Avoiding Syntactic Violations in Forms-XML

Y. S. Kuo, N. C. Shih, and Lendle Tseng
Institute of Information Science
Academia Sinica, Taiwan

Jaspher Wang
Framework Technology Group, R&D Department
Trend Micro Inc.

Abstract

So far, user interfaces for XML data (documents) have been typically constructed from scratch for specific XML vocabularies and applications. Forms-XML is a generic interactive component that generates form-based user interfaces for XML vocabularies. It can be considered as an unusual XML editor that avoids, instead of detects, syntactic violations. This paper covers the model and algorithms Forms-XML adopts that always keep a working document compliant with a given XML schema. The approach gains the following benefits: First, by avoiding syntactic violations altogether, Forms-XML can free the user from any syntactic concerns. Second, the system can generate elements and element slots in the process of document construction, which serve as prominent guides to the user to produce valid documents. Third, the simple editing process Forms-XML supports enables the generation of its simple form-based user interface.

Keywords
XML Editors, Regular Expressions, Finite Automata, Syntax-Directed Editing, Graph Algorithms

Address for Correspondence
Dr. Y. S. Kuo
Institute of Information Science
Academia Sinica
Nankang, Taipei, Taiwan
Email: yskuo@iis.sinica.edu.tw

1. Introduction

Extensible markup language (XML) [22] has emerged as a standard for electronic data interchange. Many application domains have formed consortia or coalitions to define domain-specific XML vocabularies for the interchange of data among organizations within the application domains [27]. As XML database management is getting matured [23], XML vocabularies may even become the standard information model for future enterprise application systems.

With the increasing number of emerging XML vocabularies, there is apparently a need to construct user interfaces for XML data (documents) within a given vocabulary effectively. So far, user interfaces for XML data have been typically constructed from scratch for specific XML vocabularies and applications. This approach is labor-intensive and very costly considering that enterprise-level XML vocabularies typically contain hundreds, even thousands, of elements and attributes, and the user interface must take care of the dynamic XML data structures, syntactic constraints, and presentation layout, etc.

In contrast to that for relational data, the current tool support for user interfaces for XML data is far from adequate. There have been a variety of simple interactive components, e.g. grid and navigation controls, that can be used for constructing user interfaces for relational data conveniently while no user interface tools for XML data are widely available. This has motivated our development of an interactive component, Forms-XML, which can generate form-based user interfaces for XML vocabularies.

Forms-XML shares many common functions with XML editors [3][6][20]. Both are used by users to interact with and edit XML documents. They differ in that XML editors are stand-alone applications while Forms-XML, as an interactive component,

is invoked by applications. Forms-XML can be considered as a light-weight XML editor for use by end users who have no knowledge of XML.

As a generic component to build user interfaces for XML vocabularies, Forms-XML takes an XML schema [24][25] as its input, and allows the user to create and update XML data compliant with the input schema. The component must take care of two main issues: handling the syntactic constraints imposed by the schema and producing a presentation layout that is simple to use and customizable to specific XML vocabularies. The current paper only deals with the former issue. We leave the latter and the systems aspects of Forms-XML to a later report [13].

Many XML editors take a detect-and-correct approach to syntactic violations [3][20]. A user is allowed to violate syntactic constraints in the process of editing. To produce a valid document, the user must correct the syntactic violations in the document reported by a validity checker. Unfortunately, this approach requires that the user have some knowledge about the syntax of the document, which is an impractical assumption for end users that Forms-XML is targeted for. On the other hand, one might wish the system to correct the syntactic violations automatically. This turns out to be infeasible either since there are typically multiple ways to correct a syntactic violation, and the system has no way to guess what the user wants.

Quite different from the detect-and-correct approach, Forms-XML takes actions to avoid syntactic violations altogether. We would like to always keep a working document free of syntactic violations. However, what is a syntactic violation? A valid document is free of syntactic violations of course. But the converse is not true intuitively. Starting with an empty document, it may take many "violation-free" element insertions until a working document becomes valid. A working document constructed halfway is considered to be free of syntactic violations if it is a subset of a

valid document. We thus define a DTD-compliant document as a subset of a valid document. (A formal definition will be given in Section 4.)

Forms-XML is indeed a DTD-compliant XML editor, i.e. an XML editor that always keeps a working document DTD-compliant. In order to maintain DTD-compliance, the user is somehow under the control of the system, which provides "accurate" guidance and hints to the user in the process of document construction. A major portion of the current work is to establish a model and algorithms based on automata and graph theories [1][8][10] for computing the appropriate guidance and hints.

As a DTD-compliant XML editor, Forms-XML has the following novel features: First, the system avoids syntactic violations altogether, thus freeing the user from any syntactic concerns. As a consequence, Forms-XML does not require the user's any knowledge about XML. Second, since a working document is free of syntactic violations, it makes sense for the system to generate required elements and required element slots (placeholders for the user to add elements) automatically in the process of document construction. They serve as prominent guides to the user to produce valid documents. Third, the proposed model and algorithms are abstract and user-interface-neutral, being able to support user interfaces of various types. In particular, the simple editing process Forms-XML supports enables the generation of its simple form-based user interface.

While Forms-XML places more control over its user's editing actions, it leaves enough freedom for the user to construct any desirable valid document. The system imposes minimal constraints on the order that a user adds elements. In principle, when the given schema permits iterations of child elements, the user adds child elements by working one iteration at a time. This is consistent with the user's usual editing behavior.

In this paper, we use XML data and XML documents interchangeably unless distinguished otherwise. The proposed model and algorithms are applicable to DTD-compliant XML editors in general, not just Forms-XML. They work for both DTD and XML schemas. We use the term "DTD-compliant" instead of "schema-compliant" to emphasize our focus on XML syntax. In the next section, we review some related work. In Section 3, we lay the required theoretic foundations for regular expressions and finite automata. Section 4 depicts the model and algorithms for maintaining DTD-compliance. We describe the process for generating elements and element slots in Section 5. In Section 6, we consider major implementation issues. The form-based user interface that Forms-XML supports is briefly demonstrated in Section 7. The current status of this project is reported in Section 8 together with our future plan. Finally, concluding remarks are made in Section 9.

2. Related Work

Many XML editors are available at present while only a handful of XML editing/interactive components have emerged. (A long list of XML editors with short descriptions has been posted at [28], which is no longer running though.) We shall refer to these editors and components as XML editing systems collectively. As pointed out in Section 1, many XML editing systems take a detect-and-correct approach to syntactic violations, which is fundamentally different from the avoidance approach we take. Some of these systems do provide guidance or hints to the user during the editing process as Forms-XML does. However, the guidance or hints they provide are typically inaccurate, being too loose to guarantee the validity of the resulting document. (That is a major reason why they need a validity checker.)

We can classify the user interfaces of XML editing systems into 3 types: user interfaces based on text views, tree views, and presentation views. (A system may support multiple views.) A text view allows the user to edit tags as in a text editor [4]. A tree view displays the hierarchical structure of an XML document as a tree and provides operations for inserting and deleting tree nodes [20]. A grid view is considered as a variant of a tree view, which is a generalization of the grid control used for showing tabular data [3]. A presentation view results from applying some kinds of transformations to an XML document in order to present a desirable user interface to the user [2][6].

