# Handling Syntactic Constraints in a DTD-Compliant XML Editor

Y. S. Kuo, Jaspher Wang, and N. C. Shih
Academia Sinica, Taiwan
{yskuo | jaspher | ncshi}@iis.sinica.edu.tw

## ABSTRACT
By exploiting the theories of automata and graphs, we propose algorithms and a process for editing valid XML documents [4][5]. The editing process avoids syntactic violations altogether, thus freeing the user from any syntactic concerns. Based on the proposed algorithms and process, we build an XML editor with forms as its user interface.

## Categories and Subject Descriptors
H.5.2 [**Information Interfaces and Presentation**]: User Interfaces –*Theory and methods, Interaction styles, Graphical user interfaces (GUI).*

## General Terms
Algorithms, Design

## Keywords
XML Editor, Regular Expression, Automata Theory

## 1. INTRODUCTION
Current XML editors [6] provide guidance or hints to assist the user to construct valid XML documents. However, the guidance or hints are typically too loose to guarantee the validity of the resulting document. Frequently the user still needs to correct the syntactic violations reported by a validity checker. This requires the user's knowledge about the syntax of the document.

By exploiting the theories of automata and graphs [1][3], we propose algorithms and a process for editing valid XML documents. With this approach, an XML editor can provide accurate guidance and hints to the user, thus avoiding syntactic violations altogether. Consequently, the user requires no knowledge about XML and DTD to edit valid XML documents. To demonstrate the effectiveness of the algorithms and editing process, we build an XML editor with forms as its user interface.

## 2. GLUSHKOV AUTOMATA [1]
A regular expression E over a finite alphabet of symbols $\Sigma = \{x_1, x_2, \ldots, x_n\}$ is simple if each symbol $x_i$ appears in E only once. The language L specified by E can be recognized by a deterministic finite automaton (DFA) G, known as the Glushkov automaton for E, defined as follows:

(1) Every symbol of $\Sigma$ is a state of G. G has two additional states s and f as its start and final states, respectively. (If L contains the empty string, s is also a final state.)


DocEng'03, November 20–22, 2003, Grenoble, France. Copyright 2003 ACM 1-58113-724-9/03/0011…$5.00.

(2) The transition function $\delta(x_i, x_j) = x_j$ for any $x_i \in \Sigma$ and $x_j \in$ follow($x_i$), i.e. $x_j$ immediately follows $x_i$ in some string in L. $\delta(s, x_j) = x_j$ for any $x_j \in$ first(E), i.e. $x_j$ is the first symbol in some string in L. $\delta(x_i, \$) = f$ for any $x_i \in$ last(E), i.e. $x_i$ is the last symbol in some string in L, where \$ is a special end symbol appended to every string.

Note that the functions first(E), last(E) and follow($x_i$) can be computed easily by traversing the tree structure of E once. Take the regular expression E = *(a, b, c*, (d | e+))** as an example. The Glushkov automaton G for E is as shown in Figure 1.

Edges in the Glushkov automaton G are of several types: If E1 and E2 are two subexpressions of E in sequence, i.e. (E1, E2) is a subexpression of E, then G contains a sequence edge (u,v) for every u ∈ last(E1) and every v ∈ first(E2). Sequence edges and edges from the start state s and edges to the final state f are referred to as forward edges collectively. If E1* is a subexpression of E, then G contains an iteration edge (u,v) for every u ∈ last(E1) and every v ∈ first(E1). In general, an edge may be a sequence edge as well as an iteration edge. An iteration edge that is not a sequence edge is referred to as a backward edge.

For a subexpression E1 of E, let A(E1) denote the set of symbols in $\Sigma$ that E1 covers. Let reachable(u) denote the set of states in G reachable from state u, and f-reachable(u) the set of states in G reachable from u through forward edges.

Lemma 1. The forward edges in G form no cycles.

Lemma 2. Let E1 be a subexpression of E. For any x ∈ A(E1), there exists some u ∈ first(E1) and v ∈ last(E1) such that x ∈ f-reachable(u) and v ∈ f-reachable(x).

