Checking Potential Validity of XML Documents*

Ionut E. Iacob†
Dep. of Computer Science
University of Kentucky
Lexington, KY 40506
eiaco0@cs.uky.edu

Alex Dekhtyar‡
Dep. of Computer Science
University of Kentucky
Lexington, KY 40506
dekhtyar@cs.uky.edu

Michael I. Dekhtyar§
Dep. of Computer Science
Tver State University
Tver 170000, Russia
michael.dekhtyar@tversu.ru

ABSTRACT

Document-centric XML documents used in numerous applications, may have a highly irregular structure. More importantly, the process of their creation often starts with an existing text, into which human editor introduces various markup. In such situations, it is very likely that the intermediate XML is almost never valid with respect to the DTD used for the encoding, so checking validity turns to be a badly chosen operation. In this paper we introduce the notion of potential validity of XML documents, which allows us to distinguish between XML documents that are invalid because the encoding is simply incomplete and XML documents that are invalid because some of the DTD rules guiding the structure of the encoding were violated during the markup process. We show that the set of potentially valid XML documents for a given DTD is context-free and we give a linear-time algorithm for checking potential validity for documents and document updates.

1. INTRODUCTION

EXAMPLE 1. *Consider a DTD shown on figure 1. Let us use the elements of this DTD to encode the text "A quick brown fox jumps over a lazy dog" in the following two ways:*

```
<r><a><b>A quick brown fox</b><e></e><c> jumps
 over a lazy</c> dog</a></r>

<r><a><b>A quick brown fox</b><c> jumps over
a lazy</c> dog<e></e></a></r>
```

We now ask two questions: is there a difference between these two XML fragments with respect to our DTD? And if yes, then, what is this difference?

Figure 2 depicts the DOM trees for both XML fragments. By comparing the encodings with the structure required by the DTD, it is easy to notice that both XML fragments are *not valid* [3] with respect to it. That is, a validating XML parser will not distinguish between these two fragments.

```
<!ELEMENT r (a+)>
<!ELEMENT a (b?, (c | f), d)>
<!ELEMENT b ( d | f)>
<!ELEMENT c #PCDATA>
<!ELEMENT d (#PCDATA | e)*>
<!ELEMENT e EMPTY>
<!ELEMENT f (c, b, e)>
```

Figure 1: A sample DTD.

We note, however, that there is an important difference between the two XML fragments above. The first fragment is not valid because the order in which the tags `c` and `e` are found in the text contradicts the DTD. At the same time, we can see that the second XML fragment does not contain any "hard" violations of the DTD, rather, it is simply an incomplete encoding that can be converted into a valid XML document by adding the two tags `d`: one inside `b` enclosing the same content as `b` and the other around the XML fragment `"dog<e></e>"`. Figure 3 shows how the DOM tree of the second fragment can be completed to produce a valid XML document.

The importance of having fast algorithms for checking post-updates XML documents validity is emphasized in [14]. Both database updates and XML editors would benefit of such algorithms. However, as shown in the example, the notion of XML (non-)validity is not sufficient to distinguish between the two classes of non-valid XML encodings: those that can be turned into valid XML documents only by inserting new elements and those that require elements deletion (and possible new insertions) in order to become valid.

Two major kinds of XML documents emerge from applications: *data-centric* and *document-centric*. Data-centric documents are characterized by a fairly regular structure and occur as a standard format for structured data exchange and representation of semistructured data. Document-centric XML has, in general, a much more irregular structure and is often encountered as the means of document markup. A distinction between the two aforementioned classes of non-valid XML documents is especially important for document-

*Technical Report #TR 403-04, Department of Computer Science, University of Kentucky, Lexington, KY 40506

†Work funded, in part, by a Collaborative Research Award from the National Endowment for the Humanities (NEH grant RZ-20887-02) and the Andrew W.Mellon Foundation.
‡Partially supported by NSF grant ITR-0219924.
§Work sponsored by Russian Fundamental Studies Foundation (Grant 04-01-00565). Also partially supported by NSF grant CCR 0100040, thanks to Judy Goldsmith.

