A Formalism for Navigating and Editing XML Document Structure

Frithjof Dau, Mark Sifer

School of Information Systems and Technology
University of Wollongong, Australia
{dau,msifer} (at) uow.edu.au

Abstract. The use of XML has become pervasive. It is used in a range of data storage and data exchange applications. In many cases such XML data is captured from users via forms or transformed automatically from databases. However, there are still many situations where users must read and possibly write their own XML documents. There are a variety of both commercial and free XML editors that address this need. A limitation of most editors is that they require users to be familar with the grammar of the XML document they are creating. A better approach is to provide users with a view of a document's grammar that is integrated in some way to aid the user. In this paper, we formalise and extend the design of such an editor, Xeena for Schema. It uses a grammar tree view to explicitly guide user navigation and editing. We identify a key property that such an editor should have, stable reversable navigation, then via our formal treatment extend the Xeena for Schema design to satisfy it.

1 Introduction

The eXtensible Markup Language (XML) [16] has become a standard foundation for both data exchange and electronic documents. Many XML languages for display and data exchange have been defined: XHTML [19] for web browsers, SVG and X3D for 2D and 3D graphics, MathML for mathematical equations, and DocBook for written documents such as technical reports. Documents in these languages can be displayed through suitably configured web browser or dedicated viewers. XML is already adopted by OpenOffice 2.0, and Microsoft is adopting XML for it's office applications as well, as they define languages for files created by their word processor and spreadsheet applications. Universal business language (UBL) is one of many languages developed for the exchange of business data. These are only a small selection of the many document and data oriented XML languages that exist.

Dedicated tools usually create XML documents. Web authoring tools provide a WYSIWYG interface that allows users to drag and drop web page elements from a toolbar and save the generated web pages as XHTML documents. Similar tools support other document-oriented languages such as MathML and DocBook. Specific business tools and applications convert data from a variety of sources into XML languages for exchange and processing. Another approach for the entry of data documents is web based interactive forms generated from a DTD or schema by systems such as XForms [18] and Forms-XML [7].

There are XML languages for which dedicated editors are either not available or not freely available. Because an XML document is a text file, it can be created and edited with any text editor. An XML enabled web browser such as Microsoft's Internet Explorer can check it for well-formedness and validity. However, this requires authors to observe XML syntax and grammar constraints as they create and edit. Authors that rarely use XML must learn how to write with XML tags and attributes, while authors that are familiar with XML must still learn the document's grammar. If an author does not use a language regularly or the language itself is very large or complex this can be a large learning task. A better choice is an XML editor.

XML editors integrate knowledge of XML syntax and grammar constraints when given a DTD or Schema. They support the creation of arbitrary XML documents, providing automatic assistance to keep generated documents well formed and valid. The significant number of commercial XML editors [15, 20, 21] and free XML editors [2, 17] that are currently available, confirm user demand for such editors. These editors provide a grammar view that shows one level of potential elements.

The Xeena for Schema XML editor [12, 13] integrates a view of a document's grammar to aid the user. It uses a more powerful deeper grammar view that explicitly assists navigation and guides editing. This paper extends earlier work [12, 13] in two respects. Firstly, we present here a *formal* treatment of Xeena for Schema. We identify and formally prove nice key properties of the original Xeena for Schema design. Our formalism describes a grammar view tree, a document view tree and the structure preserving mappings between them that facilitate navigation and editing. Secondly, though a navigational limitation of the original design is identified, we show how this limitation can be overcome.

2 Related Work

When text documents need to conform to a grammar, authors need to know the grammar or receive some interactive assistance, in order to create them manually. Such documents include: computer source codes that must conform to a particular programming language, technical documents for which allowable structures have been defined, and more recently XML documents. Many editors that provide such interactive assistance with some kind of grammar directed editing have been created. The earliest were syntax directed program source code editors [3, 14, 22] which have had limited success, as evidenced by the fact that existing commercial program editors do not use grammar directed editing, but typically support syntax highlighting or keyword completion only. Many editors for structured documents, usually large technical document have also been developed. These have been more successful. In this section we review these document editors and the reasons for their relative success. We also review recent XML navigation tools and editors.

2.1 Structured Document Editors

Grammar based editors called structured document editors within the technical document community have been successful. This is evidenced by the range of commercial and free structured document editors such as SGML editors currently available. A key driver in the design of document preparation systems was the need to separate content from presentation, so that the same content could be re-used in a variety of settings. This required content to be structured in some way, so document portions could be referred to or located in a systematic way for presentation processing. Content was represented as a tree of portions or elements whose arrangement satisfied a grammatical structure.

Unlike computer programs, structured documents contain markup; start tags and end tags that delimit an element and explicitly type the enclosed content. In many document languages, each grammar rule corresponds to an element in a document. This means that regular grammars are often sufficient for many structured document languages rather than the richer context free grammars required for programming languages. For most computer programming languages, the non-terminal grammar rules have no corresponding visible artifact in a program. To contrast, a structured documents nested element tree can reveal its grammar while a program sources lack such an explicit tree structure. This suggests users will have greater success manipulating a structured document's more visible element structure via its grammar.

Grif [6] is an early structured document editor. It provides several document views. A plain text global view for content entry, an outline or table of contents view, a presentation view that shows final presentation and a specialised view for editing mathematical formula. During editing in the global view a popup menu shows the elements that can be validly added or inserted at the current document location, while the various views are coordinated around this current location. Grif does not provide a raw text view that includes markup tags so its documents are always well formed, and because only valid edit operations are offered the element structure is always consistent with the grammar. Grif has also been proposed as a HTML editor for web documents [10]

Another early structured document editor is Rita [5]. Its authors noted "since various types of documents require tags with different placement, the creator of a document must learn and retain a large amount of knowledge". That is, because many documents grammars are large, it can be demanding for users to be familiar enough with a document's collection of tags to manually create one. To aid users, the Rita interface provided a presentation view, an element structure view, and a dynamic menu of element tags. A user could select an element in the structure view and be presented with the choice of elements that could be validly inserted. Users could also transform elements via a menu of valid transformations. To support greater editing flexibility invalid portions of a document could be marked with a special tag so they were treated as text, while a patch area allows arbitrary cut and pasting. Both Grif and Rita guide user editing with one visible grammar level.

2.2 Querying and Mining XML Structure

An integrated grammar view can assist user navigation of large structured documents. It can assist navigation from one instance of a grammar rule such as a section header to the next or previous instance. BBQ [8] is such a system for navigating and querying XML documents. It presents tree views of both a document and its grammar. The grammar view recursively presents each grammar rule as a node in an indented tree. The grammar tree view provides an overview of the grammar a user can explore to an arbitrary depth while also supporting navigation. BBQ also supports join-based queries that a user builds via multiple grammar tree views. Its focus is the application of a grammar tree view to the querying of large XML documents, while the focus of our work is the application of a grammar tree view to XML document editing.

