigms/styles that will be used.

Taking this fact into account and inspired in ALMA, our visualization tool for program animation, the eXVisXML interface for the visual inspection of XML documents will be divided into 3 main parts:

- one window that displays the source document;
- one window exhibiting the tree associated with the source document — both tree representations, the graphical one (see Fig. 3 and Fig. 4) and the hierarchical textual view (like the one in Fig. 1), will be available;
- one window to show the Attribute Table (AT), formally a map: $Name \times Value$ — for each *element* selected over the tree, the AT shows the set of attributes of that element and the actual value of each one.

5.2 Slicing

According to a *slicing criterion* given by the end-user, the tool shall be able to select and highlight the path from the root until the node satisfying the criterion. If the *slicing criterion* matches an attribute, not only the node where the attribute appears will be highlighted, but also the corresponding line in the Attribute Table.

Considering again the working example, RJIs document globally shown in Fig. 3, suppose that "*Greg*" was chosen as the *slicing criterion* value in order to find all the screenplay components where actor *Gregory* appears.

The slicing algorithm, included in our tool, will traverse the tree looking for all matches of "*Greg*" with the value of each attribute and each leaf (#PCDATA value). The result of this *slicing operation* will be the enhancement of each path from the root of the document tree until each node where a match happened, as can be seen in Fig. 5.

As an additional feature, eXVisXML can generate a new XML document including only the components along the pathes highlighted in the previous *slicing operation*; the result is shown in Fig. 6. Notice that this new XML is also valid according to the submitted DTD, hence the structure of the XML document is not changed.

5.3 Metrics

Applying to the working example (the RJIs screenplay in appendix A), the set of metrics defined in section 4, we obtain the measures listed below. To evaluate part of those parameter values it was necessary to build first the *sucessor graph* for the given DTD. Fig. 7 sketches that SG.

- $Size(DTD) = 25 + 2 = 27$
- $Compl(DTD) = e - n + 1 + n_{IDREF} = 37 - 25 + 1 + 0 = 13$
-

$$Depth(n) = \begin{cases} 0 & n \text{ is leaf} \\ max(Depth(n_i)) + 1 & for\ each\ n_i \text{ each child node of n} \end{cases}$$

Taking n as the root, the Depth of the SG is 7, since the deepest branch has depth 6.

For the Fan-in and Fan-out metrics, let us consider the node *scene*.

- $Fan - in(n) = \#\{n_i | n_i \text{ is parent node of } n\} = 3$.
- $Fan - out(n) = \#\{n_i | n_i \text{ is child node of } n\} = 6$.

Considering the node *title*, the value of Fan-in and Fan-out metrics are 6 and 0, respectively.

Using the metrics above, the result of Instability metric is computed:

- $Instability(scene) = \frac{Fan-out}{Fan-in+Fan-out} \times 100\% = 3,3\%$
  $Instability(title) = 0\%$
- $TI(SG) = \frac{n*(e-n+1)}{(n-1)*(n-2)} * 100\% = \frac{24*(33-24+1)}{23*21} * 100\% = 58,9\%$
- $AttrsEle(DTD) = \frac{\sum n_A}{n_{EL}} = \frac{2}{25} = 0,08$
- $AttrsEle(XML) = \frac{\sum n_{Au}}{n_{ELu}} = \frac{2}{74} = 0,027$
- $NonAttr(XML) = Attr(DTD) - Attr(XML) = 0$
- $NonElem(XML) = Elem(DTD) - Elem(XML) = 25 - 24 = 1$
  The element stagedir under the context of the element line is never used.
- $TxtLen(XML) = \frac{\sum length(PCDATA)}{n_{PCDATA}} = \frac{2135}{57} = 37,46$
- $AttTxtLen(XML) = \frac{\sum length(AttPCDATA)}{n_{Au}} = \frac{2}{2} = 1$

When the user selects the appropriate option from eXVisXML menu, our tool will compute automatically the metrics above, and open a new window to display the values obtained.

6 Conclusion

Along the paper, we defend the idea that an useful tool to explore XML documents can be setup merging principles from similar areas (like software and grammar engineering, comprehension and quality assessment), as well as resorting to technological solutions already implemented.

As a proof of concept, we conceived and partially built eXVisXML, as proposed in section 5.