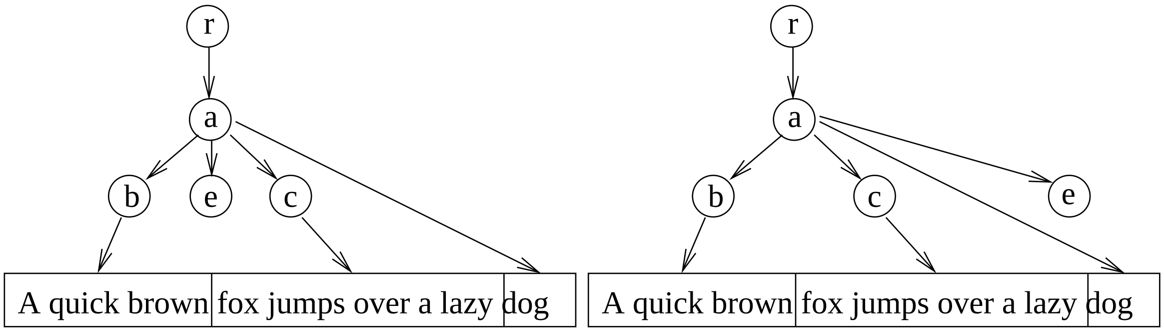

Figure 2: DOM Trees for Example 1

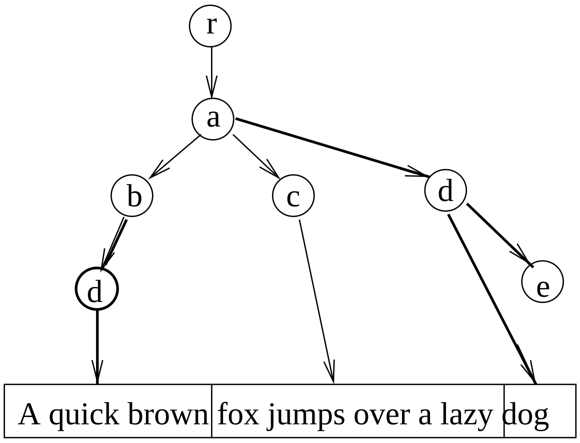

Figure 3: Extending encoding to obtain valid XML.

centric XML editors. As pointed out in [6], XML editors can be classified as *text editors* and *structural editors*. The majority of existing XML editors fall in the second category and they provide, in general, optional validity checking on user updates. However, in many document-encoding applications of XML, the human editor already has the entire content (PCDATA) of the document to be marked up. The task of the XML editing software in this case is to load the text and allow the human editor to insert (or delete) markup, *one tag at a time*. In such situations, the XML document may not become valid for a significant part of the editing process: the order in which the markup is introduced in the text may depend not on the specifics of the DTD, but rather on the modus operandi of the human editor. Imposing validity constrains for update operations might be too prohibitive in such cases: not any update operation (or a set of consecutive update operations) yields a valid document. Further update decisions are to be taken based on the current status of a document. At the same time, it is important to be able to verify at each moment of time, that the current XML fragment is "on track", i.e., that the human editor has not committed any structural error while introducing the markup (in which case markup deletion being required).

In this paper, we introduce the notion of *potential validity* of an XML document that formalizes the idea of distinguishing between the two classes of non-valid XML documents described above. We show that given a DTD, the set of all potentially valid XML documents for it is context-free. We present two implementations of potential validity checking: one based on the Earley's algorithm [7], and another, a fast and scalable linear-time (in the size of input XML documents) algorithm specific for the class of grammars that we construct to check potential validity. We also give algorithms for checking potential validity for basic update operations (delete, insert, and rename tags) on potentially valid XML documents.

The rest of the paper is organized as follows. In Section 2 we define potential validity and show how context-free grammars for checking it can be constructed. In Section 3 we discuss two different algorithms for checking potential validity and in Section 4 we present the results of testing. We give related work in Section 5 and conclude in Section 6.

2. POTENTIALLY VALID XML DOCUMENTS

2.1 Potential validity

Informally, we can specify *potential validity* of an XML document[1] as follows.

DEFINITION 1. *An XML document is* **potentially valid** *w.r.t. a given DTD if either the document is valid w.r.t the given DTD or it can be made valid by inserting more markup tags, from the given DTD, at some positions.*

As informally described in Section 1, we want to decide (given a specific DTD) whether or not a given XML document (presumably not valid) can be transformed into a valid XML document instance using only markup insertions. This assumption is consistent with a typical procedure of introducing XML markup into an existing text. Moreover, we want to determine whether or not an update operation on a potentially valid document yields a potentially valid document. We emphasize that we do not exclude markup deletion operations. However, a potentially valid document

[1]Through this paper, all XML documents (or simply *documents* unless other meaning is specifically stated) we consider are *well-formed* XML documents [3]. We call *XML string* a string representation of a document.

is either valid or it can become valid using only markup insertions, whereas a non potentially valid document requires also markup deletions (or renaming) to make the document valid.