Lemma 3. Let E1* be a subexpression of E. Then for any two symbols u and v in A(E1), we have v ∈ reachable(u) and u ∈ reachable(v).

Consider a regular expression E' over a finite alphabet of symbols $\Sigma' = \{y_1, y_2, \ldots, y_m\}$ in general. One can map E' to a simple regular expression E over an alphabet $\Sigma = \{x_1, x_2, \ldots, x_n\}$ by renaming each occurrence of symbol $y_i \in$ E' as a distinct symbol $x_j \in \Sigma$. Let origin($x_j$) = $y_i$ denote the original symbol $y_i$ that $x_j$ represents. Let G be the DFA constructed above for E. One can construct an automaton G', known as the Glushkov automaton for E', from G to recognize the language L' specified by E'. Treating automata as labeled graphs, G' is constructed from G by replacing all labels $x_j$ on edges in G by origin($x_j$). Different from G, G' is a non-deterministic automaton (NFA) in general since multiple $x_j$'s may have the same origin($x_j$) in $\delta(x_i, \text{origin}(x_j)) = x_j$.

Consider the regular expression E' = *(a, b, a*, (d | e+))**. E' can be mapped to the simple regular expression E = *(a, b, c*, (d | e+))** by renaming the second occurrence of *a* in E' to *c*. We can then construct the Glushkov automaton G' for E' from the Glushkov automaton G for E in Figure 1 by replacing the label *c* on all edges by label *a*.

Notice that the W3C XML specification has imposed the constraint that regular expressions used for defining the content models of element types must be deterministic, i.e. their Glushkov automata are deterministic [4]. On the other hand, in the Glushkov automata G and G', the labels on edges can be determined by the target states of the edges. Thus they can be ignored temporarily in internal computations. G and G' become identical in this case. Labels on edges of G' are significant only when they are presented to the user.

# 3. EDITING PROCESS AND ALGORITHMS

Let E' be a regular expression over the alphabet $\Sigma' = \{y_1, y_2, \ldots, y_m\}$ and E its associated simple regular expression over alphabet $\Sigma = \{x_1, x_2, \ldots, x_n\}$. Also let G and G' be the Glushkov automata for E and E', respectively, as constructed in Section 2. Assume that E' is used as the content model of an element type. Then a valid document corresponds to a path in G from s to f, and vice versa. An XML editor, with the goal of producing valid documents, cannot guarantee always producing a valid document in the process of construction. (Initially the empty document is typically not valid.) Instead, we insist that the working document is always DTD-compliant, which corresponds to a subsequence of some path in G from s to f.

To be formal, a DTD-compliant document corresponds to a sequence $\{z_1, z_2, \ldots, z_p\}$ of states in G where $z_1$=s, $z_p$=f, and $z_{i+1} \in$ reachable($z_i$), $1 \le i \le p-1$. Document editing then consists of insertions to and deletions from such a state sequence. Deletions are of no concerns; the resulting document is DTD-compliant apparently. For insertions, it is sufficient to consider the issue given a pair of states u and v in G where v $\in$ reachable(u).

The DTD-compliant editor can generate required elements automatically in the process of document construction. An element is required if it is present in all valid documents containing the current DTD-compliant document. A required element between u and v then corresponds to a state z through which all paths from u to v pass. Such a state is known as a cut-vertex or articulation point in graph theory [3]. It is well known that one can apply a maximum flow-like algorithm to find the articulation points separating u and v, i.e. to find the required elements between u and v.

The DTD-compliant XML editor can generate not only elements but also element slots for the user to fill in. Suppose $(u,v) \notin H$, where H is the edge set of G. Then every valid document containing the current document must contain at least one element between u and v. The system thus generates an element slot between u and v automatically for the user to fill in. Such element slots are referred to as required element slots. On the other hand, the system may generate an optional element slot between u and v upon the user's request when (u,v) $\in$ H.