Large grammars are also difficult to work with. Many DTDs and XML Schema are so large that most document instances only use a small subset of the defined elements. In such cases, presenting an author with all elements that can be added may be overwhelming. The difficulty is that most grammars are designed to organise content. They are not designed explicitly for visual presentation to a user. An alternative approach is to study the patterns of element use in a document, and initially only propose those elements or tree patterns the user is likely to use next. Because this approach works by studying existing document content it can be applied to XML documents have no DTD or XML Schema defined. This is the approach Chidlovskii's structural advisor for XML document authoring [4] takes. This approach could be integrated with BBQ style grammar tree view, by restricting the displayed grammar tree to nodes for common elements only. This would be a useful extension of our work.

2.3 XML Editors

A wide variety of Commercial and free XML editors that provide grammar assistance for editing are available. Some are element structure-oriented while others are content-oriented. Xeena [17] is a structure-oriented editor. Editing is done within a tree view of document elements. Users are guided with a DTD determined list of possible elements, shown in a separate panel. Users can select one of these potential elements and add or insert around the current document position. The Topologi document editor [15] is a content oriented editor for progressively convert arbitrary text documents into valid XML documents. It allows users to rapidly apply markup to large documents. Its primary editing view is a raw text view. It also offers grammar (SGML and XML) guidance when adding elements, providing users with a choice of valid elements.

Editors can also provide multiple editable views. XMetal [20] provides several editable views: a structure view, formatted view and formatted view with exposed tags. Like Xeena it offers a grammar sensitive list of valid elements to guide editing. XMLSpy [21] provides two editable document views: an indented tree view and a nested tree element view. It also offers a grammar sensitive list of valid elements to guide editing. The Amaya editor [2, 11] provides formatted, structure, source text and specialised views for markup languages such as XHTML, SVG and MathML. It also provides a table of contents view to aid rapid navigation. All its views are editable. Again, elements can be added by choosing from a list of valid elements. Elements can also be added manually, in which case, the system tries to create a series of in-termediate elements down to the chosen type based on the grammar.

These editors use a document's grammar to determine the element list presented in a menu or panel that can be added into a document. Users make valid edits by inserting them. However, such a list provides a narrow view of the grammar that is only one level deep. It also does not indicate the grammar role of each potential element, whether it required, optional, repeatable or a choice. Our editor design is structure-oriented and uses a deeper BBQ like multi-level grammar tree view annotated with grammar roles to better support structure based navigation and editing. A design that we refine and formalise in this paper.

3 Building and Navigating

This section demonstrates Xeena for Schema editing and navigation to provide a context for the formal treatment.

3.1 Starting

On startup Xeena for Schema presents a document and its grammar. Figure 1 shows a new report document that contains a title and empty body in the right panel and its grammar tree in the left panel. The indented grammar tree has been manually unfolded to show several levels of the report grammar. It provides an overview that guides user editing. The grammar view uses the visual syntax shown in figure 2. In XML grammars, each element has a prefix called a *cardinality operator* that indicates how many times it may occur in a document according to the grammar. An '?' indicates that it is optional, i.e. it can occur zero times or once; an '*' indicates that is can occur arbitrary often, including zero times; and an an '+' indicates that is can occur arbitrary often, but it has to occur at least once. If no cardinality operator is given, the new element has to occur exactly one. In our ongoing formalization, this will be denoted by '1'. Besides the cardinality, each child element participates in either a sequence or a choice. In Xeena for Schema, round parentheses indicate the former and square parentheses indicate the latter. A grammar may be recursive. For example if the List node in figure 1 were unfolded it would contain another List node, and so on without limit. The grammar view also shows which elements have been instantiated in the document. Instantiated nodes have coloured icons while other nodes icons are grey. In figure 1 the document only contains a report, title and empty body element. Only their corresponding nodes in the grammar view are shown red. The grammar tree view provides both a comprehensive overview of potential structure and an indication of what has been instantiated so far.

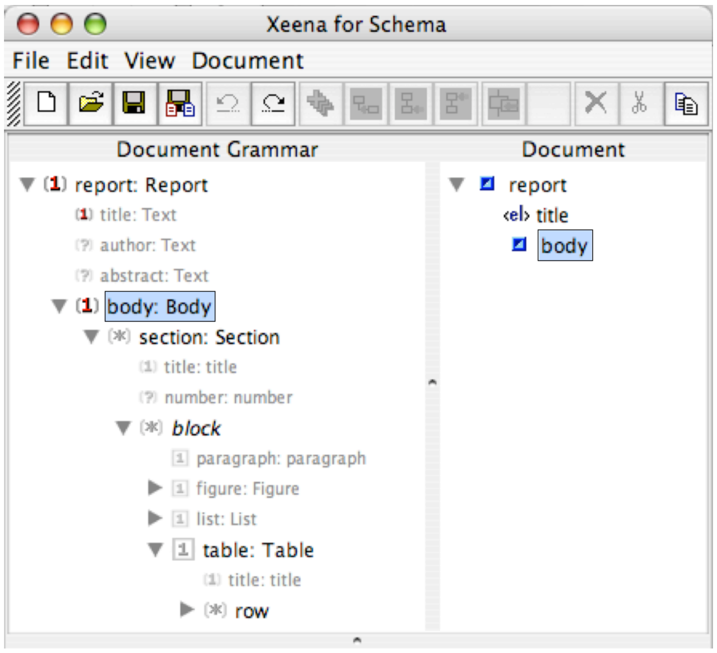

Fig. 1. A new report document

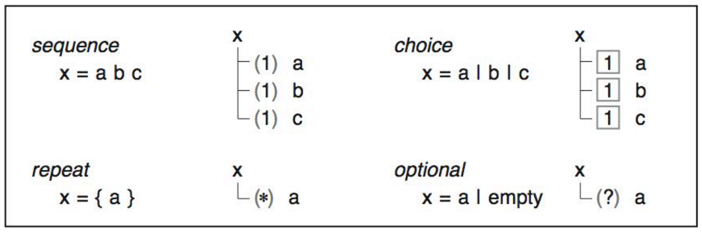

Fig. 2. Visual syntax of our grammar view

3.2 Building

A document is built by adding elements. The grammar tree view shows which elements can be added. Any grammar node that can occur multiple times or any node that can occur once but has not been instantiated (shown as grey) can be added. In figure 1 the report grammar was unfolded five levels deep to show the row elements. A user adds three row elements by just selecting it and clicking the add toolbar icon three times. Figure 3 shows the result. Three row elements, all required intermediate elements and all required compulsory elements are added by the editor.

