Journal of Universal Computer Science, vol. 17, no. 1 (2011), 126-150
submitted: 8/4/10, accepted: 12/12/10, appeared: 1/1/11 © J.UCS

Visualizing and Analyzing the Quality of XML Documents

Daniela da Cruz
(Universidade do Minho, Braga, Portugal
danieladacruz@di.uminho.pt)

Pedro Rangel Henriques
(Universidade do Minho, Braga, Portugal
prh@di.uminho.pt)

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.

Key Words: Document engineering, Quality assessment, Visualization, Slicing
Category: D.2.8, H.0

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, called eXVisXML, to aid in the inspection and analysis of XML documents.

By analogy with another tool we have developed in the past for program visualization and comprehension, we say that eXVisXML allows us to capture 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 we 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). This feature 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 address 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 discuss metrics to assess XML documents. The paper ends in Section 6 with some 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 (Clark & DeRose, 1999; Olteanu *et al.*, 2002) and XQuery (Chamberlin, 2002) 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[1]; XPath Visualizer[2]; XPath Viewer by Microsoft[3]; XPath Query Editor by Stylus Studio[4].

Although these tools offer a (textual) hierarchical view with highlighted syntax 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.

The tool closest to our proposal is XML Schema Designer[5]; 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*

[1] http://www.altova.com/products/xmlspy
[2] http://www.topxml.com/xpathvisualizer/
[3] http://msdn.microsoft.com/en-us/library/aa302300.aspx
[4] http://www.stylusstudio.com/xpath_evaluator.html
[5] http://msdn2.microsoft.com/en-us/library/ms171943(VS.80).aspx

respect to criterion C. The task of computing program slices is called *program slicing* (Tip, 1995).

As referred in (Silva, 2005), 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 XML-Slicer (Silva, 2006), using the HaXML library (Mertz, 2001). 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 (Klettke *et al.*, 2002), being applied both to DTDs and XML-schemas (XSDs).

A tool dealing with XSD metrics is XsdMetz (Visser, 2006; Lämmel *et al.*, 2005). 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 (Alves & Visser, 2005). XsdMetz tool exports successor graphs in dot format so that they could be drawn by GraphViz (Koutsofios & North, 2002).

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 (Visser, 2006; Lämmel *et al.*, 2005), 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

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.

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

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

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 (Lämmel *et al.*, 2005), 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:

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

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:

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

where n_i corresponds to a child of node n.

According to (Klettke *et al.*, 2002), an SG with a depth much higher than seven is complex and reveals a bad DTD design.

Fan-in and Fan-out

$FanIn$ gives the number of incoming edges in the node.

$$\boxed{FanIn(n) = \#\{n_i | n_i \text{ is parent node of } n\}}$$

$FanOut$ gives the number of outgoing edges in the node.

$$\boxed{FanOut(n) = \#\{n_i | n_i \text{ is child node of } n\}}$$

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 *FanIn*/*FanOut* value are more complex than other elements with a lower value.

Instability

Based on *FanIn*/*FanOut* metrics, a measure related with the *instability* of a node can be computed as follows:

$$\boxed{Instability(SG) = \frac{FanOut}{FanIn+FanOut} \times 100\%}$$

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

$$\boxed{TI(SG) = \frac{n*(e-n+1)}{(n-1)*(n-2)} * 100\%}$$

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

$$\boxed{AttrsEle(DTD) = \frac{\sum n_A}{n_{EL}}}$$

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.

$$\boxed{AttrsEle(XML) = \frac{\sum n_{Au}}{n_{ELu}}}$$

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 XML document, we define:

$$\boxed{NonAttr(XML) = Attr(DTD) - Attr(XML)}$$

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:

$$\boxed{NonElem(XML) = Elem(DTD) - Elem(XML)}$$

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:

$$\boxed{NAttr(XML) = \#NonAttr(XML)}$$

and,

$$\boxed{NElem(XML) = \#NonElem(XML)}$$

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

Text Length

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

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,

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

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 (Visser, 2006).

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 were 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.

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.

The diagram in Figure 1 depicts the architecture of eXVisXML the idea it gives is that of the flow of information through the various components, on a input/output basis, with the input being the documents and output the data produce by the metrics or slicing algorithms.

The application relies heavily on a clean user interface, as a requisite it has to be capable of synthesizing great quantities of information in small area. Special care had to be taken on the design and usability. For the effect we have designed an intuitive application flow, from the application description to the actual view of the document and metrics as depicted in the following diagram of figure 2.

