Edits in Xylia
Validity Preserving Editing of XML Documents

Pouria Shaker, Theodore S. Norvell, and Dennis K. Peters
Faculty of Engineering and Applied Science,
Memorial University of Newfoundland,
St. John's, NFLD, Canada

Abstract

This paper presents an overview of the Xylia Toolkit's approach to preserving the validity of XML documents subjected to edit actions such as insertions, replacements, and deletions. The Xylia Toolkit is a library of classes that provides the Java programmer with the means necessary to construct a WYSIWYG (What You See Is What You Get) XML editor. In addition, a default XML editor (with essential features) created by the toolkit is included in the package, which can be used as framework for creating an XML editor tailored to the user's need. An XML document is valid if it complies with its associated Document Type Definition (DTD). A DTD imposes structural constraints on a document by defining the set of element types for the document and the allowed parent child relations between them. Validating XML parsers can check a document against its DTD, but the main challenge is to maintain the validity of a document while edits such as insertions, replacements, and deletes are taking place. Xylia's strategy for preserving validity is to disallow operations that would invalidate the document rather than to attempt repairing the damage made.

1. About Xylia

The Xylia Toolkit provides the Java programmer with the means necessary to construct a WYSIWYG (an acronym for "what you see is what you get") XML (Extensible Markup Language) editor. In addition, a default XML editor (with essential features) created by the toolkit is included in the package; it can be used as framework for creating an XML editor tailored to the user's need. A WYSIWYG (pronounced "wiz-ee-wig") editor is one that allows the developer to see the end result of a document while creating it through an easy to use GUI (Graphical User Interface). Examples of currently available HTML WYSIWYG editors are Microsoft's FrontPage, Adobe's PageMill, and Adobe's GoLive.
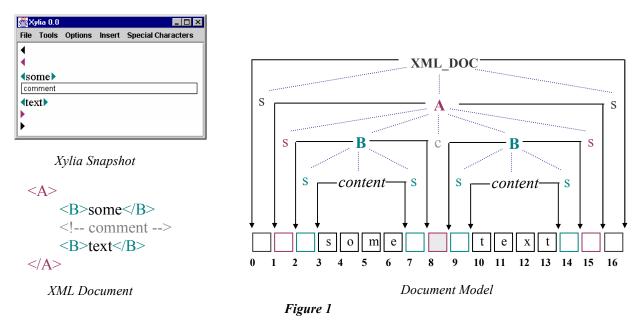
2. About XML

XML [1] is a standard developed by the World Wide Web Consortium (W3C) [2] used to design customized markup languages. A markup language describes the way the content of a document should be interpreted. Unlike HTML, which is a pure markup language, XML does not offer a predefined set of elements and rules of structural validity; rather it defines an infinite set of related languages, and a meta-language for expressing the set of elements and the validity rules of a particular language. The similarity between XML languages allows a single software tool, such as a parser of an editor, to be designed that can handle any XML language, including those not yet conceived of.

3. Modelling XML Documents

Xylia is based on the Swing library of the Java language [3]. The representation of XML documents in Xylia is partly imposed by need to interact with existing parts of the Swing library, and partly by the nature of XML and the needs of the Xylia editor kit.

In Xylia, XML documents are modelled by two data structures: a sequence of characters holding the textual content of the document, and an element tree, consisting of a finite set of nodes satisfying the usual tree properties, extended upon this sequence to give the document structure (see *Figure 1*)

Xylia Snapshot

```
<A>
    <B>some</B>
    <!-- comment -->
    <B>text</B>
</A>
```

XML Document

Document Model

Figure 1

Nodes in the element tree are either branches or runs; runs being the nodes with no children (tree leaves). Each branch node, except the root, represents an XML element. Run nodes are either *content* nodes or *dummy* nodes. Content nodes cover the portion of the character sequence containing textual data. Dummy nodes are of three types: start sentinels, end sentinels (both shown as 's' in *Figure 1*), and comment dummies (shown as 'c' in *Figure 1*). Each branch node has a start sentinel as its first child and an end sentinel as its last child. Comment dummies, as the name suggests, represent comments in the XML document. Dummy nodes cover a single space (' ') character in the character sequence. Branch nodes cover the union of the characters covered by their children. There is one branch element, XML_DOC, which covers the entire sequence. Empty elements such as <D></D> (or <D/>) are considered branch elements and have exactly 2 children: a start sentinel and an end sentinel.