An XML grammar may be recursive. Our grammar view supports this. For example in a report document, a list contains listItems which contain text and

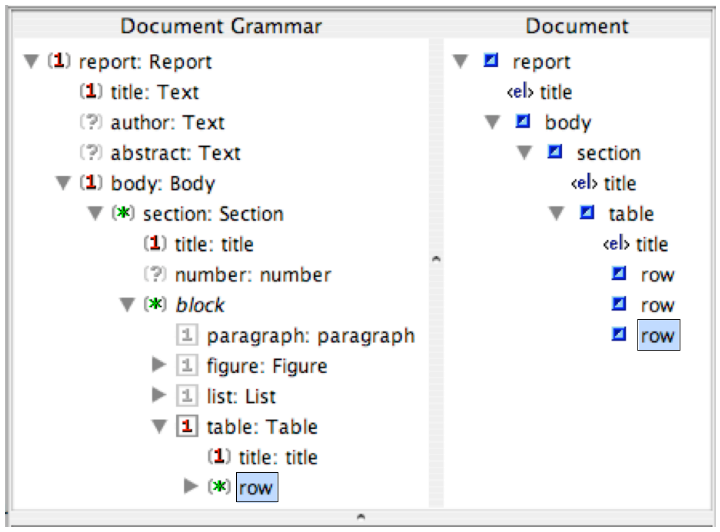

Fig. 3. The document after adding 3 rows

an optional list. Our design does not address the general recursive case directly because a user will never unfold a recursive grammar tree to an infinite depth, but it does address the finite unfoldings of a recursive grammar that a user does when they progressively open the grammar tree. For example when a user opens the list node in the grammar view its child node listItem becomes visible, then after opening listItem its children a text node and a list node become visible, then this list node could be opened; increasing the depth of the visible tree further. We note that grammar tree nodes always have unique path names, for example the path names section.list and section.list.listItem.list are distinct.

3.3 Unstable and Stable Navigation

The grammar view always shows the surrounding context for the current document cursor element. After adding the three row elements in figure 3, if the user selects the table node in the grammar view, the document cursor will move to the corresponding table element. Figure 4 shows this. To reverse this, the user can select the third row element in the document view again. The result would be figure 3 again. However, with Xeena for Schema if the user attempts to do this via the grammar view, i.e., if he clicks in the last step not on the third row element in the document view, but on the row element in the grammar view instead, the behaviour is different. The user would expect to return to the third row after this up-down navigation via the grammar view, but instead the document cursor has lost it's initial position (third row) and ended up on the first row. This is shown in figure 5. This is a limitation of the Xeena for Schema design that our formal treatment will characterise and address.

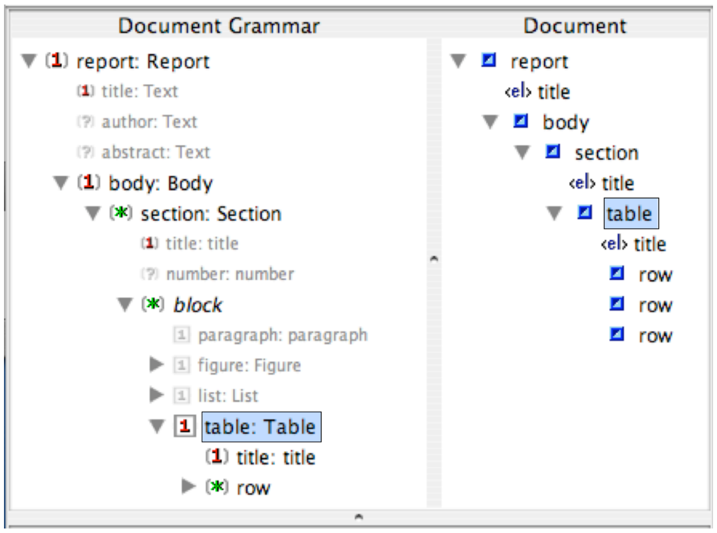

Fig. 4. The user selects the table node in the grammar view movings the document cursor to the corresponding table element

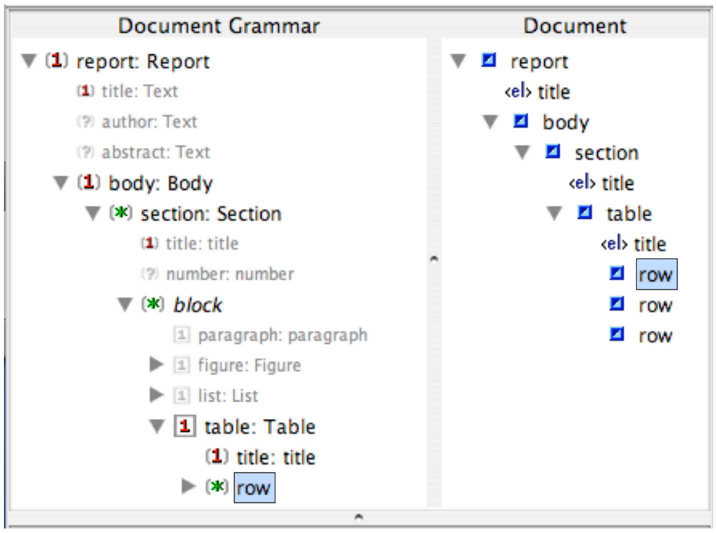

Fig. 5. The user selects the row node again in the grammar view moving the document cursor to the first row element.

Figure 6 outlines a stable navigation design that addresses this. The document cursor is separated into two parts: (i) the element the user has clicked (shown with a black rectangle) which established the range of interest and (ii) the original document cursor (shown with blue fill). Navigation in the grammar view only changes the document cursor, leaving the range cursor unchanged. The range cursor only changes when the user manually selects an element in the document view. In figure 6 (i) both of cursors start as element c_2 but separate in (ii) after b is selected in the grammar view, into document cursor b and range cursor c_2. In (iii) after c is selected in the grammar view, the document cursors returns to c_2 it's initial position, while the range cursor has not changed and so remains positioned at c_2. This downwards navigation has retained the initial context, set when the user selected the c_2 element in the document view.

If the user selects a non-comaprable node in the grammar view there are several approaches which preserve stable navigation. We describe these alternative approaches for stable navigation later after providing our formal treatment.