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 Figures 6 and 7) — an excerpt of the well-known screenplay by

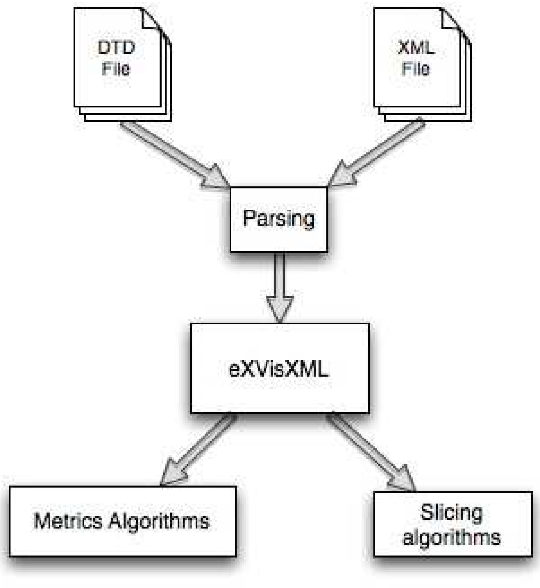

Figure 1: eXVisXML architecture

William Shakespeare, The Romeo and Juliet Love Story[6], (RJls) — previewing the output that it will produce. Moreover, this will give a flavor of eXVisXML behavior.

5.1 XML/DTD Parsing

Parsing technologies differentiate themselves, mainly on the approach taken to undergo the access and representation of a document, which ends up having an increased importance in the parser's choice.

- Simple API for XML (SAX) - One of the fastest and most efficient mechanisms, much due its event oriented sequential reading.
- Document Object Model (DOM) - Performs a document traversal building a in-memory tree representation, and for that reason this is a slow and memory demanding mechanism.
- Streaming API for XML Parsing (StAX) - A compromise between the last two types of parser technologies, it does not require a document traversal or

[6] http://www.ils.unc.edu/˜bluec/gutenbergDTD/

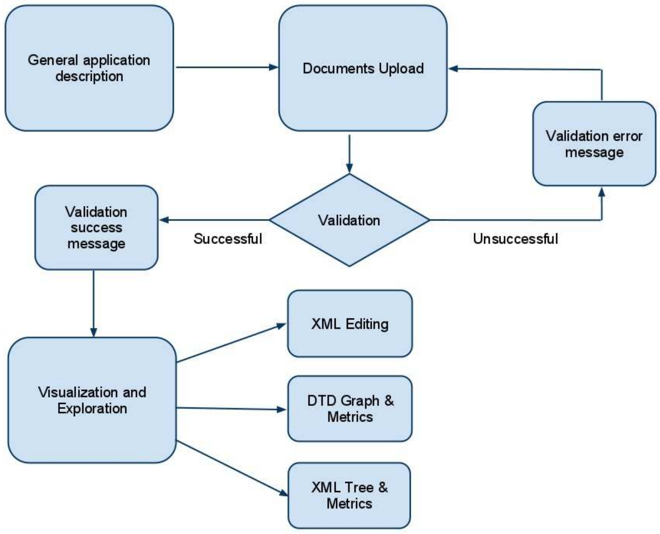

Figure 2: Flowchart of eXVisXML

sequential reading, it can position itself wherever on the document, reading information as needed.

There are many implementations of this technologies, across a number of programming languages, for that reason it becomes essential to find not only the most adequate implementation, performance wise that is, but also the most sustainable one, meaning, that help us focus on the problem itself and not on technical difficulties, so that the application reaches its full potential.

Parser	**Technology**	**Language**
libxml2	SAX	C
WoodStox	StAX	Java
NSXMLParser	SAX	Objective-C
lxml	SAX	Python

Table 1: XML parsers in this work, according to parser type and language implementation.

XML Parser

The parameters established for the selection of the parser where based on the technology in which the parser was design (SAX, DOM or StAX), the language it was conceived, as well as performance and general documentation quality. There is, however, one more relevant factor in the election of the parser, it is necessary that the programming language that the parser will be developed on allows for a simple yet powerful graphical support.

In order to determine the most high-performing parser a test was designed. The test consists in the parsing of a three distinct documents of different size, the parser should perform three simple operations: add the number of elements and attributes in the document, and count the text size in #PCDATA text elements.

DTD Parser

There are no specific parameters for a DTD parser selection. DTD have fairly simple structures offering no adversities in parsing, and so no special consideration is taken in the selection of a particular DTD parser, functional or performance wise. There is only a programming language dependency, between the XML parser chosen and the DTD parser. It is then part of our list of considerations, which continuos to be leaded by XML parser performance.