We can, however, make Definition 1 more formal. First, we introduce some notation. Consider a DTD $T = \langle \Gamma, \mathcal{T} \rangle$, where Γ is the set of Element Type Declarations[2] [3] and $\mathcal{T}$ is the set of all element types defined in the DTD (i.e., the set of all left-hand sides of the element type declarations from Γ). In addition, we assume that one of the elements from $\mathcal{T}$, $\mathtt{r}$, will be the root element of XML documents to be encoded in T.

Given an XML string w, we let *content*(w) be the concatenation of all *character data* of w, taken in w's document order [3]. To make distinction between element types in DTD and element tags in document, we use *tag* to denote an element tag and *element* to denote an element type. We let *root*(w) denote the *root element* in w and *elements*(w) denote the set of all elements in w. For a tag x in w we let *element*(x) be the element of x in $\mathcal{T}$. For an element $a \in \mathcal{T}$ we employ XML syntax to denote *start tag* by `<a>` and *end tag* by `</a>`. We can now formalize the notion of potential validity of an XML document w w.r.t. DTD T.

Definition 2. *Let w be an XML string with elements(w) from DTD $T = \langle \Gamma, \mathcal{T} \rangle$. The set of extension strings of w with respect to the set of elements $\mathcal{T}$, denoted $Ext(w, \mathcal{T})$, is defined recursively as follows:*

1. *$w \in Ext(w, \mathcal{T})$;*
2. *a string $w^* \in Ext(w, \mathcal{T})$ if there exists an element $\alpha \in \mathcal{T}$ and there exist three strings w_1, w_2, and w_3 such that*
   - *(a) $w_1 w_2 w_3 \in Ext(w, \mathcal{T})$;*
   - *(b) w^* can be written as: $w^* = w_1{<}\alpha{>}w_2{<}/\alpha{>}w_3$ and*
   - *(c) w^* is a well-formed XML document.*

Intuitively, $Ext(w, \mathcal{T})$ is the set of all possible (well-formed) extensions of the XML string w obtained by tagging with elements of $\mathcal{T}$. Then, we can say that w is potentially valid if at least one of its extensions is a valid XML document. We formalize this after introducing some more notations.

Given a string C, and a DTD T, we let $\mathcal{D}(T, C, r)$ denote the set of all strings which represent well-formed XML encodings of C, valid with respect to the DTD T, whose root element is $r \in \mathcal{T}$.

Definition 3. *Let $T = \langle \Gamma, \mathcal{T} \rangle$ be a DTD and $r \in \mathcal{T}$. An XML string w with content$(w) = C$ and root$(w) = r$ is called* **potentially valid** *with respect to T and r if $(\exists w^* \in Ext(w, \mathcal{T}))(w^* \in \mathcal{D}(T, C, r))$.*

[2]Potential validity is affected by the structure of the DTD described in the element type declarations (one per element type). We need not consider attribute declarations: their presence or absence does not affect our consideration of the problem presented in this paper in any way.

Let now $\mathcal{D}^*(T, C, r)$ denote the set of all potentially valid XML documents w.r.t. T, with content C and root r.

Definition 4. *A markup inserted in a string $w \in \mathcal{D}^*(T, C, r)$ at given positions that produces a new string $w^* \in \mathcal{D}^*(T, C, r)$ is called* **potentially valid markup**.

Example 2. *Consider again the DTD T in figure 2 and the XML documents instances in Example 1. So $\mathcal{T} = \{r, a, b, c, d, e, f\}$. Consider the root element r and let $C =$* `"A quick brown fox jumps over a lazy dog"`*. Let*

```
w = "<r><a><b>A quick brown fox</b><c> jumps
over a lazy</c> dog<e></e></a></r>"
s = "<r><a><b>A quick brown fox</b><e></e><c> jumps
over a lazy</c> dog</a></r>"
```

Then $w \in \mathcal{D}^*(T, C, r)$ because the string

```
w'= "<r><a><b><d>A quick brown fox</d></b><c> jumps
over a lazy</c><d> dog<e></e></d></a></r>"
```

is in $D(T, C, r)$ and $w' \in Ext(w, \mathcal{T})$.
We have $s \notin \mathcal{D}^*(T, C, r)$ because we won't get the order b, e, and c of elements contained by element a, no matter what further markup we introduce in s.