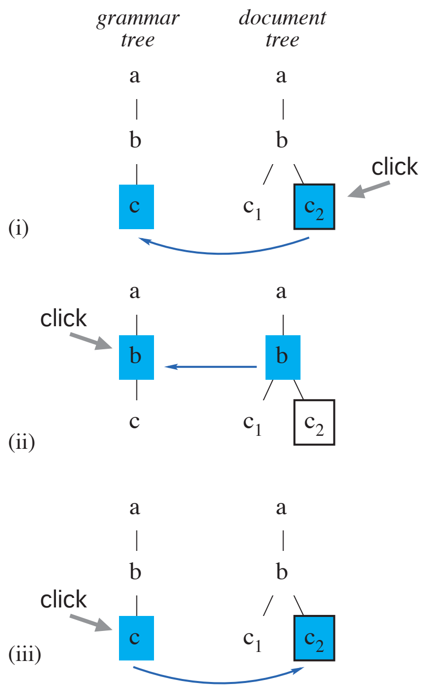

Fig. 6. Stable navigation separate the document cursor into document cursor and range cursor

4 Labelled Ordered Trees

In this section, XML grammars and XML documents are formalized as LABELLED ORDERED TREES, according to the notions of [1] or [9]. We assume that the reader is familiar with trees. To clarify matters, trees are in our approach formalized as posets $(P, \leq)$, where the smallest element of P is the ROOT of the tree. For $p \in P$, we set $\downarrow p := \{q \in P \mid q \leq p\}$ and $\uparrow p := \{q \in P \mid q \geq p\}$. Technical terms like UPPER/LOWER NEIGBOURS, MINIMAL/MAXIMAL ELEMENTS, CHILDREN, SIBLINGS and LEAVES etc. are defined as usual. The INFIMUM of two elements p and q is denoted by $p \wedge q$. For $p, q \in P$ with $q < p$, let q^p denote the uniquely given upper neighbour of q with $q < q^p \leq p$. In the Hasse-diagram representation of trees, we will draw them 'upside down', i.e., the bottom element of the tree is at the top of the representation.

Each element $p \in P$ gives rise to the PATH $\downarrow p$. In the following scrutiny on XML-trees, we are more interested in the paths than the elements of P, but as the elements stand in one-to-one correspondence to the paths, we do not provide a notation for the paths on its own. Note we have $p \leq q \Longleftrightarrow \downarrow p \subseteq \downarrow q$.

Labelled ordered trees will be defined to be trees, where the nodes in the tree correspond to XML elements, their labels denote the type names of the elements. Given a node, its children are totally ordered. For the ongoing formal treatment, it is convenient to extend this additional order to all elements of the tree.

Definition 1 (Additional Order on Trees). *Let $(P, \leq)$ be a tree. A partial order $\sqsubseteq$ on P is called* ORDER ON SIBLINGS, *iff*[1] *whenever two distinct elements p, q are comparable w.r.t. $\sqsubseteq$, they are siblings. If moreover siblings are always comparable w.r.t. $\sqsubseteq$, we call $\sqsubseteq$ a* TOTAL ORDER ON SIBLINGS, *and $(P, \leq, \sqsubseteq)$ is called* ORDERED TREE.

To each order on siblings $\sqsubseteq$, we assign an extension $\sqsubseteq^{*}$ of $\sqsubseteq$ as follows: For two incomparable (w.r.t. $\leq$) elements $p_1, p_2 \in P$, let $q := p_1 \wedge p_2$, and we set $p_1 \sqsubset^{*} p_2 :\Leftrightarrow q^{p_1} \sqsubset q^{p_2}$. Let $\sqsubseteq^{*}$ be the reflexive closure of $\sqsubset^{*}$.

In the Hasse-diagram representation of trees, given two siblings p and q, we will draw p to the left of q iff $p \sqsubseteq q$.

Of course, $\sqsubset^{*}$ is not simply another order on P, but it has to respect some additional restrictions. For example, if a path A is left to a path B, then each subpath of A should be left to B as well. We first fix this idea.

Definition 2 (Respecting $\leq$). *A relation $R \subseteq P \times P$* RESPECTS $\leq$, *if we have $R \cap \leq = \Delta_P$ $(:= \{(p,p) \mid p \in P\})$, and for all p, q, p', q' with $p \neq q$, $p \leq p'$ and $q \leq q'$, we have $pRq \Leftrightarrow p'Rq'$.*

Now, given a tree $(P, \leq)$, the orders R on P which respect $\leq$ are exactly the orders obtained by extending orders on siblings. Moreover, such an R is obtained from extending a *total* order on the siblings iff R is a total order on the leaves of P. This is captured by the following lemma.

[1] iff: if and only if

Lemma 1 (Extending Sibling Orders). *Let $(P, \leq)$ be a tree, let $\sqsubseteq$ be an order on siblings of P. Then $\sqsubseteq^*$ is an order extending $\sqsubseteq$ which respects $\leq$. Moreover, $\sqsubseteq$ is a* total *sibling order iff $\sqsubseteq^*$ is a* total *order on the leaves.*

Vice versa, let $(P, \leq)$ be a tree and R be an order on P which respects $\leq$. Then there exists a order $\sqsubseteq$ on the siblings with $R = \sqsubseteq^*$.

Proof: We start with the first propostion. We first show that $\sqsubset^*$ is a strict order. As $\sqsubset^*$ is defined only on pairs of incomparable (w.r.t. $\leq$) elements, it is obviously irreflexive. Now let $p_1 \sqsubset^* p_2 \sqsubset^* p_3$; we have to show $p_1 \sqsubset^* p_3$. Let $q_1 := p_1 \wedge p_2$, $q_2 := p_2 \wedge p_3$, and $q := p_1 \wedge p_3$. We have $q_1^{p_1} \sqsubset q_1^{p_2}$ and $q_2^{p_2} \sqsubset q_2^{p_3}$. Moreover, as $q_1, q_2 \leq p_2$, we see that q_1 and q_2 are comparable. We do a case distinction. For $q_1 = q_2$, we have $q = q_1 = q_2$, thus $q^{p_1} \sqsubset q^{p_2} \sqsubset q^{p_3}$, hence $q^{p_1} \sqsubset q^{p_3}$, thus $p_1 \sqsubset^* p_3$. For $q_1 < q_2$, we have $q = q_1$, thus $q^{p_1} \sqsubset q^{p_2} = q^{p_3}$, hence $q^{p_1} \sqsubset q^{p_3}$, thus $p_1 \sqsubset^* p_3$. The case $q_1 > q_2$ is done analogously to the last case. So $\sqsubset^*$ is a strict order, thus $\sqsubseteq^*$ is an order.