This tree model of the document is manipulated by editing actions and is rendered onto the display by a system of *view* objects. The mapping of tree nodes to view objects is controlled by a Cascading Style Sheet **[2]**. While interesting, the issue of displaying the Xylia's trees will not be further considered in this paper.

4. Well-formedness vs. Validity

The *Well-formedness* constraint, as specified by W3C, is mainly concerned with the syntactical correctness of an XML document, such as the requirement that start and end tags are consistently nested. A complete description of the constraint can be found under the XML 1.0 recommendation available for viewing at the W3C website. XML parsers have been programmed to parse documents adhering completely to this constraint and will fail to complete their task when confronted with an ill-formed document. The document model discussed in the previous section is a result of the parsing of a well-formed document. The characteristics of the data structures described, therefore, serve as well-formedness invariants, some of which are imposed by Swing, while others are specific to Xylia (e.g. the existence of XML_DOC as a default root, the presence of dummy elements, etc.).

An XML document is *valid* if it complies with the document type definition (DTD) associated with the document. A DTD imposes structural constraints on a document by defining the set of element types for the document and the allowed parent-child relations between them. The following is a sample DTD element declaration:

```
<!ELEMENT A ((B, C) | C | D*)>
```

This simply states that elements can have type A and that an element of type A could have the following set of sequences of types for its children:

{BC, C, ε , D, DD, DDD , …}

The expression `((B, C) | C | D*)` is termed a regular expression. A regular expression is a way of expressing a certain pattern, in our case, a child sequence pattern. In the example given, '|' means 'or', ',' means 'juxtaposition', and '*' means 'zero or more instances of'. A detailed treatment of regular expressions is outside the scope of this paper. The set of type sequences associated with a regular expression r is called r's language and is denoted L(r). The occurrence of #PCDATA in a regular expression indicates that *parsed content data* is allowed as a child; "parsed content data" simply means characters, and corresponds to the content nodes in our tree structure.

 Xylia uses a validating XML parser that is capable of checking the document against its associated DTD and of reporting any discrepancies. However the more challenging task is to maintain the validity of a document while edits such as insertions, replacements, and deletes are taking place. In other words, all operations modifying the document model should ensure that the sequence of children of each branch element (excluding all dummies) remains compliant to its DTD declaration.

5. Insertion Sequences

Assume that the regular expressions associated with element types A, B, C, and D are as follows:

A :: B C | C | D*   L(A) = {BC, C, ε , D, DD, DDD , …}
B :: C (A C)* | D   L(B) = {D, C, CAC, CACAC , … }
C :: #PCDATA   L(C) = {ε , #PCDATA, #PCDATA#PCDATA, …}
D :: #PCDATA   L(D) = {ε , #PCDATA, #PCDATA#PCDATA, …}

Suppose that in the course of editing, the user has made the following selection in the document:

<B> <C></C> <A></A> <C></C> </B>

Deleting the selection would violate the DTD since CC is not a member of B's L(r). Inserting character data would also violate the DTD since B does not accept #PCDATA. However, the following set of replacements would be valid:

{A, ACA, ACACA, …}

Now consider the following example:

<A> <C>Hello World</C> </A>

Deletion would be valid since ε is a member of A's L(r). Other replacement options would include:
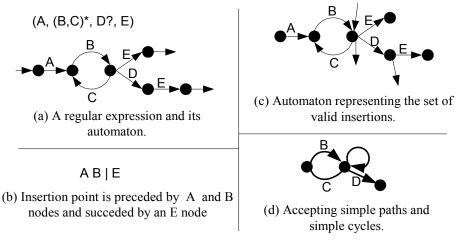
{BC, C, ε, D, DD, DDD, …}

The same set of type sequences would be applicable to the following insertion request:

<A>|</A>

It can be seen that whether the desired edit is insertion, replacement, or deletion, a set of valid insertion sequences must be derived based on the selection made and the affected element's content model. This set would then determine whether the requested operation should be allowed. So Xylia's strategy for preserving validity is to disallow operations that would invalidate the document rather than to attempt repairing the damage made.