Decision

Our investigation lead us to narrow down the number of options increasingly. It became immediately clear that the most used parser technologies nowadays present some limitations on parsing, namely as the document size increases, therefore, DOM type parsers were immediately excluded. We suppose that eX-VisXML will be used mostly to analyze large documents, it is then unwise to store a very large document under a data structure in memory, such as DOM does.

And so the attention turned to SAX based parsers, altogether more efficient. However, we still required a intermediate representation for some functionalities, the creation of such a representation would be time consuming and not relevant for the objective of the project at hand. Moreover SAX still has to pass through an entire document for each event callback, although much more efficient than DOM, it bares an unnecessary weight. The solution figured as a compromised between this two technologies, which fortunately it exists. StAX offered us the compromise we catered between DOM and SAX but, unfortunately, it does not provide any type of efficient in-memory object style structure.

StAX first appeared in the Java community, a number of Java parsers took immediate advantage. It was only natural that our search for a StAX based parser started in the platform where it is most mature.

JAXB provided us with all the capabilities needed, and more. It allows for the validation and parsing of XML documents. But mostly it distinguishes itself with two features, object mapping and the ability to drop down to a SAX or DOM based parser. Object mapping is an interesting feature, it works by binding a xml document to a java class, a process called marshaling or inversely, unmarshaling. This feature is most helpful in a development point of view, sadly JAXB requires that a class model exist, making this capability unuseful in this project context.

This lead us to search for yet another compromise. We came to Woodstox, a very fast StAX based parser, with support for XML validation. Woodstox in conjunction with Apache Axiom[7] provides a efficient parser with object mapping capabilities. Furthermore, Apache Axiom sits on top of Woodstox reading the parser events, transforming and adding them to a object tree and storing it in memory on a per read basis, bringing together all the advantages of DOM with the speed of StAX.

5.2 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 as discussed in Section 2. 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, eXVisXML interface is 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 Figure 3) and the hierarchical textual view, will be available;

[7] http://ws.apache.org/axiom/

– 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.

The selection of the graphical environment was done according to the following criteria: ease of development; documentation; platform independent.

It became apparent as we examined and searched for viable solutions that the current state of desktop based platform independent UI frameworks was unsatisfactory with the most advanced and capable one being Java Swing, furthermore, there is a high tendency towards web based data visualization toolkits, an important part of the application being discussed. Given the current advances in web development it was possible to develop a functional version of eXVisXML completely on web, a convenient solution, although we still desired the speed provided by a desktop based implementation.

The idea of making eXVisXML accessible from a web interface became more real as the desire for taking it to a wider crowd grew. Unfortunately it has its shortcomings: although broadband connections are common nowadays, there is still the technical impossibility of uploading a very large document, both client side (upload rate) and server side (storage capacity). We expect eXVisXML to cater people using large documents, however few documents should be large enough for uploading to become impractical, furthermore eXVisXML is a tool for analyzing the structure not content of XML documents, we can safely predict that most documents are comprised of a recurrent structure. Therefore the application should serve the purpose it stands for without neglecting its primary user base.

It is fair to say that any application dealing with large XML documents is limited by hardware, and so we do not believe that this is an impediment. However we do realize that eXVisXML with it's emphases in visual evaluation is more appealing to users with dealing with complex documents.

The desire of building a desktop version of eXVisXML is still well alive, and so the option of building eXVisXML as a modular application, independent from interface became clear, as it provides more alternatives in a long-term future.

Decision

The application discussed in this document is a proof of concept, as such, it's acceptance is attained by the average concordance with the approaches taken, it also means that as a piece of software underdevelopment.

By making it available through the internet and by making it easy for any user to tinker with it, without the need to download any additional software we expect to attract a more significant user base.

A wider and varied user base gives us the opportunity to study how users interact with the platform. With this precious feedback the tool should improve and mature greatly, hopefully proving our initial assumptions right.

The central piece of the software is no doubt the visualization platform, used to render an image of the documents. Through the course of this project we have looked into a handful of visualizations toolkit, both from a desktop point of view as well as a web point of view. From these handful a few have distinguished, though only one was chosen in the end. These where:

Prefuse was among the firsts to investigate mainly because it was also used in previous iterations of eXVisXML Prefuse latest release dates of 2007, meaning the project currently is not actively maintained, which was a major downside. Nevertheless it is still among the best visualizations tools for the desktop if we had chosen to follow that specific path.