2.2 A Grammar for checking potential validity

We define the problem of checking potential validity of XML documents as follows:

> **Problem PV:** Given a DTD $T = \langle \Gamma, \mathcal{T} \rangle$, and an XML string w, *content*$(w) = C$, marked up by the elements of $\mathcal{T}$ with the root $\rho \in \mathcal{T}$, output "yes" if $w \in D^*(T, C, \rho)$ and "no" otherwise.

In this section we show that given a DTD T, and a root element ρ, the set of all XML strings w for which $PV(T, \rho, w) =$ "yes" is context-free. We do this by constructing an extended context-free grammar (ECFG) [3] that recognizes XML documents with root ρ valid w.r.t. T, and modifying it to recognize *potentially valid markup*.

The ECFG for checking validity

Given a DTD $T = \langle \Gamma, \mathcal{T} \rangle$ and an element $\rho \in \mathcal{T}$, an extended context-free grammar for recognizing strings representing XML documents valid w.r.t. T with root ρ can be constructed in a fairly straightforward manner. Let $G_{T,\rho} = (N, \Sigma, R, S)$ be an extended context free grammar, where N is the set of nonterminals, Σ is the set of terminals, R is the set of grammar rules, and S is the start symbol. The set

[3]Extended context-free grammars (ECFGs) enhance the syntax of context-free grammars by allowing regular expressions on the right-hand sides of productions. Languages recognized by ECFGs are context-free.

N of nonterminals contains a nonterminal, $PCDATA$, corresponding to a $\#PCDATA$ keyword in the Element Type Definition. In addition, for each element $A \in T$, there are two corresponding nonterminals, $A, \hat{A} \in N$:

$$N = \{S, PCDATA\} \cup \{A, \hat{A} | A \in \mathcal{T}\}$$

Σ contains a terminal σ, corresponding to a string of non-markup characters of length at least one. For each element $A \in \mathcal{T}$ there are two corresponding terminals, one for its *start tag* and the other – for its *end tag*:

$$\Sigma = \{\sigma\} \cup \{< A >, < /A > | A \in \mathcal{T}\}$$

The set of rules, R, contains the rules $S \rightarrow \rho$ and $PCDATA \rightarrow \sigma \mid \epsilon$ (here, ϵ denotes an empty string). For each $A \in \mathcal{T}$, with `<!ELEMENT A A_content>` as its Element Type Definition in T we add two rules into R: $A \rightarrow < A > \hat{A} < /A >$, and $\hat{A} \rightarrow A\_content$:

$$R = \{S \rightarrow \rho, PCDATA \rightarrow \sigma \mid \epsilon\} \cup \\ \{A \rightarrow < A > \hat{A} < /A >,\ \hat{A} \rightarrow A\_content | A \in T\}.$$

Example 3. *For the DTD T in figure 2, let $\rho = a$. Then $G_{T,a} = (N, \Sigma, R, S)$ where (for clarity, we use upper-case letters for nonterminals):*

$$N = \{S, A, \hat{A}, B, \hat{B}, C, \hat{C}, G, \hat{G}, D, \hat{D}, E, \hat{E}, F, \hat{F}\}$$

$$\Sigma = \{\sigma, < a >, < /a >, < b >, < /b >, < c >, < /c >, \\ < g >, < /g >, < d >, < /d >, < e >, < /e >, < f >, \\ < /f >\}$$

$$\begin{aligned} R = \{&S \rightarrow A, PCDATA \rightarrow \sigma \mid \epsilon, \\ &A \rightarrow < a > \hat{A} < /a >, \\ &\hat{A} \rightarrow B?\ (C|G)\ D, \\ &B \rightarrow < b > \hat{B} < /b >, \\ &\hat{B} \rightarrow (D|F), \\ &C \rightarrow < c > \hat{C} < /c >, \\ &\hat{C} \rightarrow PCDATA, \\ &G \rightarrow < g > \hat{C} < /g >, \\ &\hat{G} \rightarrow FDF, \\ &D \rightarrow < d > \hat{D} < /d >, \\ &\hat{D} \rightarrow (PCDATA \mid E)*, \\ &E \rightarrow < e > \hat{E} < /e >, \\ &\hat{E} \rightarrow \epsilon, \\ &F \rightarrow < f > \hat{F} < /f >, \\ &\hat{F} \rightarrow PCDATA\} \end{aligned}$$