An XML editing system that supports a presentation view may appear differently depending on the transformation it supports. In the most general case, if it supports transformations by programming, the editing system would, in principle, be able to generate any desirable user interface [6]. However, the effort and technical skill for developing such a presentation view would be excessive. Two types of presentation

views that are more restricted have emerged. A word-processor view, hiding the tags of an XML document, frequently appears as an extension of a visual text editor such as Microsoft Word. A form view displays a hierarchy of forms, which contain controls for interacting with the user [17][18]. A form view or a word-processor view may be customized with modest effort.

As it generates form-based user interfaces, Forms-XML can be considered as a form generator based on a data model specified by an XML schema (or DTD). Conventional form generators, which are based on the relational data model, have been used extensively [14][16][19]. However, they cannot be adapted to XML data in general since XML has a structure much more complicated.

The editing process Forms-XML supports falls within the scope of syntax-directed editing. Syntax-directed editing has been researched extensively in the 80's [11][15]. We have observed two fundamental differences between XML editing and syntax-directed editing in the 80's. The former deals with structured documents while the latter focused on programming languages. When one edits a program, there is a strong tendency that one will produce the tokens of a program statement sequentially. Syntax-directed editing takes advantage of this trait. In contrast, while editing a structured document, the user must have far more freedom in selecting his actions, not necessarily producing document elements in sequential order. On the other hand, syntax-directed editors for programming languages are used by (perhaps non-professional) programmers while the user interface of Forms-XML is targeted for end users.

Despite the many available XML editing systems, their model and algorithms have not been published as far as the authors know. The current work is most related to Rita, an early editor prototype for manipulating structured documents. The concept of

"DTD-compliant" was introduced in a report regarding Rita [7], referred to as "subsequence-incomplete". The developers of Rita established a non-deterministic automaton model, and proposed algorithms based on multiple shortest paths in order to maintain documents subsequence-incomplete. Perhaps because they considered a document model more general than XML, their approach was inadequate in terms of both efficiency and completeness. As a consequence, subsequence-incompleteness was not included in the Rita system eventually.

The authors have outlined the ideas regarding DTD-compliant XML editors in a previous paper [12]. The current work is a refinement of that work from the perspective of an interactive/editing component, with full exposition of the concepts, theories, and implementation considerations.

3. Glushkov Automata

A well-formed XML document has a nested multi-level hierarchical structure. For the purpose of syntactic validity, it is sufficient to consider the syntactic constraint imposed on each element and its child elements separately. The content type of an element is defined by a regular expression, which determines the valid sequence of occurrences of its child element types. We adopt the notation of DTD to express regular expressions.

Let us start with simple regular expressions. 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 [1][5][10], defined as follows:

(1) Every symbol of Σ is a state of G. G has two additional states s and f as its start and final states, respectively.

(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.

$\delta(s, \$) = f$ if L contains the empty string, 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 simple expression E = *((a | b)*, c)* as an example. We have first(E) = *{a,b,c}*, last(E) = *{c}*, follow(*a*) = follow(b) = *{a,b,c}*, and follow(c) = *{ }*. 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 Є last(E1) and every v Є first(E2). (A regular expression E has a natural tree representation. A subexpression of E corresponds to a subtree of the tree representation of E.) 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 Є last(E1) and every v Є 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 the DFA G in Figure 1, *(a,c)* and *(b,c)* are sequence edges as well as forward edges. *(s,a)*, *(s,b)*, *(s,c)* and *(c,f)* are forward edges. *(a,a)*, *(a,b)*, *(b,a)*, and *(b,b)* are iteration edges as well as backward edges.

For a subexpression E1 of E, let A(E1) denote the set of symbols in Σ that E1 covers. For instance, if E1= *(a | b)**, then A(E1)= *{a,b}*. Let reachable(u) denote the set of states in G reachable from state u, i.e. reachable(u) = {z | there exists a path in G from u to z}. A forward path is a path only consisting of forward edges. Let f-reachable(u) denote the set of states in G reachable from u through forward edges, i.e. f-reachable(u) = {z | there exists a forward path in G from u to z}. With these definitions, we can make the following observations.

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

Proof: We can label all symbols in E such that the labels are in increasing order from left to right. The forward edges are always from symbols with smaller labels to symbols with larger labels. Thus they cannot form cycles. Q.E.D.

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

Proof: The lemma is trivial if E1 is a symbol. It is sufficient to prove the lemma on the basis that the child subexpressions of E1 satisfy the lemma. There are three cases:

(1) E1 is an iteration, i.e. E1= E2* for some subexpression E2, which satisfies the lemma. Since A(E1)=A(E2), first(E1)= first(E2), and last(E1)= last(E2), E1 satisfies the lemma as E2 does.

(2) E1 is a choice, i.e. E1= (E2 | E3), where E2 and E3 satisfy the lemma. For any x $\in$ A(E1), we have either x $\in$ A(E2) or x $\in$ A(E3), say x $\in$ A(E2). (The case for x $\in$ A(E3) is symmetric.) Since E2 satisfies the lemma, there exists some u $\in$ first(E2) and v $\in$ last(E2) such that x $\in$ f-reachable(u) and v $\in$ f-reachable(x). Since first(E1)= first(E2)$\cup$first(E3) $\supseteq$ first(E2) and last(E1)= last(E2)$\cup$last(E3) $\supseteq$ last(E2), we have proven the lemma for E1.

(3) E1 is a sequence, i.e. E1= (E2, E3), where E2 and E3 satisfy the lemma. For any x $\in$ A(E1), we have either x $\in$ A(E2) or x $\in$ A(E3), say x $\in$ A(E2). (The case for x $\in$ A(E3) is similar.) Since E2 satisfies the lemma, there exists some u $\in$ first(E2) and z $\in$ last(E2) such that x $\in$ f-reachable(u) and z $\in$ f-reachable(x). Since first(E2) $\subseteq$ first(E1), we have u $\in$ first(E1). Since E3 satisfies the lemma, there exists some w $\in$ first(E3) and v $\in$ last(E3) $\subseteq$ last(E1) such that v $\in$ f-reachable(w). Since E1= (E2, E3), there is a sequence edge from z to w. We have found a forward path from x to z, w, and finally v $\in$ last(E1), which completes the proof for the lemma. Q.E.D.

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

Proof: From Lemma 2, there exists x1, x2 $\in$ first(E1) and z1, z2 $\in$ last(E1) such that u $\in$ f-reachable(x1), z1 $\in$ f-reachable(u), v $\in$ f-reachable(x2) and z2 $\in$ f-reachable(v). On the other hand, there exists an iteration edge from z2 to x1 and an iteration edge

from z1 to x2. We have found a cycle connecting x1, u, z1, x2, v, z2, and back to x1. This completes the proof. Q.E.D.

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' [1][5][10], 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 $\delta(x_i, origin(x_j)) = x_j$.