Prefuse provides visualization for the Java programming language, making it easy to integrate with the core eXVisXML application if the opportunity arises.

Prefuse flare[8] is a more recent version of the Prefuse toolkit, web oriented and developed in ActionScript for being played with the adobe flash player. Prefuse flare was used in the beginning of development quickly became apparent that it was incapable of rendering larger documents smoothly. Initial tests proved that a smaller document ranging from 100KB to 300KB was sufficient to bring down most browsers, raising CPU utilization to 100% and making normal computer utilization impossible.

The connection between this visualization toolkit and the core application was facilitated with the use of a JGraphT a graph library for java which was capable of exporting in GraphML format, however the control over the document exportable format was almost inexistent, it would often grow immensely and still not provide all the information we would find useful.

ZGRViewer[9] showed up as an alternative to Prefuse flare's CPU utilization problem. ZGRViewer reads graphviz dot documents and displays it in the browser by the means of a Java applet. Once again export was facilitated though limited through JGraphT.

ZGRViwer would render significantly faster then Prefuse flare, although the quality and usability of the graphs wasn't nearly as good.

In the XML tree view ZGRViewer would zoom out to a whole view of the tree, even small documents can take up more than one screen space, the visualization would initially appear as a squeezed web and magnification was also slow, it soon became clear that ZGRViewer would not suit our needs also.

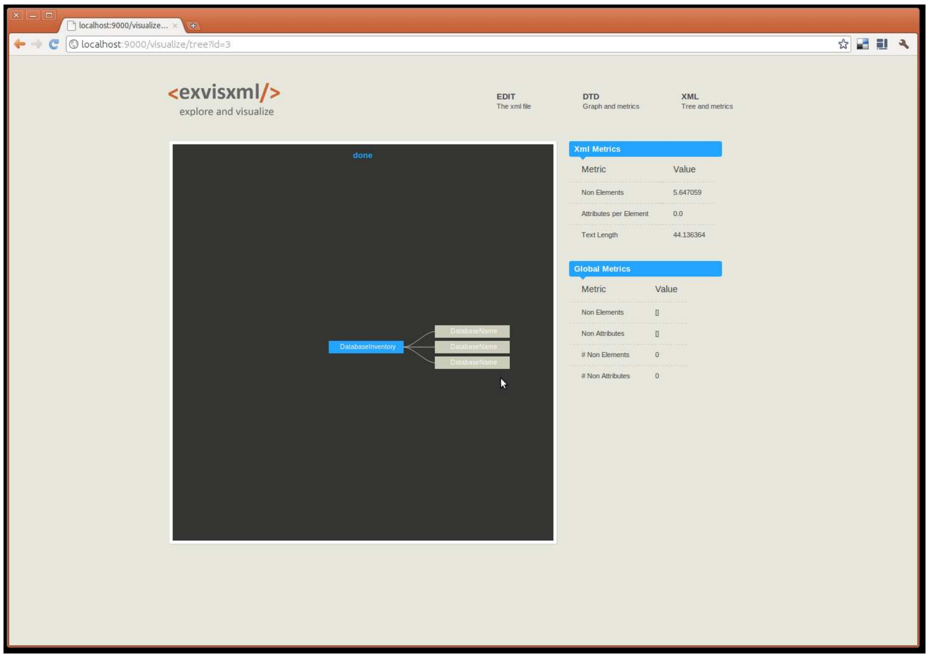

Figure 3: Tree representation for RJls Doc. — partial view (some nodes collapsed)

Javascript Infovis Toolkit[10] met all of our requirements, a number of visualization examples are provided in the toolkit page, each one caters to a specific data and structure.

It was developed using Javascript and so it was compatible with all browsers, a big plus as Prefuse Flare needed a compatible flash player for example. Javascript is client side, meaning that all the visualization are to be rendered by the client's computer, plus selective capacities such as "on-demand nodes" makes up for a much fluid user experience, in fact, documents that would struggle to pass the tests in the two previous toolkits had no problem, other than the occasional hiccups.

The utilization of Javascript Infovis toolkit meant however a redesign of the core application in order to deal with JSON documents capable of synthesizing the Tree and Graph structure of the XML and DTD documents respectively. Nevertheless the time spent adapting the visualizations in other toolkits was far more consuming than the time needed to adapt the core application. The adaptation proved successful and very important to the outcome of this project.