For the purpose of checking validity, we need to convert XML documents into strings recognized by the ECFGs described above. Given a string C with no XML encoding in it, we set $\delta(C) = \sigma$. Let $w = w_1 < a > w_2 < /a > w_3$ and let $\delta(w_1) = d_1, \delta(w_2) = d_2$ and $\delta(w_3) = d_3$. Then $\delta(w) = d_1 < a > d_2 < /a > d_3$. This procedure results in replacing all consecutive character data in the input XML string with a single σ terminal, while preserving the XML markup structure. Similarly we define an operator χ such that $\chi(C) = \sigma$ and $\chi(w_1 < a > w_2 < /a > w_3) = \chi(w_1)A\chi(w_3)$. For example,

$\delta($`"<a><b>A quick brown fox</b><c> jumps over a lazy</c><d> dog<e></e></d></a>"`$) =$ `"<a><b>`σ`</b><c>`σ`</c><d>`σ`<e></e></d></a>"`.
$\chi($`"<b>A quick brown fox</b><c> jumps over a lazy</c><d> dog<e></e></d>"`$) =$ `"BCD"`.

We can now state formally that ECFGs constructed as described above recognize valid XML:

Proposition 1. *An XML string w encoded using elements from the DTD T with root ρ is valid w.r.t. T **iff** $\delta(w) \in L(G_{T,\rho})$.*

ECFG for potential validation

The grammars $G_{T,root}$ described above are useful for validating XML documents as soon as the markup process is finished. In order to accept (check for potential validity) intermediate stages of the XML document during the encoding process the grammar $G_{T,\rho}$ needs to be enhanced.

To represent all potentially valid XML documents (actually, document structures) for a given DTD T and root $\rho \in T$, we define an extended grammar $G'_{T,\rho} = (N, \Sigma, R', S)$. In this grammar, N, Σ and S are the same as in $G_{T,\rho}$. The new set of rules R' is defined as follows:

$$R' = R \cup \{A \rightarrow \hat{A} | A \in \mathcal{T}\}.$$

The intuition behind this extension of $G_{T,\rho}$ is as follows. Given a valid (w.r.t. DTD T) XML document w with root ρ, we can construct potentially valid XML documents from it by selecting one or more tags in w and removing them to produce document w^*. In the derivation of $\delta(w)$ in $G_{T,\rho}$ (see Figure 4), the "open tag" and "close tag" terminals are derived via the rules of the form $A \rightarrow < A > \hat{A} < /A >$. By inserting the rule $A \rightarrow \hat{A}$ for each XML element A, we are allowing the grammar $G'_{T,\rho}$ to mimic the derivation of $\delta(w)$, and convert it into the derivation of $\delta(w^*)$ by electing not to derive the XML tag terminals where needed.

Example 4. *For the DTD T in figure 2 and $\rho = a$, we construct the following grammar $G'_{T,a} = (N, \Sigma, R', S)$:*

$$N = \{S, A, \hat{A}, B, \hat{B}, C, \hat{C}, G, \hat{G}, D, \hat{D}, E, \hat{E}, F, \hat{F}\}$$

$$\Sigma = \{\sigma, < a >, < /a >, < b >, < /b >, < c >, < /c >, \\ < g >, < /g >, < d >, < /d >, < e >, < /e >, < f >, \\ < /f >\}$$

$$\begin{aligned} R' = \{&S \rightarrow A, PCDATA \rightarrow \sigma \mid \epsilon, \\ &A \rightarrow < a > \hat{A} < /a >, \\ &A \rightarrow \hat{A}, \\ &\hat{A} \rightarrow B?\ (C|G)\ D, \\ &B \rightarrow < b > \hat{B} < /b >, \\ &B \rightarrow \hat{B}, \\ &\hat{B} \rightarrow (D|F), \\ &C \rightarrow < c > \hat{C} < /c >, \\ &C \rightarrow \hat{C}, \\ &\hat{C} \rightarrow PCDATA, \end{aligned}$$

$G \rightarrow < g > \hat{G} < /g >$,
$G \rightarrow \hat{G}$,
$\hat{G} \rightarrow FDF$,
$D \rightarrow < d > \hat{D} < /d >$,
$D \rightarrow \hat{D}$,
$\hat{D} \rightarrow (PCDATA \mid E)*$,
$E \rightarrow < e > \hat{E} < /e >$,
$\hat{E} \rightarrow \epsilon$,
$E \rightarrow \hat{E}$,
$F \rightarrow < f > \hat{F} < /f >$,
$F \rightarrow \hat{F}$,
$\hat{F} \rightarrow PCDATA\}$