Basically we reuse visualization principles (section 2), slicing techniques (section 3), and software/grammar metrics (section 4), aiming at an exploration environment that allows us to comprehend by visual inspection the structure and contents of XML documents, and provides quantitative information to reason about the quality of the mark-up schema as well as the annotation itself.

The complete implementation of eXVisXML is the task we are working on, at moment. This is crucial to test the tool and prove our ideas, as well as to carry out performance, and usability measurements. After that we will apply the tool to a vast suite of test-cases in order check the set of metrics here proposed; maybe some of them are useless, and some others are missing. Of course, that test suite will be useful to tune the visualization, as well as to verify the effective importance of the slicing functionality for document understanding and re-engineering.

References

[Alt07] Altova. Xmlspy. http://www.altova.com/products/xmlspy, 2007.

[AV05] Tiago Alves and Joost Visser. Metrication of sdf grammars. Research report, Departamento de Informática, Universidade do Minho, Maio 2005.

[CD99] James Clark and Steve DeRose. Xml path language (xpath) version 1.0. Technical report, World Wide Web Consortium, 1999.

[Cha02] D. Chamberlin. Xquery: An xml query language. *IBM Syst. J.*, 41(4):597–615, 2002.

[KN02] Eleftherios Koutsofios and Stephen North. *Drawing graphs with* dot, 2002.

[KSH02] Meike Klettke, Lars Schneider, and Andreas Heuer. Metrics for xml document collections. In *EDBT '02: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, pages 15–28, London, UK, 2002. Springer-Verlag.

[LKR05] R. Lämmel, Stan Kitsis, and D. Remy. Analysis of XML schema usage. In *Conference Proceedings XML 2005*, Novembro 2005.

[Mer01] David Mertz. Transcending the limits of DOM, sax, and xslt: The haxml functional programming model for xml. *IBM developerWorks (XML Matters column)*, October 2001.

[Mic07] Microsoft. Xpath viewer. http://msdn2.microsoft.com/en-us/library/aa302300.aspx, 2007.

[Mic08] Microsoft. Xml schema designer. http://msdn2.microsoft.com/en-us/library/ms171943(VS.80).aspx, 2008.

[New01] Greg Newby. Xml and project gutenberg. http://www.ils.unc.edu/ bluec/gutenbergDTD/, 2001.

[OMFB02] D. Olteanu, H. Meuss, T. Furche, and F. Bry. Xpath: Looking forward, 2002.

[Sil05] Josep Silva. Slicing xml documents. In *WWV*, pages 121–125, 2005.

[Sil06] Josep Silva. Xmlslicer. http://www.dsic.upv.es/ jsilva/xml/, 2006.

[Stu07] Stylus Studio. Xpath query editor. http://www.stylusstudio.com/xpath_evaluator.html#, 2007.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[Top07] Xpath visualizer. http://www.topxml.com/xpathvisualizer/, 2007.

[Vis06] Joost Visser. Structure metrics for xml schema. In *XATA - XML: Aplicações e Tecnologias Associadas, Portalegre - Portugal*, Fev 2006.

A The Romeo and Juliet love story

In this appendix we list the two documents—a DTD and an XML text—used as the eXVisXML input for the working example run along the subsections of section 5.