When the system generates an element slot between u and v, it also computes a candidate state set $C \subseteq \Sigma$. C is mapped to a candidate element list $C' \subseteq \Sigma'$ as options for the element slot, where $C' = \{y \in \Sigma' \mid y = \text{origin}(x) \text{ for some } x \in C\}$. The system lets the user select a desired element, say y, from C'. The system then maps $y \in C'$ back to a state $x \in C$ for addition to the current document. The candidate state set C contains the "possible" paths connecting u to v. A necessary condition for a state z to be in C is to satisfy z $\in$ reachable(u) and v $\in$ reachable(z). However, this condition is not sufficient, too loose to give appropriate options in general. We thus aim at computing a "minimal" candidate state set that does not involve cycles and unnecessary backward edges.

Lemma 4. Let u and v be states in G for which v $\in$ reachable(u). Assume v $\notin$ f-reachable(u). Then there exists a subexpression E1* of E covering u and v such that w $\in$ f-reachable(u) for some w $\in$ last(E1) and v $\in$ f-reachable(z) for some z $\in$ first(E1). This constructs an acyclic path P connecting u to v.

Algorithm FindCandidateStates1 computes the candidate state set C for a required element slot between states u and v where v $\in$ reachable(u) and $(u,v) \notin H$. C is composed of the intermediate states in the acyclic paths from u to v determined by Lemma 4.

```
Algorithm FindCandidateStates1
IF v ∈ f-reachable(u) THEN
   C= {x ∈ Σ | x ∈ f-reachable(u) and v ∈ f-reachable(x)}
ELSE
   let E1* be the smallest iteration subexpression of E that covers
   both u and v
   C= {x ∈ A(E1) | x ∈ f-reachable(u) or v ∈ f-reachable(x)}
ENDIF
```

Let us illustrate Algorithm FindCandidateStates1 with the Glushkov automaton G in Figure 1. The state pair *(a,e)* satisfies *e* $\in$ f-reachable(*a*). Thus we have C= *{b,c}*. For the state pair *(c,a)*, *a* is not reachable from *c* through forward edges. *c* must reach *a* through *d* or *e*. Thus, we have C= *{d,e}*.

Algorithm FindCandidateStates2 computes a candidate state set C for states u and v where v $\in$ reachable(u) and (u,v) $\in$ H. An optional element slot is inserted between u and v if the result C is not empty. Here (u,v) can be a forward edge, an iteration edge or both. If (u,v) is a forward edge, C is first computed as in FindCandidateStates1. On the other hand, if u is the end or v is the beginning of an iteration or (u,v) is a backward edge, a new iteration can be inserted between u and v by adding its symbols to C.

```
Algorithm FindCandidateStates2
IF (u,v) is a forward edge THEN
   C= {x ∈ Σ | x ∈ f-reachable(u) and v ∈ f-reachable(x)}
   IF u ∈ last(E1*) for some iteration subexpression E1* of E,
      and let E1 be the largest one, THEN
      C1= {x ∈ A(E1) | v ∈ f-reachable(x)}
      C= C ∪ C1
   ENDIF
   IF v ∈ first(E2*) for some iteration subexpression E2* of E,
      and let E2 be the largest one, THEN
      C2= {x ∈ A(E2) | x ∈ f-reachable(u)}
      C= C ∪ C2
   ENDIF
ELSE /* (u,v) is a backward edge          */
   let E3* be the largest iteration subexpression of E
      satisfying u ∈ last(E3) and v ∈ first(E3)
   C= A(E3)
ENDIF
```

Consider the Glushkov automaton G in Figure 1. For the state pair *(a,b)*, we have C= { }, which indicates no optional element slot should be inserted between *a* and *b*. For the state pair *(b,d)*, we have C= *{c}*. For the state pair *(b,c)*, since *c* $\in$ first(*c**), we have C= *{c}*, which allows the user to iterate *c*. For the state pair *(c,c)*, which is a backward edge, we have C= *{c}*. The user can add an iteration *c* between the two iterations. For the state pair *(e,f)*, since *e* $\in$ last(*e**) and *e* $\in$ last(E) while E is the outer iteration, we have C= *{a,b,c,d,e}*. The user may want to add the inner iteration or the outer iteration. The system provides the user with all possibilities.

Theorem 1. Under the editing process, the user always produces a

DTD-compliant document, and can construct any desirable valid document.