We note that the set of rules R' for $G'_{T,\rho}$ can be trivially simplified, by substituting rules of the form $A \rightarrow A\_content$ for $A \rightarrow \hat{A}$ and rules of the form $A \rightarrow < A > A\_content < /A >$ for $A \rightarrow < A > \hat{A} < /A >$ and thus, eliminating the $\hat{A}$ non-terminals. For the illustrative purposes and proofs, however, we are preserving R' as described above. We are now ready to state the main result of this section.

THEOREM 1. *An XML string w^* encoded using elements from the DTD T with root element ρ is **potentially valid** w.r.t. T **iff** $\delta(w^*) \in L(G'_{T,\rho})$.*

PROOF. (*sketch*) The general idea behind the proof is as follows. Let w^* be potentially valid. Then the set of its extensions contains a valid XML document w. Then, $\delta(w)$ has a derivation in the grammar $G_{T,\rho}$. We make a list of all XML elements that were added to w^* in order to convert it to w. For each such element a, we find an application of the rule $A \rightarrow < a > \hat{A} < /a >$ in the derivation of w in $G_{T,\rho}$ and replace it with the application of the rule $A \rightarrow \hat{A}$. The resulting construct will be a valid derivation in $G'_{T,\rho}$ and the string derived will be $\delta(w^*)$.

Going in the other direction, suppose $\delta(w^*)$ has a derivation in $G'_{T,\rho}$. We find all applications of the rules of the form $X \rightarrow \hat{X}$ in the derivation, and replace them with the applications of $X \rightarrow < X > \hat{X} < /X >$. The resulting derivation will also be a valid derivation in $G_{T,\rho}$. This means that some extension w of our string w^* is a valid XML document and therefore w^* is potentially valid. □

Figure 4 shows the derivation of the XML string

```
<a><b>A quick brown fox</b><c> jumps over
a lazy</c> dog<e></e></a>
```

in the grammar $G'_{T,a}$ (for illustrative purposes we did not replace content with σs). This derivation mimics the structure of the DOM tree shown in Figure 3. The XML elements that are present in the string above are derived using the $X \rightarrow < x > \hat{X} < /x >$ rules, whereas the XML elements that were added to the document to make it valid are derived using the $X \rightarrow \hat{X}$.

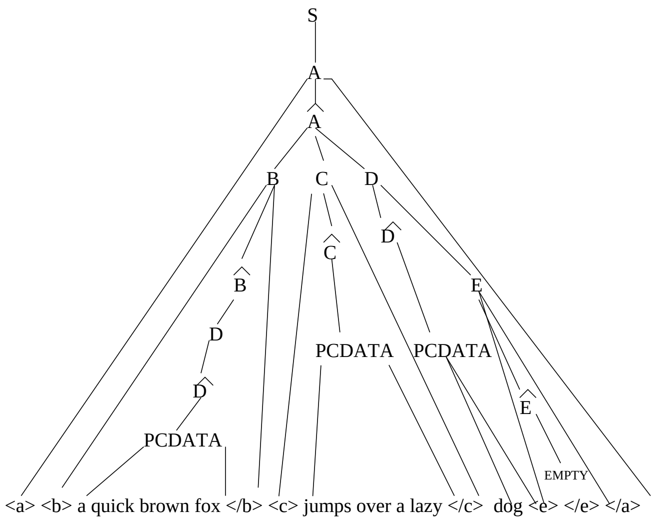

Figure 4: $G'_{T,a}$ derivation of a potentially valid XML document.

3. ALGORITHMS

It has been shown in the previous section that the problem PV is equivalent to deciding whether or not a given input string (representing an XML document instance) belongs to a language recognized by a specific ECFG. Extended context free grammars (or regular right part grammars) were introduced a long time ago [18, 11] as a more flexible and intuitive way to characterize some context-free languages. Since these grammars were defined, a lot of parsers and recognizers have been proposed ([12, 5, 15, 10, 13, 16, 4], just to name few). Some of them are based on translating the ECFG into CFG, others are parsing ECFGs directly. It is also important to note that some parsers work not for general ECFGs but for certain restricted cases.
Most of the grammars in the family of grammars $G'_{T,\rho}$ we construct in section 2 for solving the problem PV are highly ambiguous,which precludes the use of well-known linear-time parsers that require unambiguity of grammars. In general, we can always use an unrestricted CFG parsing algorithm to recognize potential validity.In this paper we choose a well-known Earley's algorithm [7, 8] for general CFGs for this purpose.