Obviously, for siblings p_1, p_2 and $q := p_1 \wedge p_2$, we have $p_1 = q^{p_1}$ and $p_2 = q^{p_2}$, thus $\sqsubseteq^*$ is indeed an extension of $\sqsubset^*$. Moreover, $\sqsubseteq^*$ respects $\leq$ by definition.

Next, let $\sqsubseteq$ be a total sibling order, and let p_1, p_2 be two distinct leaves of P. Then p_1 and p_2 are incomparable w.r.t. $\leq$. Let $q := p_1 \wedge p_2$, thus $q < p_1, p_2$. As q^{p_1}, q^{p_2} are siblings, they are comparable with respect to $\sqsubset$, thus, due to the definition of $\sqsubset^*$, p_1 and p_2 are comparable with respect to $\sqsubset^*$. Thus $\sqsubseteq^*$ is a total order on the leaves.

Vice versa, let $\sqsubseteq^*$ be a total order on the leaves, and let p_1, p_2 be two siblings. Then there are leaves p_1', p_2' with $p_1 \leq p_1'$ and $p_2 \leq p_2'$. Now p_1' and p_2' are comparable with respect to $\sqsubseteq^*$, and $\sqsubseteq^*$ respects $\leq$, so p_1 and p_2 are comparable with respect to $\sqsubseteq$.

As now the first proposition of the lemma is proven, we continue with the second. Let R be an order on P which respects $\leq$. Let $\sqsubseteq$ be the restriction of R to siblings. Let p_1, p_2 distinct elements of P. If p_1, p_2 are incomparable (w.r.t. $\leq$), for $q := p_1 \wedge p_2$ we have

$$p_1 \, R \, p_2 \overset{R \text{ resp.} \leq}{\iff} q^{p_1} \, R \, q^{p_2} \overset{Def. \sqsubseteq}{\iff} q^{p_1} \sqsubseteq q^{p_2} \overset{\text{Def. } \sqsubseteq^*}{\iff} p_1 \sqsubseteq^* p_2$$

If p_1, p_2 are comparable (w.r.t. $\leq$), then $(p_1, p_2) \notin R$ by Def. 2 and $(p_1, p_2) \notin \sqsubseteq^*$ by Def. 1. We conclude $R = \sqsubseteq^*$. $\square$

So a labeled ordered tree could be alternatively be defined to be a tree with an additional order which respects $\leq$ and which is a total order on all leaves of the tree. To ease the notation, if an ordered tree $(P, \leq, \sqsubseteq)$ is given, we will from now on identify the total order $\sqsubseteq$ on the siblings with its extension $\sqsubseteq^*$. Note that for two distinct nodes p, q, they are incomparable w.r.t. $\leq$ iff they are comparable w.r.t. $\sqsubseteq$.

Definition 3 (Labeled Ordered Tree). *Let $\mathcal{L}$ be a set of* LABELS, *denoting the type names of the elements. An* ORDERED, LABELED TREE (LOTO) *is a structure $(P, \leq, \sqsubseteq, l)$ where $(P, \leq)$ is an ordered tree and $l : P \to \mathcal{L}$ is a* LABELING FUNCTION.

Let $(P, \leq^P, \sqsubseteq^P, l^P)$ and $(Q, \leq^Q, \sqsubseteq^Q, l^Q)$ be ordered, labeled trees. We say that $p \in P$ and $q \in Q$ are CONGRUENT *and write $p \cong q$ iff the total orders $\downarrow p$ and $\downarrow q$ are 'the same sequence of labels', i.e., there exists a order-isomorphism $i : \downarrow p \rightarrow \downarrow q$ with $l^Q(i(s)) = i(l^P(s))$ for all $s \leq p$. An ordered, labeled tree $(P, \leq, \sqsubseteq, l)$ is called* PURIFIED*, iff there are no distinct, congruent nodes. Finally, for two distinct siblings p, q, we set $p \sqsubseteq_l q \;:\Longleftrightarrow\; p \sqsubseteq q$ and $p \cong q$.*

The acronym 'loto' is adopted from [9] and stands for 'labeled ordered tree object'. Note that, similarly to $\sqsubseteq^{*}$, the order $\sqsubseteq_l$ respects $\leq$.

Now we are finally prepared to define XML grammars and documents by means of labeled ordered trees.

Definition 4 (XML-Trees). *A* DOCUMENT TREE *is a nonempty labeled ordered tree. A* GRAMMAR TREE *is a structure $(P, \leq, \sqsubseteq, num, nk)$, where $(P, \leq, \sqsubseteq)$ is a nonempty, purified, labeled ordered tree, $num : P \rightarrow \{*, +, \texttt{?}, \texttt{1}\}$ is a mapping such that nk maps the bottom of P to $\texttt{1}$, and $nk : P \rightarrow \{\texttt{s}, \texttt{c}\}$ is a mapping.*

The mapping *num* formalizes the cardinality operator of XML grammars, and the mapping *nk* (nodekind) indicates whether we have to make a choice among the siblings or not. Note that we consider only non-ambigous XML-grammars, that is, the corresponding XML-trees are purified.

5 Mapping Documents to Grammars

As described in section 3, in each state the user has selected an element in the XML grammar and an element in the XML document. Changing the selected element in the XML grammar affects the selected element in the XML document, and vice versa, changing the selected element in the XML document affects the selected element in the XML grammar. To describe the interdependence between the elements in the XML grammar and in the XML document formally, in this and the following section, we define mappings between the grammar tree and the document tree.

In this section, we start with the easier direction, which is mapping nodes in the document tree to nodes in the grammar tree. In order to do so, we have to define when a document tree conforms to a given grammar as well. This is done by the following definition.

Definition 5 ((Partial) Valid Docs). *Let $\underline{G} := (G, \leq^G, \sqsubseteq^G, l^G, num^G, nk^G)$ be a grammar-tree, let $\underline{D} := (D, \leq^D, \sqsubseteq^D, l^D)$ be a document-tree. We say that $\underline{D}$* IS A PARTIAL VALID DOCUMENT TREE W.R.T. *$\underline{G}$, iff*

1. *For each element $d \in D$, there exists exactly[2] one $g \in G$ with $d \cong g$. This element will be denoted $\psi(d)$.*

[2] As $\underline{G}$ is purified, we could replace 'exactly' by 'at least'.

2. *For all $d_1, d_2 \in D$ we have*

$$d_1 \sqsubseteq^D d_2 \quad \Longrightarrow \quad \psi(d_1) \sqsubseteq^G \psi(d_2)$$

That is, ψ respects the left-right-order $\sqsubseteq$.