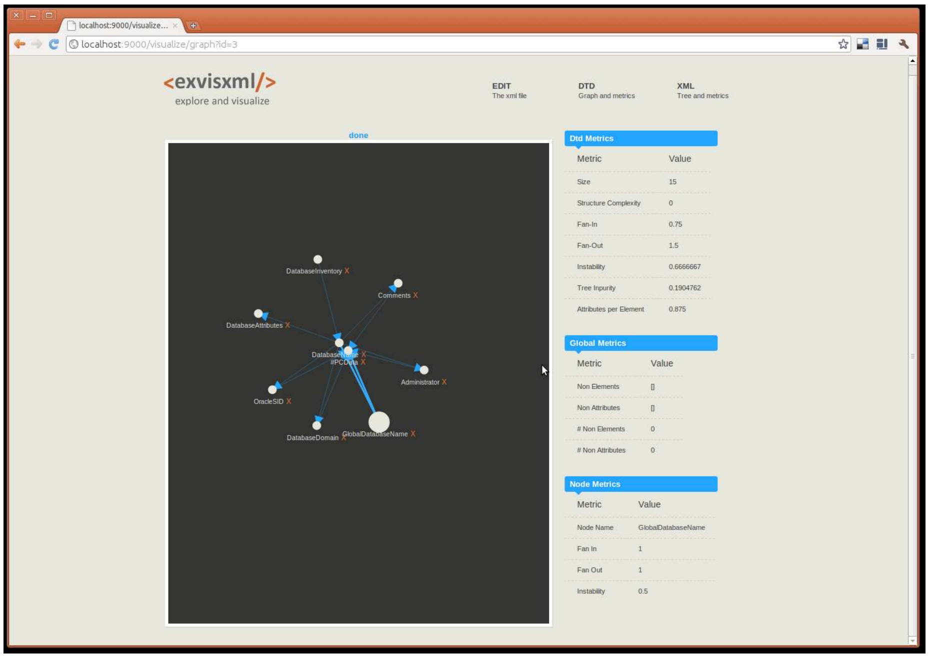

Figure 4: Sucessor Graph for RJls DTD

5.3 Slicing

According to a *slicing criterion* (see Section 3) 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, 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 an additional feature, eXVisXML can generate a new XML document including only the components along the pathes highlighted in the previous *slicing operation*. Notice that this new XML is also valid according to the submitted DTD, hence the structure of the XML document is not changed.

5.4 Metrics

There are two main data representation structures in eXVisXML it was obvious from the documents makeup that these were an n-ary tree and a unweighted directed graph, for the XML document and DTD respectively.

Each XML element is a node in the tree, each node has an id, name, a data field, and a list of children (other nodes). The id is used to distinguish nodes, as a XML document can have several elements with the same name, in this way we are able to reference the node should the need to alter back the document arises. For now the identifier is nothing more than the Java object's id and so the current implementation changes from different instances of the tree as to be expected, after all a new instance calls for a new Java object, the intention is to develop a sequential hash function, similar to the one being used to generate the object id, yet generating a constant id across instances, for an element of a certain XML document with a determined structure. The data field contains an element's text, in this case the node name will always be defined as pcdata.

The directed graph of the DTD document is comprised of a list of vertex, each vertex resembles the structure of the tree node discussed above, having an id, name and set of adjacent edges. The id concept applies here as well. Each adjacent edge has a "node to" and a "node from" field.

We have previously explained our decision in the use of Apache Axiom and pull parsers, it may seem contrary to our initial approach to develop the same structures present in a DOM parser, this can be, in fact, easily explained: these structures do not hang in memory as a DOM parser tree would for example, they are merely used for the calculation of metrics, they are completely disregarded after the process is completed, they do not possess the same amount of information either, being much slimmer they have in fact a more tenuous impact, performance wise.

Both of these structures were build from the scratch and they carry the data necessary for visual representation. The next obvious step was to fill in these structures with the data coming from both XML and DTD files, we developed several methods to do this, the ones we are interested in this section also initiate and perform some metric evaluations, the reason is simple: though we have a handful of metrics that require advanced algorithms most get by with simple summations.

Applying to the working example (the RJls screenplay in Figures 6 and 7, the set of metrics defined in Section 4, we obtain the measures listed in Table 2. To evaluate part of those parameter values it was necessary to build first the *sucessor graph* for the given DTD. Figure 4 sketches that SG. When the user selects the appropriate option from eXVisXML menu, our tool will compute automatically the referred metrics, and open a new tab with the values obtained.