At the same time, as we show in Section 4, the performance of Earley's algorithm on large XML documents suffers when the DTDs generate highly ambiguous grammars, which makes this approach not scalable.

As it turns out, however, the family $G'_{T,\rho}$ of grammars for recognizing potential validity, possesses a number of properties, that allow us to develop a fast, linear-time parsing algorithm.

Given as input the triple (T, ρ, w), where T is a DTD, w is an XML document instance and ρ is an element in T considered to be the root element of w, this algorithm works in in two stages: (i) a recognizer is constructed based on T; (ii) the recognizer decides $w \in D^*(T, C, \rho)$.

In the applications of potential validity checking that we are considering, such as document-centric XML editors, it is likely that the two stages of the check will be performed at different times. The first stage depends solely on the structure of the DTD and can be performed when the DTD is loaded. As the editorial process proceeds, the structure of the XML document will change, and each change can result in a potential validity check. Each such check can take advantage of the stage (i) preprocessing, and proceed directly to stage (ii). Because of these considerations, we separate our description and evaluation of algorithms for both stages, concentrating more on stage (ii), the actual potential validity check for a specific instance.

From Section 2 we can conclude that the size of the XML fragment in bytes may not be the right measure of the size of input for our parsing algorithms. Indeed, the transformation δ of the XML file into the string recognized by $G'_{T,\rho}$ grammars factors the length of content strings out of the consideration. Similarly, from the point of view of potential validity check, the length of the XML element names is irrelevant[4]. Instead, we use the *number of tokens* measure defined below.

Definition 5. *Let w be an XML document with the DOM tree D that contains α element nodes and β #PCDATA nodes. Then the* number of tokens *in w, denoted $size(w)$ is defined as*

$$size(w) = 2 \cdot \alpha + \beta.$$

Basically, the number of tokens measure counts how many distinct tokens are found in the XML document. We count twice the number of element nodes to take into account both *start* and *end* tags; for the content, we count as one any string between two tags. For each algorithm to be presented we give time bounds for deciding problem PV on a given XML document instance w w.r.t $size(w)$. In Section 4 we show that number of tokens is, indeed a more reasonable measure of the size of XML document then its size in bytes.

We also observe that to decide PV for a given XML document instance w, a recognizer doesn't need the string w verbatim. Rather, we consider the tokenized version $\delta(w) = X_1, X_2, \ldots, X_n$, $n = size(w)$, (as in section 2). For each token X_i, we let $type(X_i)$ denote its type: start tag, end tag, content, and $element(X_i)$ denote the DTD element represented by the tag (e.g., $element(< a >) = element(< /a >) = a$; we also assume that $element(\sigma) = \#PCDATA$).

3.1 ECFG recognizer for PV

In most cases, the grammars we use for solving the PV problems turn out to be very ambiguous, so traditional parsers (LL, LR) either don't work for these grammars or require preprocessing in order to eliminate the ambiguities. In [7, 8] Earley proposed an algorithm for parsing any CFG, with no restriction. Although the algorithm is given for a CFG, an extension for the case when "star closure" ("*") operator is used in the right hand side of a production is given in [7]. Further extensions for the use of "at most one" ("?") operator and "positive closure" ("+") are straightforward. These extensions make Earley's algorithm appropriate for ECFGs as well. However, there is one more detail to be taken care of: Earley's recognizer uses a grammar having rules (productions) with no choice operator ("|"). Each production that contains choices has to be split up into equivalent productions with no choice operator.
Thus, to use Earley's algorithm, the set of rules in grammar $G'_{T,\rho}$ (described in section 2) has to be modified so that no rule contains the operator "|" (these changes are likely to introduce new non-terminals). Rules satisfying the above constraint can be generated in a straightforward manner for each Element Type Declaration of a given DTD.

For instance, in example 3, the rules for non-terminals $PCDATA$ and $\hat{D}$ have to be rewritten as: $PCDATA \to \sigma$, $PCDATA \to \epsilon$, $\hat{D} \to PCDATA*$, $\hat{D} \to E*$.