3. *There does not exist a $d \in D$ with distinct children $d_1, d_2 \in D$ which satisfy $nk(d) = \mathtt{c}$ and $d_1 \ncong d_2$ (i.e., ψ respect choices).*

Let $d \in D$. We say that d is COMPLETED, *iff we have:*

1. *If $nk(\psi(d)) = \mathtt{s}$, then:*
   (a) *For each child g' of $\psi(d)$ with $num(g') = +$, there exists at least one child d' of d with $\psi(d') = g'$.*
   (b) *For each child g' of $\psi(d)$ with $num(g') = \mathtt{?}$, there exists at most one child d' of d with $\psi(d') = g'$.*
   (c) *For each child g' of $\psi(d)$ with $num(g') = \mathtt{1}$, there exists exactly one child d' of d with $\psi(d') = g'$.*
2. *If $nk(\psi(d)) = \mathtt{c}$, and if there is a child g' of g with $num(g') \in \{+, \mathtt{1}\}$, then d has at least one child.*

We say that a partial valid document tree D IS A COMPLETE VALID DOCUMENT TREE W.R.T. *G iff all $d \in D$ are completed.*

The mapping ψ is the mapping from grammar trees to document trees, which is used to describe the navigation formally. A first nice property in terms of *stable navigation* is that ψ respects $\leq$, and on purified downsets, it is even an order-embedding.

Lemma 2 (Properties of ψ). *Let $\underline{G} := (G, \leq^G, \sqsubseteq^G, l^G)$ be a grammar tree and $\underline{D} := (D, \leq^D, \sqsubseteq^D, l^D)$ be partial valid document tree w.r.t. $\underline{G}$. Then ψ respects $\leq^D$, i.e., for all $d_1, d_2 \in D$ we have $d_1 \leq^D d_2 \Rightarrow \psi(d_1) \leq^G \psi(d_2)$, and $\psi[D]$ is a downset of G (with $\psi[D] := \{\psi(d) \mid d \in D\}$).*

If $B \subseteq D$ is a purified downset, then ψ restricted to B is even an order embedding, i.e., for all $b_1, b_2 \in B$ we have $b_1 \leq^D b_2 \Leftrightarrow \psi(b_1) \leq^G \psi(b_2)$.

Proof: Let $d_1, d_2 \in D$ with $d_1 \leq^D d_2$. Then there is a $g' \in G$ with $g' \leq^G \psi(d_2)$ and $g' \cong \psi(d_1)$, thus $g' = \psi(d_1)$ due to the definition of ψ. Hence we get $\psi(d_1) \leq^G \psi(d_2)$. It can be similarly argued that $\psi[D]$ is a downset.

Now let $B \subseteq D$ be a purified downset and $b_1, b_2 \in B$ with $\psi(b_1) \leq^G \psi(b_2)$. Then there exists $d \in D$ with $d \leq^D b_2$ and $d \cong \psi(b_1)$, hence $d \cong b_1$. As B is a downset, we have $d \in B$, and as B does not contain congruent distinct elements, we have $d = b_1$. Hence we obtain $b_1 \leq^D b_2$. □

6 Mapping Grammars to Documents

In the last section, we defined the mapping ψ from document trees to grammar trees. The definition of ψ is straight forward. In this section, we consider the other direction, that is, mapping nodes in the grammar tree to nodes in the document tree.

As the document tree will usually be only partial valid, but not completed, we cannot expect that we find for each grammar node a congruent document node. On the other hand, it might happen that for a given node in the grammar tree, we can have a number of different congruent nodes in the document tree.

The basic idea of our approach is that we have a selected node, called RANGE CURSOR, in the document tree. This range cursor will be used to identify a set of nodes in the document tree which can be considered to be a *range of interest* fixed by the range cursor. It is the introduction of this range cursor into our formalism that makes stable navigation possible. The mapping from the grammar tree to the document tree will map each grammar node to a document node in this range of interest. We will use the letter φ to denote the mapping from grammar trees to document trees. As φ depends on on a range cursor r, it will be indexed with r (i.e., we write φ_r).

Assume we have an XML-document, and a range cursor, which is a node r in the corresponding document tree. This node corresponds to a path. To which other paths can we move? We can move to subpaths of the given path $\downarrow r$, but not to paths which are congruent to a subpath of the given path $\downarrow r$ without already being a subpath. To rephrase it: If we have a path which deviates from the given path, then the node where it deviates must have a different label. This idea will be fixed by a TARGET AREA $TA(r)$. Now, the target area may still contain congruent, but different nodes. If we have such a choice, we choose the left-most path. The left-most elements in the target area will be called the RANGE of r. An example of target areas and ranges is given in Fig. 7. We first have to fix these notions formally.

Definition 6 (Target Area, Range). *Let $\underline{D} := (D, \leq, \sqsubseteq, l)$ be a labeled, ordered tree, let $r \in D$ be an element. Let*

$$TA(r) := \{d \in D \mid \forall q \leq d \, \forall c \leq r : q \cong c \Rightarrow q = c\} \quad .$$

Now let $Rg(r)$ be the set of all minimal elements of $TA(r)$ with respect to $\sqsubseteq_l$. We call $Rg(r)$ the RANGE OF *r.*

Due to the informal description, it should be clear that the target area of a given document cursor is a down-set. This carries over to the range. Moreover, the range is purified. These claims are proven in the following lemma.

Lemma 3 (Properties of Range). *Let $\underline{D} := (D, \leq, \sqsubseteq, l)$ be a labeled, ordered tree, let $r \in D$. Then $Rg(r)$ is a downset w.r.t. $\leq$, and it is purified.*

Proof: We first prove that $Rg(r)$ is a downset. Let $d \in Rg(r)$ and $e < d$. We obviously have $Rg(r) \subseteq TA(r)$, thus $d \in TA(r)$, and $TA(r)$ is a downset w.r.t. $\leq$, thus $e \in TA(r)$ as well. Assume that e is not minimal w.r.t. $\sqsubseteq_l$, i.e., there exists $f \in TA(r)$ with $f \sqsubset_l e$. Then, as $\sqsubseteq_l$ respects $\leq$ due to Lem. 1, we have $f \sqsubset_l d$ as well, in contradiction to the minimality of d w.r.t. $\sqsubseteq_l$ in $TA(r)$. So we obtain $e \in Rg(r)$.