Observing Table 2, some of the conclusions we can draw are:

Metric	Value
Size	27
Compl	13
Depth	7
FanIn(*scene*)	3
FanIn(*title*)	6
FanOut(*scene*)	6
FanOut(*title*)	0
Instability(*scene*)	3,3%
Instability(*title*)	0%
TI	58,9%
AttrsEle(DTD)	0,08
AttrsEle(XML)	0,027
NonAttr	0
NonElem	1
TxtLen	37,46
AttrTxtLen	1

Table 2: Metrics for the *Romeo and Juliet* screenplay

- From the *Size* metric, we conclude that the DTD has a medium size.
- From the *Compl* and *Depth* metrics, we conclude that the DTD as a considerable complexity, as the structure depth is in the borderline considered by *Klettke et al* in (Klettke *et al.*, 2002). In fact, looking at the SG in Figure 4, we can observe that there are many loops, which notably increases the DTD complexity.
- Comparing the values of *Instability* in nodes *scene* and *title*, we can say that the node *scene* has few dependencies with respect to other nodes, while *title* does not have any dependency (which is corroborated by the respective *FanOut*). So, we conclude that the resilience of these nodes is very low.
- Comparing the pairs *NonAttr/NonElem* (almost all attributes and elements are used) and *TxtLen/AttrTxtLen* (long PCData fields and short attribute values) we can conclude that the DTD was well designed. Specifically, from the *NonElem* metric and comparing the DTD against the XML, we conclude that the element stagedir under the context of the element line is never used.
- Parameters *AttrsElem(DTD/XML)*, confirm the easiness of using that DTD.

5.5 Validation

Unfortunately being a web application adds up to extra validations as restrictions have to apply in order to manage space and processing power, guaranteeing a limited scalability. The following validations are done on each session:

– Document input validation

– Document size validation

– Schema validation

– Syntactic validation

– Graph dimension validation

These are performed in a specific order, failing to conform to any of them results in an error message, preventing the user from going further in the visualization. The message contains enough information for the user to correct the error and try again.

The error messages distinguish themselves in two different groups, the ones who imply an hardware restriction, and the ones who imply a software restriction. For example, "Document size validation" and "Graph dimension validation", conform to hardware restrictions, as "Schema validation" and "Input validation" conform to a software restrictions.

The hardware restrictions prevent the user from inputting potential large documents, incapable of being processed in an web environment. Software restriction prevent the user from inputting erroneous documents and thus generating inaccurate or even absent results, given that the system was not able to parse them. We shall go threw each restriction one by one and analyze them in sufficiently.

Document input validation - When the user is prompted to upload the documents, it is provided with two input forms, one for the XML and another for the DTD document, failing to fill any of these two forms will result in validation error.

Document size validation - The system will prevent any XML document that exceeds the size of 100KB to be uploaded. This size limit was determined through try and fail. While conditions may vary, standard internet connection and general hardware capacities should be representative of the system bottlenecks and performance. There are mainly two bottlenecks inherent to the software: the metrics calculation or re-calculation, that involves graph and tree traversals; and retrieving the XML tree to the Javascript Infovis toolkit. Some measures can be taken, and were taken to alleviate these problems although they do not solve them completely.

The metrics calculation is a necessary burden, in order to reduce its impact we resort to the use of memory cache. The algorithm is simple, the calculations are performed on start and stored in cache, from that moment on the metrics are only re-calculated on a per-need basis, for example, if the user alters the xml document and saves it, or on a cache miss, that is, if the application doesn't find the correspondent metric instance on cache.

Retrieving the XML tree to the Javascript Infovis toolkit happens when some modification is done to the XML document. The tree is passed on to the user's browser as a JSON file, this file is generally bigger than the corresponding XML document, this should not represent any bottleneck in a high bandwidth internet connection providing the limit of 100KB XML document. In tests, considerably bigger documents stalled the navigation of a XML tree, these occurs as the JSON file is stored in browser memory, running on the Javascript machine. We can reduce the download time using Apache's mod_gzip, effectively compressing the JSON file, this method cannot provide consistent results though, it should however help in some particular cases.

Schema validation - In order for the metrics to have a meaning, consistent data must be provided. Therefore we verify that a given DTD schema is according with the provided XML document. The application does not have the capacity to parse the XML document for Schema declarations, the DTD Schema must be yielded as a separate file.

Syntactic validation - A requirement for any parser to perform his job. Syntactic checking is done after uploading a document, if the test fails, the document is discarded and the user is solicited to make the necessary corrections and upload again.