When the user wishes to insert at a given position (or replace a given selection of children) under a given parent node, a finite set of insertion sequences is computed and offered to the user on a menu. First the regular expression description of the content model is translated into a deterministic finite state automaton (see *Figure 2 (a)*). By adjusting the start state according to the sequence of children of prior to the insertion point (or selection) and the set of final states according to the sequence of children after the insertion point (or selection), we can obtain a second automaton that represents the set of valid insertion sequences (see *Figure 2 (b)* and *(c)*). In the figure, this set is {ε, D, CB, CBD, CBCB, CBCBD, ...}. As this is an infinite set, it is not suitable to be offered to the user on a menu. A *simple path* in a graph contains no repeated nodes; a *simple cycle* is a path that contains no repeated nodes except that the first and last nodes are the same. We can identify the set of accepting paths in this automaton that are either simple paths or simple cycles (see *Figure 2 (d)*). The set of sequences of labels along these paths constitutes the set insertion sequences offered to the user. In the figure, this is {ε, D, CB}.
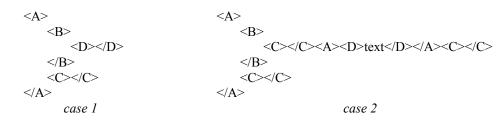
It is important to note that any valid document can be transformed to any other valid document by a suitable combination of the following user actions: pick an insertion point, select a range, insert an offered insertion sequence, and insert a character. Such a set of user actions is termed a *complete set*. For example, in *Figure 2*, if the user really wants to insert CBCBD, they can do so by first inserting CB, moving the cursor to after the second B and inserting CBD. In general any accepting path through the automaton can be decomposed into simple paths and simple cycles which will be offered given appropriate cursor placement.

(A, (B,C)*, D?, E)

(a) A regular expression and its automaton.

A B | E

(b) Insertion point is preceded by A and B nodes and succeded by an E node

(c) Automaton representing the set of valid insertions.

(d) Accepting simple paths and simple cycles.

Figure 2

6. From Sequences to Trees

Suppose that the user chooses to insert the BC insertion sequence after being offered a choice from the set {BC, C, D} (the empty insertion sequence request ε is really a request for deletion and not insertion) for the <A>|</A> insertion request example. Possible insertions would include:

```
    <A>                        <A>
        <B>                        <B>
            <D></D>                    <C></C><A><D>text</D></A><C></C>
        </B>                       </B>
        <C></C>                    <C></C>
    </A>                       </A>
            case 1                             case 2
```

and infinitely many more. For each element in the chosen insertion sequence (in our case BC) a default insertion tree must be selected based on the element type's associated L(r). Recall A, B, C, and D's associated L(r)s:

A :: B C | C | D*   L(A) = { BC, C, ε , D, DD, DDD , … }
B :: C (A C)* | D   L(B) = {D, C, CAC, CACAC , … }
C :: #PCDATA   L(C) = { ε , #PCDATA, #PCDATA#PCDATA, … }
D :: #PCDATA   L(D) = { ε , #PCDATA, #PCDATA#PCDATA, … }

In case 1, at the first level, D was chosen for B and the empty string was chosen for C. At the second level, the empty string was chosen for D.

In case 2, at the first level, CAC was chosen for B and the empty string was chosen for C. At the second level, D was chosen for A and the empty string was chosen for C. Finally at the third level the string "text" was chosen for D.