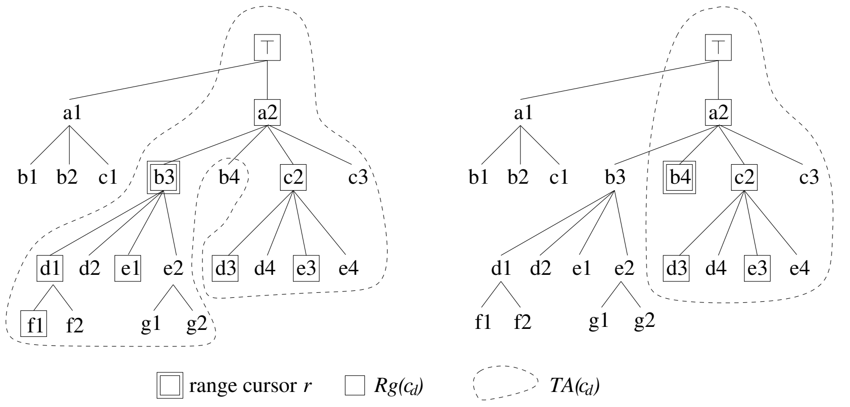

Fig. 7. Target areas and ranges

Assume $Rg(r)$ is not purified. So there are $d, e \in Rg(r)$ with $d \neq e$ and $d \cong e$. Then d, e are incomparable w.r.t. $\leq$. Let $q := d \wedge e$, thus $q < d, e$. From $q^d \leq d$, $q^e \leq e$, and $d \cong e$ we conclude $q^d \cong q^e$. Thus q^d, q^e are comparable w.r.t. $\sqsubseteq_l$, thus d, e are comparable w.r.t. $\sqsubseteq_l$. So d or e is not minimal w.r.t. $\sqsubseteq_l$, which is a contradiction. $\square$

Now we can define the desired mapping φ_r and prove that it is, similar to ψ, order-preserving, and that it to some extent the inverse mapping to ψ.

Lemma 4 (Mapping Induced by a Document Cursor). *Let a grammar-tree $\underline{G} := (G, \leq^G, \sqsubseteq^G, l^G)$ be given and let $\underline{D} := (D, \leq^D, \sqsubseteq^D, l^D)$ be a partial valid document tree w.r.t. $\underline{G}$, let r be a document cursor. Let $\varphi_r : G \to D$ be defined as follows:*

$$\varphi_r(g) = \max_{\leq^D}\{d \in Rg(r) \mid \psi(d) \leq^G g\}$$

Then φ_r is well-defined and order-preserving (w.r.t. $\leq$). For each $b \in Rg(r)$, we have $\varphi_r(\psi(b)) = b$. Finally, we have $\psi(\varphi_r(g)) \leq^G g$ for each $g \in G$.

Proof: Let $\Phi(g) := \{d \in Rg(r) \mid (\psi(d) \leq^G g\}$ for each $g \in G$. The root of D is an element of $Rg(r)$, thus it is an element of $\Phi(g)$, so $\Phi(g) \neq \emptyset$. Let $d_1, d_2 \in \Phi(g)$. Then as $\psi(d_1) \leq^G g$ and $\psi(d_2) \leq^G g$, $\psi(d_1)$ and $\psi(d_2)$ are comparable w.r.t. $\leq^G$. As $Rg(r)$ is a purified downset due to Lem. 3, from Lem. 2 we conclude that d_1 and d_2 are comparable w.r.t. $\leq^D$ as well. So $\Phi(g)$ is a nonempty total order, thus it has a maximal element, i.e., φ_r is well-defined. Moreover, for $g_1 \leq^G g_2$, we have $\Phi(g_1) \subseteq \Phi(g_2)$, thus $\varphi_r(g_1) = \max_{\leq^D}\Phi(g_1) \leq^D \max_{\leq^D}\Phi(g_2) = \varphi_r(g_2)$, so φ_r is order-preserving.

We have $\Phi(\psi(b)) = \{d \in Rg(r) \mid (\psi(d) \leq^G \psi(b)\} = \{d \in Rg(r) \mid d \leq^D b\}$ for each $b \in Rg(r)$ due to Lem. 2, which yields $\varphi_r(\psi(b)) = b$.

Finally, $\psi(\varphi_r(g)) \leq^G g$ follows immediately from the definition of φ_r. $\square$

For illustrating φ_r, we come back to the second example of Fig. 7. The corresponding mapping φ_r is depicted in Fig. 8.

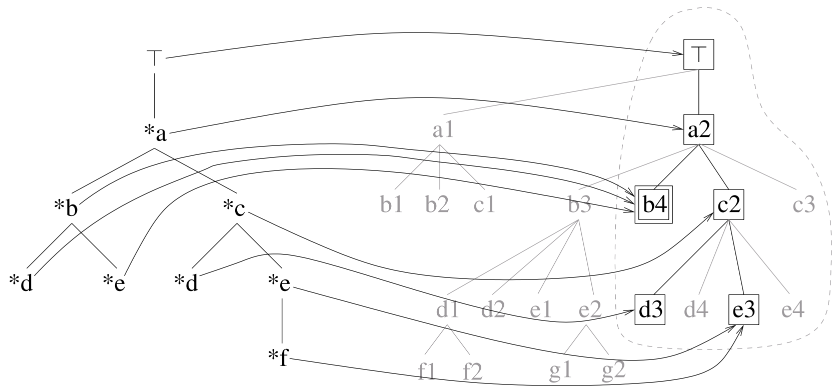

Fig. 8. Example for $\boldsymbol{\varphi_r}$

7 Stable Navigation Approaches

Our formal description of the grammar to document mapping in the previous section introduced a range cursor that was distinct from the target area. That is, the document may contain two cursors from a users perspective. A cursor that captures where the user last clicked and a cursor that shows where the target of the grammar cursor is. Figure 6 introduced a visual syntax for these cursors. In this section we describe how this separation provides the foundation for several stable navigation designs.

When using Xeena for Schema, in each state an element in the XML grammar, the *grammar cursor*, and an element in the XML document, the *document cursor*, is selected. Changing the selected element in the grammar view induces a new selected element in the document, and vice versa, changing the selected element in the document view induces a new selected element in the grammar. This changes have been formally captured by the mappings ψ and φ_r. The mapping φ_r relies on a range cursor r. For our formalization of navigation, let a STATE be a triple (c_g, c_d, r) with a GRAMMAR CURSOR $c_g \in G$, a DOCUMENT CURSOR $c_d \in D$, and an RANGE CURSOR $r \in D$. Now the navigation can be captured as follows:

1. If a user clicks in a state (c_g, c_d, r) on a grammar node g, then let $(g, \varphi_r(g), r')$ be the new state.
2. If a user clicks in a state (c_g, c_d, r) on a document node d, then let $(\psi(d), d, r')$ be the new state.