Consider the regular expression E' = *((a | b)*, a)*. E' can be mapped to the simple regular expression E = *((a | b)*, c)* by renaming the second occurrence of *a* in E' to *c*. We have constructed the DFA, G, for E as shown in Figure 1. Thus the Glushkov automaton G' for E' can be constructed from G by replacing the label *c* on all edges in G by label *a*. The resulting NFA is shown in Figure 2.

Notice that 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 while labels on edges of G' are used to represent the documents saved to the disk.

Even though the Glushkov automaton G' for a general regular expression E' is non-deterministic in general, the W3C XML specification has imposed the constraint

that regular expressions used for defining the content models of element types must be deterministic for compatibility with SGML [22]. In other words, the Glushkov automaton G' must be deterministic, which can be used to recognize valid documents efficiently.

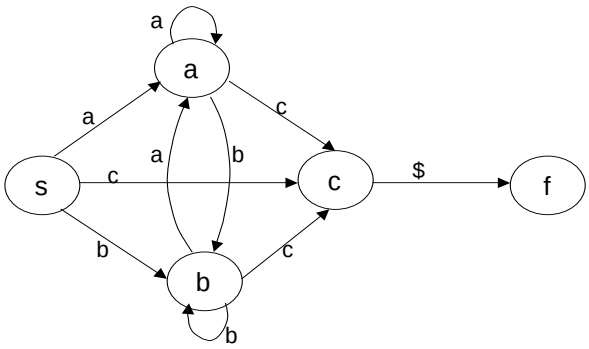

Figure 1. Glushkov automaton G for ((a | b)*, c)

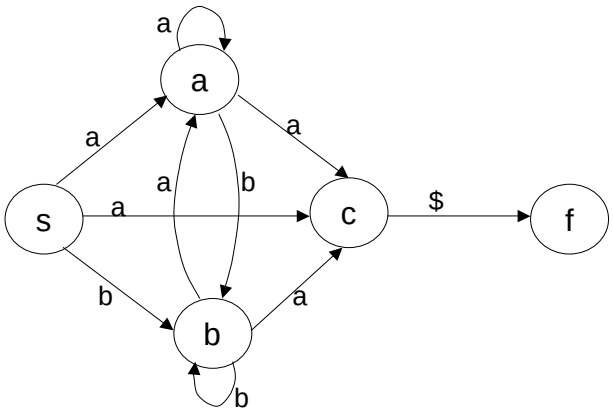

Figure 2. Glushkov automaton G' for ((a | b)*, a)

4. Maintaining DTD-Compliance

Assume that regular expression E', over the alphabet $\Sigma' = \{y_1, y_2, \ldots, y_m\}$, is used as the content model of an element type where $y_1, y_2, \ldots, y_m$ are its child element types. Let E be the simple regular expression associated with E' over alphabet $\Sigma = \{x_1, x_2, \ldots, x_n\}$ where $x_1, x_2, \ldots, x_n$ are symbols that make sense to users for distinguishing the multiple occurrences of child element types in Σ'. For instance, if a paper has a content model specified by E'= *(section+ | (pre-section, section*))*, one can define E= *(section_without_pre-section+ | (pre-section, section_with_pre-section*))* where *section_without_pre-section* and *section_with_pre-section* are used to distinguish the two occurrences of *section* in E'.

Let G and G' be the Glushkov automata for regular expressions E and E', respectively, as constructed in the last section. We shall use G to maintain the states of a working document upon interactive editing while using G' to store and restore the document states. From the definition of Glushkov automata, it is apparent that a valid document corresponds to a path in G (G') from the start state s to the final state f, and vice versa.

Forms-XML, with the goal of producing a valid document at the end, 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 system always maintain a DTD-compliant document, which is a subset of a valid document. To be precise, in a DTD-compliant document, the child elements of an element form a subsequence of a valid child element sequence. In other words, a DTD-compliant document corresponds to a subsequence of some path in G from s to f. Take the regular expression *(a?, b, (c, d)+, e)* as an example. The string, *acdde*, is not valid but is DTD-compliant since it is a subsequence of the string, *abcdcde*, which is valid.

Using a DTD-compliant editor like Forms-XML, a user starts with an empty document and makes element additions under the control of the system so that the working document is always DTD-compliant. A user can delete an arbitrary element from the working document; the resulting document is still DTD-compliant apparently. A user iterates the process of element addition and deletion so as to complete a desirable valid XML document eventually.

With this model, a working 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), $1 \leq i \leq p-1$. The actual document is the sequence {origin(z_2), origin(z_3), …, origin(z_{p-1})}. The main issue is, when the user intents to insert an element between a pair of consecutive elements (states), say z_i and z_{i+1}, how to compute an appropriate set of candidate elements for the user to select from so that the insertion of the selected element results in a DTD-compliant document. To be formal, the system computes a candidate state set $C \subseteq \Sigma$ for the user to select from. Once the user selects a state $x \in C$, the element origin(x) $\in \Sigma'$ is inserted into the working document.

Without loss of generality, assume that the user intends to insert an element between two states u and v where v $\in$ reachable(u). A candidate state set C is a representation of the "possible" paths for 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. Consider the Glushkov automaton G for the regular expression E= *(a, b, c*, (d | e+))** as shown in Figure 3. Due to the outer iteration of E, every state is reachable from every other state (except s and f). If states in C just had to satisfy the necessary condition, C would contain all states in any case. The necessary condition is too loose to give accurate suggestions. We thus aim at computing a "minimal" candidate state set that does not involve cycles and detours (e.g. unnecessary

backward edges) in general. Cycles are involved in a candidate state set only in restricted cases. The following lemma characterizes “direct” paths from u to v.

Lemma 4. Let u and v be states in G for which v Є reachable(u). Assume v ∉ f-reachable(u). Then there exists a subexpression E1* of E covering u and v such that w Є f-reachable(u) for some w Є last(E1) and v Є f-reachable(z) for some z Є first(E1). This constructs an acyclic path P connecting u to v. (If the subexpression E1* is chosen to be the smallest one covering u and v, such a path P is referred to as a minimal backward path.)

Proof: Let E2 be the smallest subexpression of E covering both u and v. Assume to the contrary that E2 and all its ancestors are not iteration subexpressions, i.e. they are sequence or choice subexpressions. Consider any ancestor E3 of E2. E3 may only induce forward edges from states in its left subexpression to states in its right subexpression. However, both u and v are covered by one of its subexpression. Thus the forward edges induced by E3 have nothing to do with whether v is (forward) reachable from u. In other words, whether v is (forward) reachable from u depends on E2.

If E2 is a choice subexpression, since u and v are covered by the two subexpressions of E2, respectively, u cannot reach v, which is a contradiction. In the case that E2 is a sequence subexpression where E2= (E3, E4), there are two cases: If u Є E4 and v Є E3, then u cannot reach v, which is a contradiction. If u Є E3 and v Є E4, then u can reach v through forward edges by applying Lemma 2 to both E3 and E4, which is also a contradiction. We have derived contradictions in all cases. Thus there must exist an iteration subexpression E1* covering both u and v. From Lemma 2, there exists w Є last(E1) such that w Є f-reachable(u), and z Є first(E1) such that v Є f-reachable(z). Since (w,z) is an iteration edge, we have found a path P connecting u to v.