Lemma 5. All algorithms presented in this section take linear time, linear in the size of the Glushkov automaton G.

## 4. FORM-BASED USER INTERFACE

Form-based user interfaces are easy to use and well accepted. But their applications to XML editors are still not mature due to the complexity of XML syntax. This motivates our development of a DTD-compliant XML editor with forms as its user interface. To cope with the dynamic structure of XML documents, we build a level-limited tree structure into a form. Thus the user interface is indeed a mixture of forms and tree views.

Figure 2 shows the top-level form when an empty document is created. This form displays two levels of elements: *levelone* has two required child elements *clinical_document_header* and *body*. *clinical_document_header* has 5 required child elements. *body* contains a required element slot as its child for the user to fill in. These required elements and required element slots are generated by the system automatically. The system has a parameter LEVEL_LIMIT that determines the number of levels of elements a form may contain. An element at the bottom level in a form appears as a hyperlink if it may have child elements. One can click it to display a child form that shows its child (and grandchild, etc.) elements. A child form may have child forms again so that child forms may nest indefinitely.

As shown in Figure 2, each form has two buttons for displaying and hiding all optional element slots, respectively. When a form does not contain many elements and attributes, this simple setup may be adequate and desirable. When a form contains many elements and attributes, displaying all optional element slots may clutter the form with slots. Alternatively, one may want to display optional element slots only around a selected element. When one moves the pointer over an element, a small OPTIONAL icon may pop up following the element in the same row as shown in the figure. If the user clicks the icon, the optional element slots around the current element are displayed as shown in Figure 3. Also shown in Figure 3, the user can fill in an element slot by selecting from a menu of candidate elements generated by the system. Upon the user's any selection, the system guarantees the resulting document DTD-compliant.

## 5. RELATED WORK

The current work is most related to Rita, an editor prototype developed in the late 80's for manipulating structured documents [2]. The concept of "DTD-compliant" was introduced in Rita, referred to as "subsequence-incomplete". In Rita, the candidate elements for insertions were computed based on a non-deterministic finite automaton, and by finding multiple shortest paths, which is inadequate in terms of both efficiency and completeness. Required elements and required element slots were not addressed either.

## 6. CONCLUDING REMARKS

XML editors have been an area with more practices than theories. XML editors demand simple user-friendly user interfaces for non-technical users. That cannot be achieved without techniques to handle the XML syntactic constraints. The major contribution of this work is to lay some theoretic foundations for XML editors for handling syntactic constraints, which would potentially render these systems more robust and user-friendly.

## 7. REFERENCES


[1] A. Bruggemann-Klein, "Regular expressions into finite automata", Theoretical Computer Science, 120, 1993.

[2] D. D. Cowan, E. W. Mackie, G. M. Pianosi, and G. de V. Smit, "Rita – an editor and user interface for manipulating structured documents", Electronic Publishing, 4(3), Sept. 1991, pp 125-150.

[3] S. Even, Graph Algorithms, Computer Science Press, Maryland, 1979.

[4] W3C, Extensible Markup Language (XML) 1.0, W3C Rec., Feb. 10, 1998.

[5] W3C, XML Schema Part 1: Structures, W3C Rec., May 2, 2001.

[6] XMLSoftware, http://www.xmlsoftware.com/editors.html


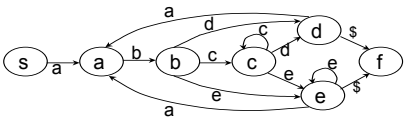


Figure 1. Glushkov automaton G

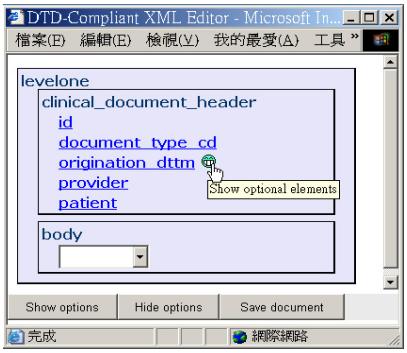


Figure 2. A top-level form


Figure 3. Optional element slots