We leave it for a moment open how r' is obtained.

We have shown that both ψ and φ_r are order preserving. That is, for example, if a user clicks in a given state on a grammar element below the grammar cursor, the document cursor changes to a new document cursor below the old one as well. So, the navigation is to some extent well behaved.

Now assume a user selects a grammar cursor g_1, then selecting a grammar cursor $g_2 \geq g_1$, and then he selects g_1 again. This yields a series of states (g_1, c_1, r_1), (g_2, c_2, r_2), (g_1, c_3, r_3). The question is: Does at the end of this 'hopping up and back' procedure on the grammar side, we are back on the same document cursor as well? If this is true, i.e. if we necessarily have $c_3 = c_1$, we call our navigation UPWARD STABLE W.R.T. THE GRAMMAR. The notions of DOWNWARD STABLE W.R.T. THE GRAMMAR (where we then impose the condition $g_2 \leq g_1$) and UPWARD/DOWNWARD STABLE W.R.T. THE DOCUMENT (where we navigate on the document instead of the grammar) are defined analogously. Obviously, it is desirable that the navigation is in all these respects stable.

Let us first note that, as ψ does not depend on the range cursor, our navigation is always upward and downward stable w.r.t. the document. So we have to discuss only stable grammar navigation.

In the Xeena for Schema implementation, the range cursor is simply set to be the document cursor (i.e., we have only states (c_g, c_d, c_d). In this case, the navigation is upward stable w.r.t. the grammar. This can easily be seen, as we have $c_2 \geq c_1$, thus $c_1 \in Rg(c_2)$ (see Lemma 3). But, as discussed in section 3.3, this navigation is not downward stable w.r.t. the grammar.

Automatically changing the range cursor whenever the grammar cursor is changed is not appropriate. Certainly not when a user changes from a given grammar cursor to a new grammar cursor below, the range cursor should remain. Thus a different approach for changing the range cursor is needed. There are basically two different approaches:

1. The range cursor remains if the a user changes from a given grammar cursor to a new comparable grammar cursor (below or above). If the user changes to a new, incomparable grammar cursor, the range cursor is set to the new document cursor.
2. The range cursor is never changed when the user changes the grammar cursor. Instead, the user has to explicitely set the range cursor.

These approaches yield stable navigation. It is unlikely that a pure theoretical investigation can determine which of the given behaviours is best suited for users. The different approaches need to be implemented then evaluated in future work. But regardless of the choice, as they all yield stable navigation they are a significant improvement on the Xeena for Schema implementation.

8 Conclusion

We have presented a formal description that extends the grammar based navigation and editing of XML document trees implemented in Xeena for Schema. Our formal description abstracts beyond that implementation by dealing with generic XML grammar and document trees so that the design can be applied more broadly.

A key part of our formalism was the inclusion of a range cursor that captures user selection in the document view. This extra state allowed our revised design to support stable navigation, unlike Xeena for Schema which does not. This process of finding limitations and solving them is a major benefit of formalising designs in general, but is particularly so in the case of designing visual notations and languages and the interactive systems that use them.

Acknowledgment

The second author led the development of the Xeena for Schema XML editor while he was with the Knowledge Management group at IBM's Haifa Research Laboratory in Israel.

References

[1] Abiteboul S.: Semistructured data: from practice to theory. In Proc. of the IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, (2001) 379–386

[2] Amaya editor/browser. http://www.w3.org/Amaya/Overview.html (accessed 2007)

[3] Balance R.A., Graham S.L., and Van De Vanter M.L.: The pan language-based editing system. ACM Trans. on Software Engineering and Methodology, Vol. 1(1) (1992) 95–127

[4] Chidlovskii B.: A structural Advisor for the XML document authoring. In Proc. ACM Document Engineering, (2003)

[5] Cowan D.D., Mackie E.W., Pianosi G.M., Smit G.V.: Rita an editor and user interface for manipulating structured documents. Electronic Publishing, John Wiley, Vol. 4(3) (1991) 125–150

[6] Furuta R., Quint V., Andre J.: Interactively editing structured documents. Electronic Publishing, Vol. 1(1) (1988) 19–44

[7] Kuo Y.S., Shih N.C., Tseng L., Hu H.C.: Generating form-based user interfaces for XML vocabularies In Proc. ACM Document Engineering, (2005) 58–60.

[8] Munroe K.D., Papakonstantinou Y.: BBQ: A visual interface for integrated browsing and querying of XML. In Proc. of IFIP Visual Database Systems, Kluwer, (2000) 277–296

[9] Papakonstantinou Y., Vianu V.: DTD inference for views of xml data. In Proc. of ACM PODS, (2000) 35–46

[10] Quint V., Roisin C., Vatton I.: A structured Authoring Environment for the World-Wide Web. In Proc. of the World Wide Web Conference, (1995)

[11] Quint V., Vatton I.: Techniques for authoring complex XML documents. In Proc. ACM Document Engineering, (2004) 115–123

[12] Sifer M., Peres Y., and Maarek Y.: Xeena for schema: creating xml data with an interactive editor. In Proc. of DNIS, LNCS volume 2544, Springer, (2002) 133–146

[13] Sifer M., Peres Y., and Maarek Y.: Browsing and editing xml schema documents with an interactive editor. In Proc. of DNIS, LNCS volume 2822, Springer, (2003) 97–111

[14] Teitelbaum T., Reps T.: The Cornell Program Synthesizer: A syntax-directed programming environment. Communications of the ACM, vol. 24 (9), (1981) 563–573

[15] Topologi markup editor. www.topologi.com/products/tpro (accessed 2007).

[16] XML The extensible markup language 1.0 (third edition), W3C recommendation 2004. www.w3.org/TR/2004/REC-xml-20040204

[17] Xeena at alphaworks. www.alphaworks.ibm.com/tech/xeena (accessed 2007)

[18] Xforms 1.0 (second edition), W3C recommendation 2006. www.w3.org/TR/xforms

[19] XHTML 1.0 The extensible hypertext markup language (second edition), W3C recommendation 26 January 2000, revised 1 August 2002. www.w3.org/TR/xhtml1

[20] XMetal Author. na.justsystems.com/content.php?page=xmetal (accessed 2007).

[21] Xml Spy. www.altova.com/manual2007/XMLSpy/SpyEnterprise/ (accessed 2007).

[22] Zelkowits M.: A small contribution to editing with a syntax directed editor. In Proc. of the ACM Software Engineering Symposium on Practical Software Development Environments, (1984)