To prove that P is acyclic, assume to the contrary that P visits a state p more than once. Since the subpaths connecting u to w and z to v, respectively, are forward paths, p appears on each subpath only once. Now the two subpaths connecting u to p and p to v form a forward path connecting u to v. This contradicts the assumption v ∉ f-reachable(u). Thus P is acyclic. Q.E.D.

While computing the candidate state set C between two given states u and v where v Є reachable(u), we distinguish two cases: (u,v) ∉ H (document non-valid locally) and (u,v) Є H (document valid locally) where H is the edge set of G. In Algorithm FindCandidateStates1, for the case (u,v) ∉ H, C is composed of the intermediate states in the acyclic paths from u to v determined by Lemma 4. (If the user wants a path with cycles, she can always construct an acyclic path first and add cycles later on as Algorithm FindCandidateStates2 supports.)

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

The philosophy behind Algorithm FindCandidateStates1 is for the user to add a minimal set of elements between u and v in order to render the current non-valid document valid locally. Let us illustrate FindCandidateStates1 with the Glushkov automaton G in Figure 3. The state pair *(a,e)* satisfies *e* Є 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}*.

Given two states u and v where v Є reachable(u) and (u,v) Є H, Algorithm FindCandidateStates2 computes a candidate state set C. 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 may be inserted between u and v by adding its symbols to C.

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

Consider the Glushkov automaton G in Figure 3. For the state pair *(a,b)*, we have C=

{ }, which indicates no element 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* Є 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* Є last(*e**) and *e* Є 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.

Lemma 5. At the completion of FindCandidateStates1 and FindCandidateStates2, for any state x in the candidate state set C, we have x Є reachable(u) and v Є reachable(x).

Proof: This lemma comes from the computation for C with the support from Lemma 2 and 3. Q.E.D.

Theorem 1. With the insertions supported by Algorithm FindCandidateStates1 and FindCandidateStates2, the user always produces a DTD-compliant document, and can construct any desirable valid document D' from a DTD-compliant document D if D is a subset of D'. (In other words, the approach is both sound and complete.)

Proof: From Lemma 5, the system always produces a DTD-compliant document. On the other hand, for any valid document D' containing a DTD-compliant document D, D corresponds to a subsequence of D'. Thus one can construct D' from D by adding a set of subpaths to D. For each subpath, the user can repeat the process of adding a forward subpath before a backward edge and then adding the backward edge, (The user has much more freedom though.) and eventually complete the subpath. This completes the proof. Q.E.D.

Lemma 6. All algorithms considered in this section take linear time, linear in the size of the Glushkov automaton G.

Proof: The function f-reachable( ) can be computed in linear time initially. From Lemma 1, the states in G can be so labeled that all forward edges are from states with smaller labels to states with larger labels. Thus one can compute f-reachable(x) for every state x in the reverse (i.e. decreasing) order of the state labels with a single pass. Algorithm FindCandidateStates1 and FindCandidateStates2 take linear time apparently. Q.E.D.

Algorithm FindCandidateStates1 and FindCandidateStates2 restrain insertions to guarantee DTD-compliance while leaving enough freedom to the user. In principle, the user can add elements in an arbitrary order within one iteration of a subexpression. However, the user can only add a new iteration right before or right after an existing iteration. (The user cannot add a new iteration from the middle of an existing iteration.) We believe this "one iteration at a time" working style is consistent with the user's common editing behavior.

So far we have relied on the user to distinguish the multiple occurrences of the same child element type, i.e. the user fills in an element slot by selecting from a candidate state set $C \subseteq \Sigma$. On the other hand, there may be cases that there is no need to distinguish the multiple occurrences of an element type. (A case that occurs frequently is the multiple element occurrences caused by an iteration with min-occur/max-occur bounds.) In these cases, the candidate state set $C \subseteq \Sigma$ can be mapped to a candidate element list $C' \subseteq \Sigma'$ where $C' = \{y \in \Sigma' \mid y = \text{origin}(x) \text{ for some } x \in C\}$. The user then fills in an element slot by selecting an element y from C' while the system can pick an arbitrary state $x \in C$ such that y=origin(x) for insertion into the working document.

We have considered the interactive editing of DTD-compliant documents. There is another issue with regard to the restore of DTD-compliant documents. A DTD-compliant XML editor like Forms-XML relies heavily on state information

about a working document. It is easy for the system to parse a valid document and reconstruct its state sequence since the Glushkov automaton G' is deterministic. However, there is no way to reconstruct a state sequence from a DTD-compliant document in general if the regular expression is not simple. Our solution is to write out a valid document containing a DTD-compliant but non-valid working document. Those elements that are not present in the working document are marked with special attributes indicating their non-existence in the document.

As described above, Forms-XML may not be able to restore a well-formed but non-valid document, e.g. produced by other XML tools. We consider this limitation reasonable. Forms-XML is dedicated to producing valid documents. Non-valid documents, requiring syntactic corrections, are beyond its scope.

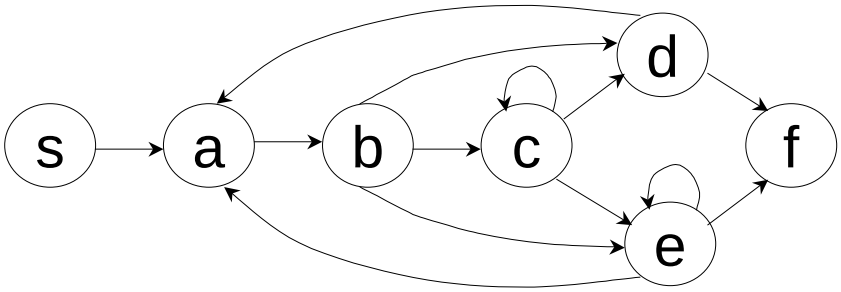

Figure 3. Glushkov automaton G for (a, b, c*, (d | e+))*

5. Generating Elements and Element Slots

In Forms-XML, a working document never contains syntactic violations. It is thus highly likely that the current document is indeed a subset of the target valid document the user desires. In this case, it makes sense for the system to generate elements and element slots (placeholders for the user to add elements) automatically based on the state of the current document. This approach has the following benefits: First, the generated required element slots serve as prominent guides to the user to produce a valid document. Second, the generation of elements and element slots reduces the user's workload. Third, the user can see the effect of an element insertion quickly. If the insertion is not what she wants, the user may be able to detect and undo it earlier.

A DTD-compliant editor like Forms-XML can generate required elements automatically upon the insertion of an element. An element is required if it is present in all valid documents containing the current DTD-compliant document. This policy certainly applies to the empty document initially. In particular, the required elements generated initially must be present in all valid documents. Normally they should not be deleted unless their parent element is deleted. As an example, consider the regular expression *(a, (b, c)?)*. Initially, *a* is a required element. Once element *b* (or *c*) is inserted, *c* (or *b*) becomes a required element.