It is not hard to see that an infinite number of such cases could be listed given the element types' L(r)s.
In Xylia, we identify a tree of minimal height for each element type. This can be done by starting with elements that allow empty content sequences and labelling them as level 0. Next we identify element types that can have content sequences consisting only of level 0 types, and labelling them as level 1. In general, a level $i+1$ type can have a content sequence that contains only types of level i and below and does not have a content sequence that contains only types of level less than i. Only the finite set of content sequences discussed in Section 5 need be considered. For each level i type we can construct a tree of height $i+1$.

There is a restrictions on regular expressions allowed in a DTD implying that #PCDATA can only be a child of an element with level 0 type. Thus the minimal height trees never contain character data.

This algorithm ensures that the trees inserted for the sequence BC are as shown in case 1.

XML elements are associated with one or more attributes. Some attributes are marked as required in the DTD. When a tree is inserted that requires an attribute, the user is prompted to supply the value.

7. A Few Notes on Implementation

Xylia editors do not allow the selection of a single sentinel element; if one is selected, the selection automatically extends to cover its matching sentinel. This is important because it prevents the deletion of a single sentinel element in the course of various edits where selections are involved. Such a deletion would violate the Xylia specific well-formedness invariant of every branch element having a sentinel element as its first and last child. This however does not completely rule out the possibility of single sentinels being deleted, as it is still possible to do so using the delete and backspace keys. Xylia handles *selection deletes* and *backspace and delete-key deletes* differently:

In the case of selection deletes the following algorithm is used:
- If deleting the selection does not violate the DTD, perform delete.
- Otherwise, do nothing.

Backspace and delete-key deletes of sentinels are slightly more involved. The following rules are tried in turn until one is found that will not make the document invalid:
- Try deleting the element parenting the sentinel and promoting its content to its parent.
- If the dummy being deleted is a start sentinel, try deleting the element parenting the sentinel and moving its content to the end of that of its left sibling.
- If the sentinel being deleted is an end dummy, try deleting the element parenting the sentinel and moving its content to the beginning of that of its right sibling.
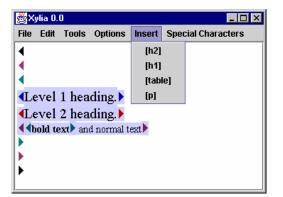- Try deleting the sentiniel's parent and all its content.
- Do nothing.

The Xylia toolkit provides the means necessary to create an *insert menu* in the editor's GUI. The insert menu automatically updates itself in response to changes in the cursor position or the current selection (see

Figure 3). The insert menu items represent the set of insertion sequences described in Section 5. Clicking on a menu-item replaces the selection with (or inserts at the cursor position) the sequence of default trees corresponding to the insertion sequence in question, as discussed in Section 6.

Cut, copy, and paste operations have been recently implemented such that validity is preserved.

```
<html>
    <body>
        <h1>Level 1 heading.</h1>
        <h2>Level 2 heading.</h2>
        <p><b>bold text</b> and normal text</p>
    </body>
</html>
```

Element declaration for body:

<!ELEMENT body (h1 | h2 | p | table)*>

Figure 3

8. Future Considerations

Validity-preserving algorithms for insertion, replacement, and deletion have been implemented. Editing actions that need to be implemented in the future include *distributed engulf* action. An XHTML (XML implementation of HTML) example will make the nature of the distributed engulf action clear. Consider the following selection:

```
<html>
    <body>
        <p>A paragraph</p>
        <table><tr><td>A table entry</td></tr></table>
    </body>
</html>
```

A distributed engulf of the tag would result in the following:

```
<html>
    <body>
        <p><em>A paragraph</em></P>
        <table><tr><td><em>A table entry</em></td></tr></table>
    </body>
</html>
```

Refrences:

[1]   World Wide Web Consortium (W3C), "Extensible Markup Language (XML)",
        [Online document], Available HTTP: http://www.w3.org/XML/
[2]   World Wide Web Consortium (W3C), **"World Wide Web Consortium"**, [Online document],
        Available HTTP: http://www.w3.org/
[3]   Sun Microsystems, Inc. **"Java^{TM} Foundation Classes (JFC)"**, [Online document], Available HTTP:
        http://java.sun.com/products/jfc/