A.1 The Screenplay DTD

The DTD specified to define an XML dialect to mark-up Shakespear screenplays.

```
<!ELEMENT guttext (markupmeta, play, endgutmeta)>
<!ELEMENT markupmeta (title, gutdate, textnum, para, gutfilename)>
<!ELEMENT play (frontmatter, playbody)>
<!ELEMENT endgutmeta (#PCDATA)>
<!ELEMENT stagedir (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT gutdate (#PCDATA)>
<!ELEMENT textnum (#PCDATA)>
<!ELEMENT para (#PCDATA)>
<!ELEMENT gutfilename (#PCDATA)>
<!ELEMENT pgroup (#PCDATA | title | persona)*>
<!ELEMENT persona (#PCDATA)>
<!ELEMENT frontmatter (titlepage, personae)>
<!ELEMENT titlepage (pubinfo, title, author)>
<!ELEMENT personae (title, pgroup+)>
<!ELEMENT playbody (scene, act+)>
<!ELEMENT scene (scndesc | title | stagedir | speech | note)+>
<!ATTLIST scene
    id NMTOKEN #IMPLIED
>
<!ELEMENT act (title?, scene+)>
<!ATTLIST act
    id NMTOKEN #REQUIRED
>
<!ELEMENT scndesc (#PCDATA)>
<!ELEMENT speech (speaker | line | stagedir)*>
<!ELEMENT note (#PCDATA)>
<!ELEMENT speaker (#PCDATA)>
<!ELEMENT line (#PCDATA | stagedir)*>
<!ELEMENT pubinfo (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

A.2 The XML document

An excerpt from RJIs screenplay by William Shakespear, annotated according to the mark-up language defined by the DTD in previous subsection.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<guttext>
    <markupmeta>
        <title>The Complete Works of William Shakespeare
The Tragedy of Romeo and Juliet</title>
        <gutdate>November, 1997</gutdate>
        <textnum>1112</textnum>
        <para>The Library of the Future Complete Works of William Shakespeare
Library of the Future is a TradeMark (TM) of World Library Inc.</para>
        <gutfilename>
******This file should be named 1ws1610.txt or 1ws1610.zip*****
Corrected EDITIONS of our etexts get a new NUMBER, 1ws1611.txt
VERSIONS based on separate sources get new NUMBER, 2ws1610.txt
</gutfilename>
    </markupmeta>
    <play>
        <frontmatter>
            <titlepage>
                <pubinfo>1595</pubinfo>
                <title>THE TRAGEDY OF ROMEO AND JULIET</title>
                <author>by William Shakespeare</author>
            </titlepage>
            <personae>
                <title>Dramatis Personae</title>
                <pgroup>
                    <title>Chorus</title>
                    <persona>Escalus, Prince of Verona.</persona>
                    <persona>Paris, a young Count, kinsman to the Prince.</persona>
                    <persona>Capulet, heads of two houses at variance with each other.</persona>
                    <persona>An old Man, of the Capulet family.</persona>
                    <persona>Romeo, son to Montague.</persona>
                    <persona>Abram, servant to Montague.</persona>
                    <persona>Sampson, servant to Capulet.</persona>
                    <persona>Gregory, servant to Capulet.</persona>
                    <persona>Lady Montague, wife to Montague.</persona>
                    <persona>Lady Capulet, wife to Capulet.</persona>
                    <persona>Juliet, daughter to Capulet.</persona>
                </pgroup>
                <pgroup>Citizens of Verona; Gentlemen and Gentlewomen of both houses;
     Maskers, Torchbearers, Pages, Guards, Watchmen, Servants, and
     Attendants.
   </pgroup>
            </personae>
        </frontmatter>
        <playbody>
            <scene>
                <scndesc>SCENE.--Verona; Mantua.</scndesc>
                <title>THE PROLOGUE</title>
                <stagedir>Enter Chorus.</stagedir>
                <speech>
                    <speaker>Chor.</speaker>
```

```
                        <line>Two households, both alike in dignity,</line>
                        <line>In fair Verona, where we lay our scene,</line>
                        <line>From ancient grudge break to new mutiny,</line>
                        <line>Where civil blood makes civil hands unclean.</line>
                        <line>From forth the fatal loins of these two foes</line>
                        <line>A pair of star-cross'd lovers take their life;</line>
                        <line>Whose misadventur'd piteous overthrows</line>
                        <line>Doth with their death bury their parents' strife.</line>
                        <line>The fearful passage of their death-mark'd love,</line>
                        <line>And the continuance of their parents' rage,</line>
                        <line>Which, but their children's end, naught could remove,</line>
                        <line>Is now the two hours' traffic of our stage;</line>
                        <line>The which if you with patient ears attend,</line>
                        <line>What here shall miss, our toil shall strive to mend.</line>
                        <stagedir>[Exit.]</stagedir>
                    </speech>
                </scene>
                <act id="1">
                    <scene id="1">
                        <scndesc>Scene I.  Verona. A public place.</scndesc>
                        <stagedir>Enter Sampson and Gregory (with swords and bucklers) of the house
of Capulet.</stagedir>
                        <speech>
                            <speaker>Samp.</speaker>
                            <line> Gregory, on my word, we'll not carry coals.</line>
                        </speech>
                        <speech>
                            <speaker>Greg.</speaker>
                            <line> No, for then we should be colliers.</line>
                        </speech>
                        <speech>
                            <speaker>Samp.</speaker>
                            <line> I mean, an we be in choler, we'll draw.</line>
                        </speech>
                        <speech>
                            <speaker>Greg.</speaker>
                            <line> Ay, while you live, draw your neck out of collar.</line>
                        </speech>
                        <speech>
                            <speaker>Samp.</speaker>
                            <line> I strike quickly, being moved.</line>
                        </speech>
                        <speech>
                            <speaker>Greg.</speaker>
                            <line> But thou art not quickly moved to strike.</line>
                        </speech>
                        <note>THE END</note>
                    </scene>
                </act>
            </playbody>
```

```
    </play>
    <endgutmeta>
End of this Etext of The Complete Works of William Shakespeare
The Tragedy of Romeo and Juliet
</endgutmeta>
</guttext>
```