Consider the state sequence representation of a DTD-compliant working document as in the last section. Given two states u and v where v ∈ reachable(u), 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 [8]. 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. When there are multiple articulation points, the maximum flow-like algorithm is

executed multiple times. The algorithm takes linear time, linear in the size of the Glushkov automaton G.

Forms-XML can generate not only elements but also element slots. Given two consecutive elements u and v in the current document, if (u,v) $\notin$ H where H is the edge set of G, then every valid document containing the current document contains at least one element between u and v. An element slot is thus generated automatically between u and v for the user to fill in. Such element slots are referred to as required element slots since the user should typically provide appropriate elements for these placeholders in order to render the current not-yet-valid document valid. While generating a required element slot, the system also computes an associated candidate state set C using Algorithm FindCandidateStates1. The user then fills in a required element slot by selecting a desired state (element) from C (candidate element list C').

Consider a content type defined by regular expression *(a, (b | c))*. While the system generates the required element *a*, it also generates a required element slot following *a*, together with a candidate element list containing *b* and *c* for the user to select from. In general, required element slots appear as prominent hints to the user, indicating that the current document is not valid. The system guides the user to produce a valid document step-by-step by generating required element slots for the user to fill in iteratively. Only when the user has filled up all required element slots and the system generates no more required element slots, the working document becomes valid.

In additional to generating required element slots automatically, a DTD-compliant editor can generate optional element slots upon the user's requests. (The user may ask the system to show all optional element slots or just to show the optional element slots near a selected reference position.) Given two states u and v where v $\in$ reachable(u) and (u,v) $\in$ H, the system computes the candidate state set C using Algorithm

FindCandidateStates2, and generates an optional element slot between u and v if C is not empty. The user can fill in optional element slots in the same way as she fills in required element slots by selecting from C (candidate element list C').

Consider the regular expression *(a,b)*+. Initially, the system generates *a* and *b* as required elements, which results in the state sequence *sabf*. Applying Algorithm FindCandidateStates2 to each pair of consecutive states, the resulting candidate state sets are: C= { } for the pair *(a,b)* and C= *{a,b}* for the pairs *(s,a)* and *(b,f)*. Thus the system may generate an optional element slot between *s* and *a*, and one between *b* and *f*, but none between *a* and *b*.

To summarize, Forms-XML supports a generation-oriented editing process consisting of three phases: the system generates required elements and required element slots; the user fills in required element slots to make the working document valid; the user requests the system to display optional element slots and fills in the optional element slots to complete a desired valid document eventually. The user typically, but not necessarily, edits a document in this order. Also, after the user fills in an element slot, the system may generate required elements and required element slots again, which results in the iteration of the three phases.

We have considered the case of element insertions. On the other hand, when the user deletes an element from the working document, the system does not try to generate required elements; though it may generate required element slots at the place the element is deleted. We insist the principle that, if the working document is not valid, some required element slots are generated for the user to fill in.

It should be noted that element slots are entities displayed on the user interface rather than real things in the working document. At most one element slot is displayed

between two consecutive elements. (It makes little sense to show more than one element slot between two consecutive elements. The user's fill-in of an element slot would have effect on its neighboring slot if one existed.) An element slot is highly dependent on its consecutive elements. Once its consecutive element is deleted, an element slot is deleted by the system, and a new element slot may be generated again.

As described in this section, Forms-XML takes the approach of generating element slots for the user to fill in while avoiding the conventional insertion operations. The two approaches differ in that the latter supports a try-and-err editing process while the former clearly exhibits to the user "legal" positions for insertions. We believe the former approach is friendlier to the user in most cases.

6. Implementation Considerations

The algorithms and editing process described in previous sections essentially form an abstract editing model for XML documents. In practice, Forms-XML must be able to handle all constructs of an XML schema, which adds many details to the abstract editing model. In this section, we report our main implementation decisions, in particular those related to syntax handling.

Given a regular expression E', its Glushkov automaton G (G') can be constructed in $O(n^3)$ time with a naïve implementation [1] where n is the size of E'. Bruggemann-Klein proposed a more sophisticated implementation which takes $O(n^2)$ time (the size of G) [5]. Note that an XML schema is typically composed of a large number of small-size regular expressions (i.e. n is very small). We have thus adopted the naïve implementation while representing the function values of first, last, follow, and f-reachable as bit vectors. This approach achieves a simple program structure as well as an effective run time proportional to n^2.

It was pointed out in the last section that required elements generated initially must be present in all valid documents. Normally they should not be deleted unless their parent element is deleted. However, note the difference between element types and element instances. An element type may have multiple instances in the input document if it is part of an iteration expression or if it has multiple occurrences in a regular expression. When an element type is required initially, not all its instances are required. We thus leave more freedom to the user. An element cannot be deleted only when its element type is required initially and it is the only instance of the element type in the current document.

We have only considered elements in previous sections. As for attributes, they are

treated as if they were elements of simple types. For each element, its attributes are considered to precede its child elements. The attributes always appear in the same order as they were declared in the input schema. Even though attributes exhibit the same behavior as child elements, we do not represent the sequential constraints imposed on them as regular expressions explicitly to minimize the overhead for setting up the Glushkov automata.

In principle, elements of complex types serve as containers while elements of simple types serve as the constituents of containers. However, simple-content complex types are an exception, which has simple-type content as well as contains attributes. To make things simple, we treat an element of simple-content complex type as a pure container, which contains attributes as well as a simple-type child element. The latter stands for the content of the element.

In addition to normal regular expressions, XML schemas support min-occur/max-occur bounds in iterations. Apparently one can convert an iteration expression with min-occur/max-occur bounds into a regular expression without these bounds:

$E^{\text{min-occur=p, max-occur=q}}$ = (E, E, …, E (p times), E*) if q is unbounded;

$E^{\text{min-occur=p, max-occur=q}}$ = (E, E, …, E (p times), E?, E?, …, E? (q-p times)) if q is bounded.

The resulting regular expression may have much more element occurrences than the original expression, though. Fortunately, p and q are typically very small in practice, which justifies the simple conversion.

In an XML schema, an element may be declared to have mixed content, which allows its child elements to be interleaved with character data. Forms-XML takes care of mixed content upon the computation of the candidate state set C (candidate element

list C'). For an element with mixed content, when the system computes C (C') for a pair of its child elements, a special element #Text is always added to C (C') for the user to select from. #Text serves as a placeholder for interleaved character data.

XML schemas support a special grouping operator "all". All elements in an all group may appear once or not at all, and they may appear in any order. Since an all group cannot be used together with other grouping operators, we have implemented all as a separate procedure independently of the syntactic model described in previous sections. Forms-XML displays as many required element slots as the number of required elements in an all group initially for the user to fill in. Optional element slots may then be displayed between a pair of consecutive elements as before.