Graph dimension validation - One of the metrics, graph depth, requires the calculation of the longest path, this calculation is known to be a NP complete problem. It's polynomial nature means that the number of vertex composing the graph have to be controlled. As we cannot evaluate the impact a small increase in vertex number has, we need to limit these feature to a predefined number of vertex. If that number is suppressed the user will be notified that feature was turned off.

5.6 Technologies used

From a technical point of view, eXVisXML consists of two distinct applications. Application number one, the core application, is composed of eXVisXMLś algorithms, entities, data structures and data access routines. Application number two, the web front end, consists of a MVC structured framework, a simple database and a visualization toolkit.

Application number one is designed as a three layer application, providing a simple API to eXVisXML rough capabilities. Application number two uses the core application as a library, passing in arguments and receiving processed data. The application as the following structure as depicted in Figure 5

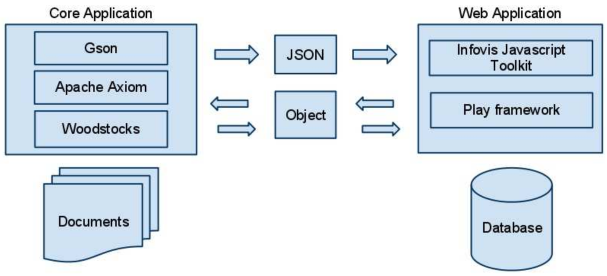

Figure 5: Technologies

Apache Axiom takes advantage of woodstocks as an alternative to the standard StAX implementation, providing StAX2 capability, a slightly faster approach, though not the main selling feature of woodstocks, in our case. Woodstocks provides us with the capability to validate an XML document against a number of schemas, for now we are only interested in DTD schemas, none the less the alternatives gives us assurance for future expansions.

The bridged between the two applications is two folded, standard java object instances and JSON documents. While it may seem self explanatory, or even obvious that the communication is done by instantiating objects between the two applications, it is not obvious why is there exchange of JSON documents, and so it is necessary that we explain its utilization and consequences.

JSON or Javascript Object Notation is more than a markup language, by providing a textual means for displaying the object structure of a javascript object, JSON is the ideal candidate to pass along data between two different Javascript applications.

Ironically JSON is quite an alternative to XML, providing a much simpler format. JSON as been growing over XML mainly in web services applications where a simple and lightweight format is of upmost importance. JSON is also the format used to describe trees and graphs in the infovis javascript toolkit.

Naturally, the representation of both the XML and DTD documents present in the core application is different of the one in infovis javascript toolkit. In order to exchange information between the two applications there was the need to transform the data structures, and thus the need for Gson. Gson is a Google project consisting of a Java library capable of converting Java Objects to JSON and vice-versa. By converting the internal representation of the XML tree and DTD graph to JSON we are able to port this description on the fly to the Javascript Infovis Toolkit, furthermore the inverse, though harder is also feasible, and so if the need arises the structure in hand should be easily adoptable to implement that specific feature.

The Javascript InfoVis Toolkit has a number of different views, the ones chosen to represent our visualizations where the SpaceTree and ForceDirected, for the XML and DTD document respectively. The SpaceTree visualization has the most interesting feature of the two: on demand nodes. On demand nodes is a technique that makes the navigation of large trees visually appealing and fluid, it works by retrieving the children nodes as the father gets selected.

Finally, the web application provides us with a graphical interface. We choose the Play framework[11] as a simple Model-View-Controller web framework that has the advantage of being built in Java, integrating easily with our core application.

Play framework has other benefits, it lets us create and interact with a database easily, define a implement an application logic in a clean way, do quick and easy field validation and import extra functionality through a plugin system, in the end helping us focus on the real features.

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 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 impor-