Fig. 3. Tree representation for RJIs Doc. — global view

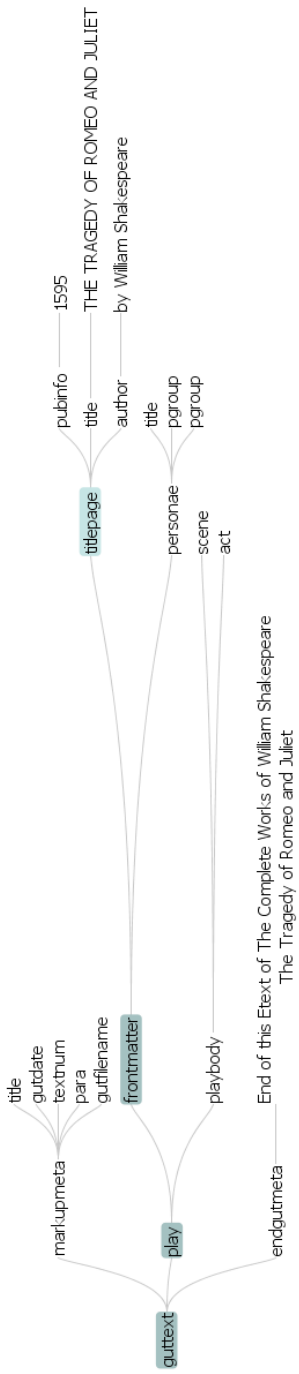

Fig. 4. Tree representation for RJIs Doc. — partial view (some nodes collapsed)

Fig. 5. Result of the *slicing criterion* “Greg”

Fig. 6. New XML generated according to the *slicing criterion*

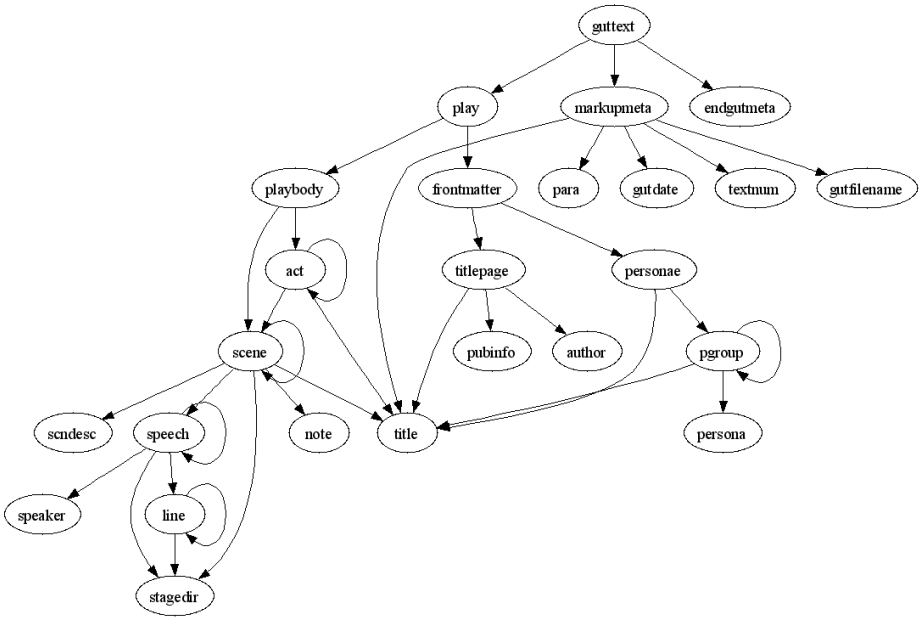

Fig. 7. Sucessor Graph for RJIs DTD