We have implemented all material in this paper as a JavaScript program executed on Microsoft Internet Explorer, which generates HTML forms as its user interface. The JavaScript code together with the MS browser component was then encapsulated within an Active Control shell, which constitutes the Forms-XML component. The Internet browser platform allows the presentation style of HTML forms to be specified by an external CSS style sheet [21]. This frees Forms-XML from the drudgery of handling presentation styles.

7. Form-Based User Interface

We have established an abstract and comprehensive editing model for XML documents, on top of which one can implement a user interface for interactive editing. In principle, the abstract editing model, being user-interface-neutral, can support user interfaces based on both tree views and presentation views among the 3 types of XML editing systems reviewed in Section 2. We have chosen to build a form-based user interface for Forms-XML. Forms are easy to use and well accepted by users with little computer skill. They have emerged as the typical user interface for e-business applications.

In the following, we take the schema for papers for the Extreme Markup Languages Conference [9] as an example to demonstrate the form-based user interface of Forms-XML. Figure 4 shows the top-level form when an empty paper is created. This form displays 3 levels of elements: *paper* has two required child elements *front* and *body*; *front* has 2 required child elements *title* and *author* while *author* has 4 required child elements; and *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. One may fill in the data fields (*fname* and *surname*) as one usually does for an HTML form.

The system has a parameter *levelsInForm* (3 in this case) that determines the number of levels of elements a form may contain. An element at the bottom level in a form, *address* and *bio* in this case, 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 and attributes for editing. A child form may have child forms again so that child forms may nest indefinitely.

Notice that if *levelsInForm* is large, the form has essentially a tree view. We believe that tree views with deep structures are confusing visually and difficult conceptually for non-technical users. Thus *levelsInForm* is typically set to a small number so that the form appears more like a conventional form than a tree view. Moreover, by setting *levelsInForm*, one may adjust the depth of nested forms.

Suppose that one wants to add co-authors in the form shown in Figure 4. One can display optional element slots around a selected element, say *author*. When one moves the pointer over the element *author*, a small PLUS icon pops up near the pointer. If the user clicks the icon, the optional element slots immediately before and after the current element are then displayed as shown in Figure 5. (Otherwise, the current PLUS icon will pop off when the pointer is moved over another element.) The user can fill in an element slot by selecting from a menu of candidate elements generated by the system.

Also shown in Figure 5 is a MINUS icon. Like a PLUS icon, when the user moves the pointer over an element, attribute or element slot, a MINUS icon may pop up near the pointer. By clicking MINUS, one can remove the current element (and all its descendants), attribute, or element slot. The pop-up of an icon is context-sensitive. A PLUS (MINUS) icon pops up only when the element on focus has adjacent optional element slots (is allowed to be removed). Both PLUS and MINUS may pop up at the same time.

Forms-XML is able to support more sophisticated layout of forms. Figure 6 shows a 2-column layout where some child elements occupy a single column and others an entire row. More about the layout of Forms-XML will be reported in a later report [13].

As illustrated, Forms-XML provides a simple form-based user interface, which is an enhancement of the familiar HTML forms with minimal editing commands. Our philosophy is to keep the forms as simple as the usual HTML forms as possible. The component only offers essential interactive editing functions; additional editing functions, if desired, are left to the application. Forms-XML supports a collection of API (application programming interface) methods and events for the application to customize and enhance its user interface.

The virtue of the user interface Forms-XML supports lies in its control-minimal, single-level layout of the child elements of an element. Consider an element with content type specified by regular expression E= (a, b*, (c | d)). A simple-minded approach to laying out the child elements is to introduce two controls (e.g. radio buttons and grid controls) to represent choice and iteration, respectively, which results in a two-level structure [17][18]. In general, except for very simple XML schemas, this approach tends to clutter forms with many controls and complex structures. In contrast, the DTD-compliant editing process enables Forms-XML to produce a control-minimal and level-minimal form layout with minimal editing commands. (On the other hand, Forms-XML supports a variety of controls for data fields.)