[11] http://www.playframework.org/

tance of the slicing functionality for document understanding and re-engineering.

```xml
<?xml version="1.0"?>
<!DOCTYPE DatabaseInventory SYSTEM "DatabaseInventory.dtd">

<DatabaseInventory>

  <DatabaseName>
    <GlobalDatabaseName>production.iDevelopment.info</GlobalDatabaseName>
    <OracleSID>production</OracleSID>
    <DatabaseDomain>iDevelopment.info</DatabaseDomain>
    <Administrator EmailAlias="jhunter" Extension="6007">
        Jeffrey Hunter
    </Administrator>
    <DatabaseAttributes Type="Production" Version="9i"/>
    <Comments>
      The following database should be considered the most stable for
      up-to-date data. The backup strategy includes running the database
      in Archive Log Mode and performing nightly backups. All new accounts
      need to be approved by the DBA Group before being created.
    </Comments>
  </DatabaseName>

  <DatabaseName>
    <GlobalDatabaseName>development.iDevelopment.info</GlobalDatabaseName>
    <OracleSID>development</OracleSID>
    <DatabaseDomain>iDevelopment.info</DatabaseDomain>
    <Administrator EmailAlias="jhunter" Extension="6007">
        Jeffrey Hunter
    </Administrator>
    <Administrator EmailAlias="mhunter" Extension="6008">
        Melody Hunter
    </Administrator>
    <DatabaseAttributes Type="Development" Version="9i"/>
    <Comments>
      The following database should contain all hosted applications. Production
      data will be exported on a weekly basis to ensure all development environments
      have stable and current data.
    </Comments>
  </DatabaseName>

  <DatabaseName>
    <GlobalDatabaseName>testing.iDevelopment.info</GlobalDatabaseName>
    <OracleSID>testing</OracleSID>
    <DatabaseDomain>iDevelopment.info</DatabaseDomain>
    <Administrator EmailAlias="jhunter" Extension="6007">
        Jeffrey Hunter</Administrator>
    <Administrator EmailAlias="mhunter" Extension="6008">
        Melody Hunter</Administrator>
    <Administrator EmailAlias="ahunter">
        Alex Hunter</Administrator>
    <DatabaseAttributes Type="Testing" Version="9i"/>
    <Comments>
      The following database will host more than half of the testing
      for our hosting environment.
    </Comments>
  </DatabaseName>

</DatabaseInventory>
```

Figure 6: The document depicts a database inventory, annotated according to the markup language defined by the DTD in Figure 7.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT DatabaseInventory (DatabaseName+)>
<!ELEMENT DatabaseName (   GlobalDatabaseName
                         , OracleSID
                         , DatabaseDomain
                         , Administrator+
                         , DatabaseAttributes
                         , Comments )
>
<!ELEMENT GlobalDatabaseName (#PCDATA)>
<!ELEMENT OracleSID          (#PCDATA)>
<!ELEMENT DatabaseDomain     (#PCDATA)>
<!ELEMENT Administrator      (#PCDATA)>
<!ELEMENT DatabaseAttributes EMPTY>
<!ELEMENT Comments           (#PCDATA)>

<!ATTLIST Administrator       EmailAlias CDATA #REQUIRED>
<!ATTLIST Administrator       Extension  CDATA #IMPLIED>
<!ATTLIST DatabaseAttributes  Type       (Production|Development|Testing) #REQUIRED>
<!ATTLIST DatabaseAttributes  Version    (7|8|8i|9i) "9i">

<!ENTITY AUTHOR "Jeffrey Hunter">
<!ENTITY WEB    "www.iDevelopment.info">
<!ENTITY EMAIL  "jhunter@iDevelopment.info">
```

Figure 7: DTD for the XML document in Figure 6.

References

Alves, Tiago, & Visser, Joost. 2005 (Maio). *Metrication of SDF Grammars.* Research Report. Departamento de Informática, Universidade do Minho.

Chamberlin, D. 2002. XQuery: An XML query language. *IBM Syst. J.*, **41**(4), 597–615.

Clark, James, & DeRose, Steve. 1999. *XML Path Language (XPath) Version 1.0.* Tech. rept. World Wide Web Consortium.

Klettke, Meike, Schneider, Lars, & Heuer, Andreas. 2002. Metrics for XML Document Collections. *Pages 15–28 of: EDBT '02: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers.* London, UK: Springer-Verlag.

Koutsofios, Eleftherios, & North, Stephen. 2002. *Drawing graphs with* dot.

Lämmel, R., Kitsis, Stan, & Remy, D. 2005 (Novembro). Analysis of XML schema usage. *In: Conference Proceedings XML 2005.*

Mertz, David. 2001. Transcending the limits of DOM, SAX, and XSLT: The HaXml functional programming model for XML. *IBM developerWorks (XML Matters column)*, October.

Olteanu, D., Meuss, H., Furche, T., & Bry, F. 2002. *Xpath: Looking forward.*

Silva, Josep. 2005. Slicing XML Documents. *Pages 121–125 of: WWV.*

Silva, Josep. 2006. *XMLSlicer.* http://www.dsic.upv.es/ jsilva/xml/.

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

Visser, Joost. 2006 (Fev). Structure Metrics for XML Schema. *In: XATA - XML: AplicaCções e Tecnologias Associadas, Portalegre - Portugal.*