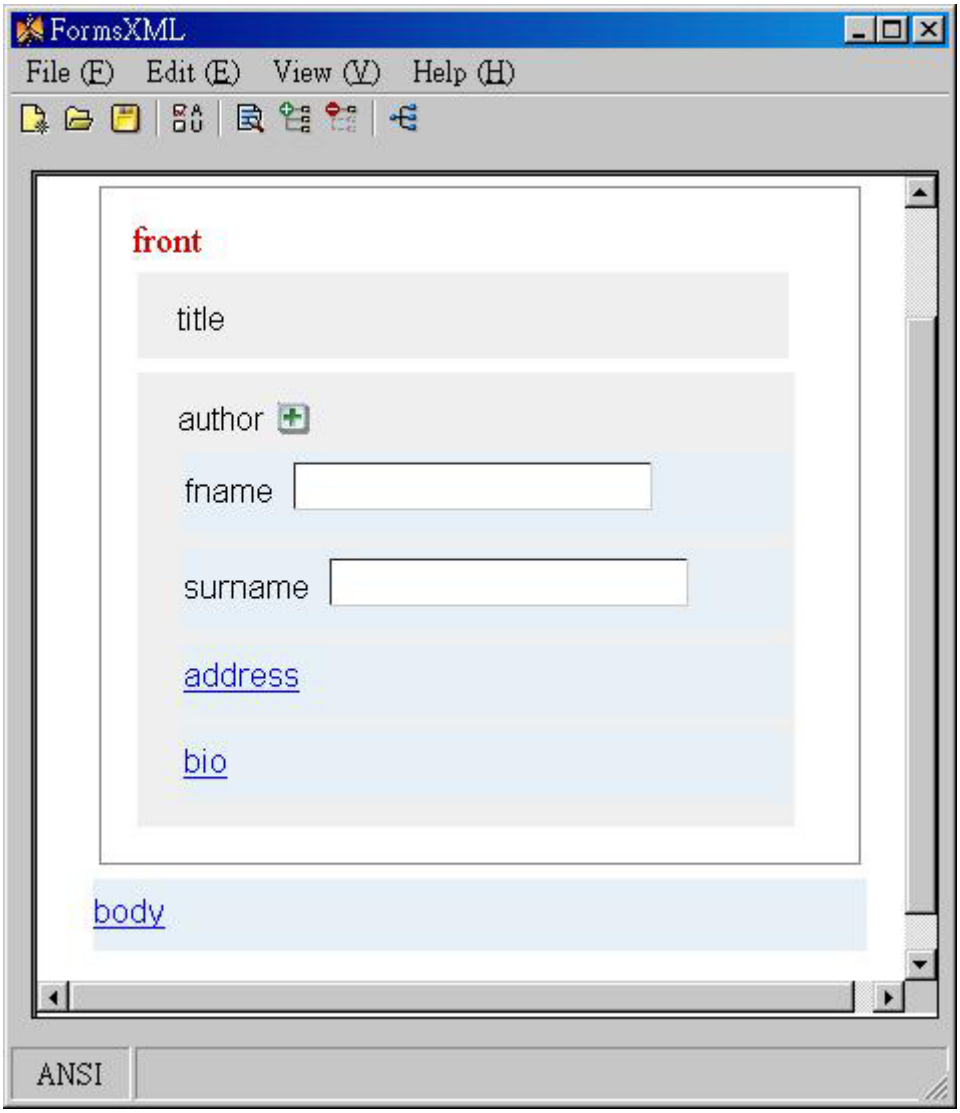

Figure 4. A top-level form

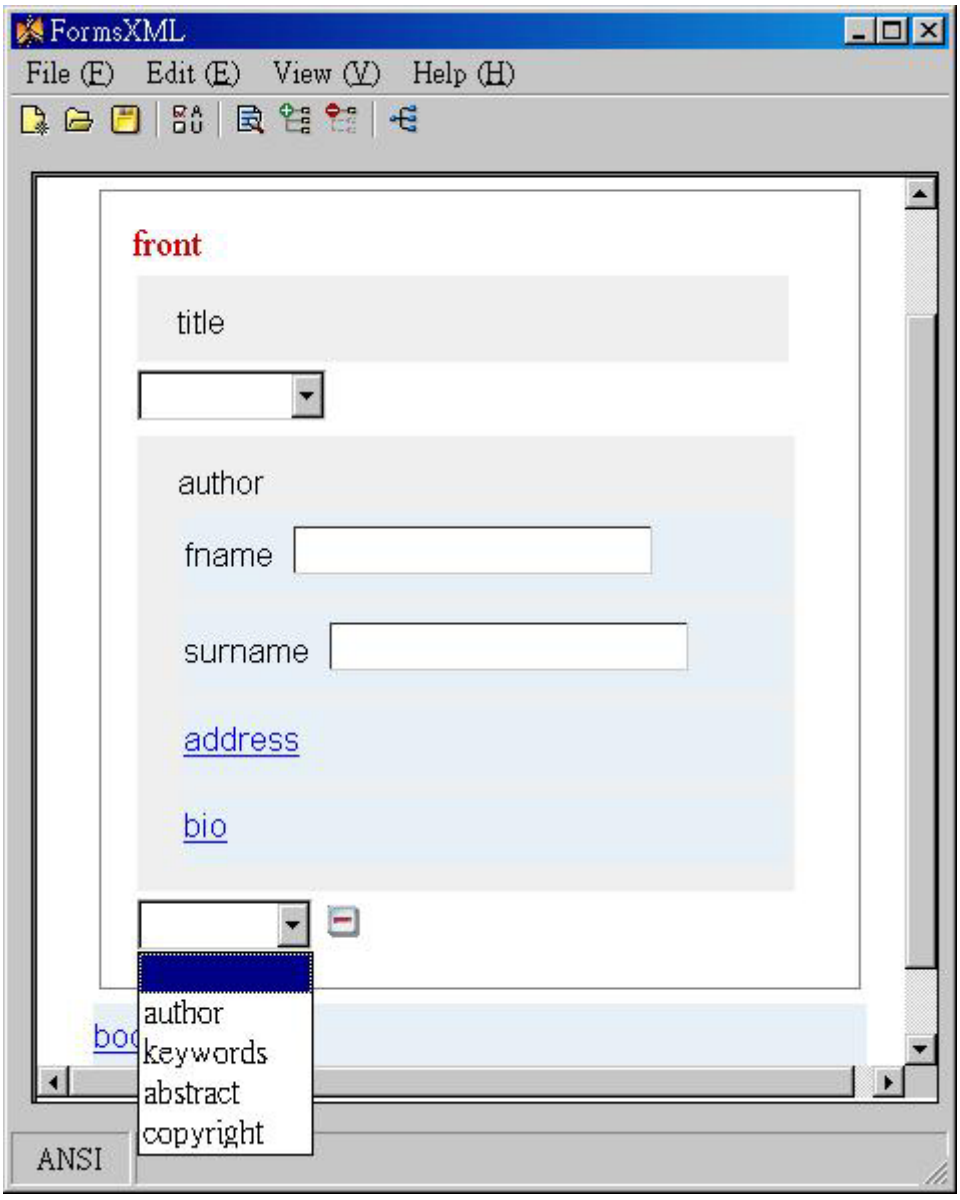

Figure 5. Optional element slots

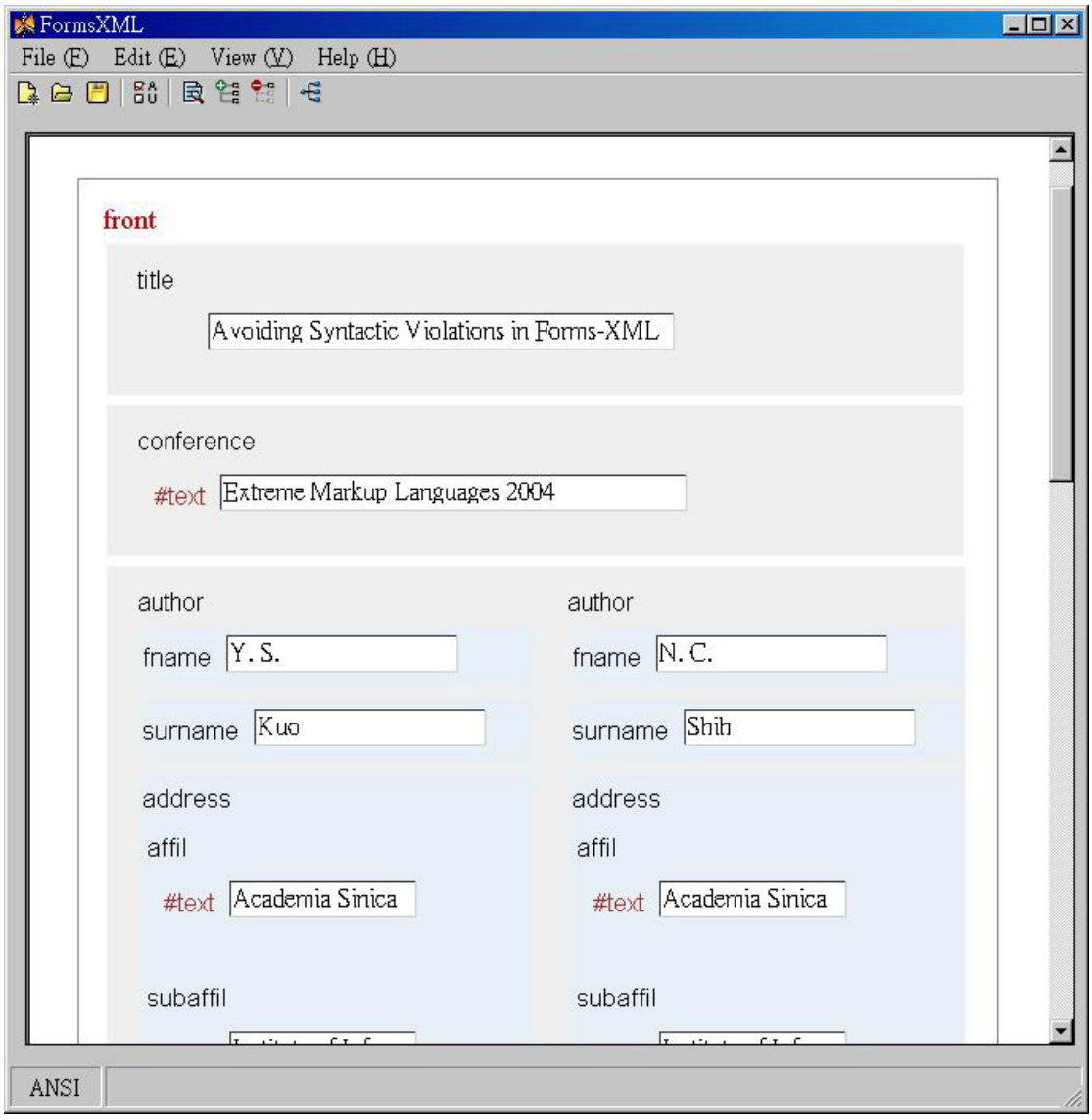

Figure 6. 2-column layout

8. Current Status and Future Work

So far we have completed a preliminary version of Forms-XML and a simple XML editor, the Valid XML Editor. The latter is a small Windows application (as shown in Section 7) that invokes the Forms-XML component. We use it as a vehicle for testing Forms-XML. We have tested the Editor against a few real-world XML vocabularies, among which is the schema (Extreme.xsd) for papers for the Extreme Markup Languages Conference [9]. (We used the Editor to create and edit the current paper.)

To convey how Forms-XML performs, we herein report some test results regarding the Valid XML Editor against the schema Extreme.xsd. (More experimental results will be reported in a later system-oriented paper [13] after more extensive testing is completed.) Extreme.xsd has 56 complex types to be processed, which contain 269 child element types in total. (Element types appearing in multiple types were counted multiple times. Attributes were not counted since they are not part of regular expressions.) The Valid XML Editor took 3.2 seconds to load Extreme.xsd, build the Glushkov automata, and compute the function f-reachable on a Celeron 1.2 GHz PC with 512 MB RAM running MS Windows 2000 Server, I.E. 6, and MSXML 4. We also observed the response time upon editing. During our editing process, the Editor took no more than 0.5 seconds to respond to the user's insertion operation, which includes computing required elements and slots and updating data structures and the display.

In additional to the extensive testing of Forms-XML, our on-going work is to provide more layout styles and options in its user interface. For instance, data fields that have the same parent element may be aligned for better appearance. The system provides an option to display a part of the XML data as tables when the data has a regular structure like relational data. On the other hand, the API that Forms-XML supports

needs to be refined continually. We anticipate applying Forms-XML to different applications, which would help refine and improve its API.

In the long run, we plan to build a visual development environment for Forms-XML. The development environment allows an application developer to specify the layout and presentation styles of the component visually. In addition, the developer can specify some kinds of transformations to be applied to the XML data, including permuting the child elements of an element, splitting and merging regular expressions, etc. All these would make the use of Forms-XML easier and enable Forms-XML to produce more flexible and versatile user interfaces.

9. Concluding Remarks

Many XML vocabularies have been defined for the interchange of data among organizations within different application domains. Building user interfaces for enterprise-level XML vocabularies is a costly and labor-intensive task. We have developed Forms-XML as a generic interactive component that generates form-based user interfaces for XML vocabularies.

As its unique feature, Forms-XML avoids instead of detects syntactic violations, that is, it always keeps a working document compliant with a given XML schema. The approach gains the following benefits: First, by avoiding syntactic violations altogether, Forms-XML can free the user from any syntactic concerns. Second, the system can generate elements and element slots in the process of document construction, which serve as prominent guides to the user to produce valid documents. Third, the simple editing process Forms-XML supports enables the generation of its simple form-based user interface.

Forms-XML falls within the area of XML editing systems, which have focused more on systems than theoretic aspects. The major contribution of this work is to establish automata theories and graph algorithms for handling the XML syntactic constraints based on regular expressions, which would potentially render these systems more robust and user-friendly.

Acknowledgements

This work has been partially supported by Remote Soft, Inc. under the contracts 05T-890701-CN and 05T-890701B-CN. We are grateful to our colleagues, Tyng-Ruey Chuang and Da-Wei Wang, at Academia Sinica for helpful comments and discussions on this work.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

[2] Altova, Authentic/Stylesheet Designer, http://www.xmlspy.com/

[3] Altova, XMLSPY, http://www.xmlspy.com/

[4] Architag International, XRay XML Editor, http://architag.com/xray/

[5] A. Bruggemann-Klein, "Regular expressions into finite automata", Theoretical Computer Science, 120, 1993, pp. 197-213.

[6] Blast Radius, XMetal, http://www.softquad.com/

[7] 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.

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

[9] Extreme Markup Languages 2004, http://www.extrememarkup.com/extreme/

[10] V. M. Glushkov, "The abstract theory of automata", Russian Math. Surveys, 16, 1961, pp. 1-53.

[11] A. A. Khwaja and J. E. Urban, "Syntax-directed editing environments: issues and features", Proc. ACM Symp. Applied Computing, March 1993.

[12] Y. S. Kuo, Jaspher Wang, and N. C. Shih, "Handling Syntactic Constraints in a DTD-Compliant XML Editor", Proc. ACM Symp. Document Engineering, Grenoble, France, Nov. 2003.

[13] Y. S. Kuo, N. C. Shih, and Lendle Tseng, "Forms-XML: generating form-based user interfaces for XML vocabularies", in preparation.

[14] A. Lulushi, Oracle Developer/2000 Forms, Prentice Hall PTR, 1999.

[15] T. F. Lunney and R. H. Perrott, "Syntax-directed editing", Software Engineering

Journal, 3(2), March 1988.

[16] S. A. Mamrak and S. Pole, “Automatic form generation”, Software – Practice and Experience, 32, 2002, pp. 1051-1063.

[17] Microsoft, Office InfoPath, http://www.microsoft.com/office/infopath/prodinfo/default.mspx

[18] NetBryx, XML Quik Builder, http://www.editml.com/

[19] J. Prosise, Programming Microsoft .NET, Microsoft Press, 2002.

[20] Tibco Software, Turbo XML, http://www.tibco.com/

[21] W3C, Cascading Style Sheets, Level 2, W3C Recommendation, May 12, 1998, http://www.w3.org/TR/REC-CSS2/

[22] W3C, Extensible Markup Language (XML) 1.0, W3C Recommendation, Feb. 10, 1998, http://www.w3.org/TR/1998/REC-xml-19980210

[23] W3C, XML Query, http://www.w3.org/XML/Query

[24] W3C, XML Schema Part 1: Structures, W3C Recommendation, May 2, 2001, http://www.w3.org/TR/xmlschema-1/

[25] W3C, XML Schema Part 2: Datatypes, W3C Recommendation, May 2, 2001, http://www.w3.org/TR/xmlschema-2/

[26] W3C, XSL Transformations (XSLT) 1.0, W3C Recommendation, Nov. 1999, http://www.w3.org/TR/xslt

[27] XML.org, The XML registry, http://www.xml.org/xml/registry.jsp

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