The next result follows from [7]:

Proposition 2. *Earley's algorithm, when used to solve problem PV for a DTD T and an XML document w has $O(size(w)^3)$ time bound.*

Even though Earley's algorithm is relatively fast in terms of the number of tokens in the input document, the constant involved might be very large in some situations. For DTD elements containing star closure or positive closure in their definition, there are many possible productions to follow. We show next that these cases can actually be reduced to a reachability problem and so a recognizer algorithm time performances can be greatly improved.

3.2 Fast Algorithm for PV

To construct a fast linear recognizer for the class $G'_{T,\rho}$ of grammars, we need to study the properties of these grammars in more detail.
Consider a DTD $T = \langle \Gamma, \mathcal{T} \rangle$. We want to decide whether or not an XML document instance, w, represented as a sequence $\delta(w) = X_1, X_2, ..., X_n$ of tokens is recognized by the grammar $G'_{T,\rho}$ where ρ is the root element. We start by defining a reachability graph for the DTD elements.

Definition 6. *The reachability graph of T is the directed graph $R_T = (V, E)$, $V = \mathcal{T} \cup \{ANY, EMPTY, \#PCDATA\}$, $E = \{(t_1, t_2) : t_1, t_2 \in \mathcal{T}, t_2$ appears in the Element Type Declaration of $t_1\} \cup \{(\#PCDATA, EMPTY)\} \cup \{(ANY, t) | t \in V\}$.*

Example 5. *Figure 5 shows the reachability graph for the DTD from Figure 1.*

The reachability graph of T tells us whether or not a given element $t_2 \in \mathcal{T}$ may be found in the content of another element t_1. To check it, it is sufficient to determine if t_2 is reachable from t_1 in R_T (we also assume that EMPTY is reachable from #PCDATA, because content strings matching #PCDATA can be empty). If we measure the size of the

[4] In addition to that, XML document may contain other information that is irrelevant for potential validation, such as attribute values.

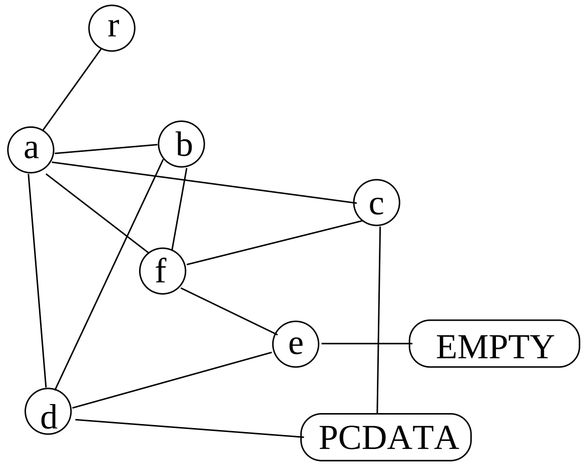

Figure 5: Reachability graph for the DTD from Figure 1.

DTD in terms of the number of tokens needed to represent its element type declarations ($size(T) = \sum_r \in \Gamma size(r)$, where $size(r)$ is the number of tokens in the righthand side of r plus one), then the graph R_T can be constructed in the time $O(size(T))$: each rule needs to be scanned only once. With R_T constructed, the reachability relation between its nodes can be precomputed in a form of a *lookup table*, L_T, such that $L_T(t_1, t_2) = true$ if there exists a path from t_1 to t_2 in R_T, and $L_T(t_1, t_2) = false$ otherwise (computing the lookup table is basically an *all pairs shortest path* problem for R_T which can be done in $O(|V|^3)$ time).

In practice, it might be the case that a DTD is constructed with not much care, or modifications to the DTD lead to cases when some elements cannot be used in any real (valid) XML document instance (they lead to infinite loops in deriving their content). An element t in T is called *usable* if there exists an XML document instance valid w.r.t. T that contains markup corresponding to t. The algorithm described below assumes that **all** elements in the input DTD are usable. It is known that given a CFG, the set of its usable nonterminals can be efficiently constructed [9]. In order to ensure that the potential validity checking algorithm receives only DTD grammars with all usable elements, we will use the usability checking algorithm [9] to prune the unusable elements from the DTD[5]. The following important property of the usable DTD elements is important for us.

Lemma 1. *Let $T = \langle \Gamma, \mathcal{T} \rangle$ be a DTD and $a \in \mathcal{T}$ be a usable element of T for some root element $\rho \in \mathcal{T}$. Let A be the nonterminal in $G'_{T,\rho}$ that corresponds to a. Then, $A \Rightarrow^*_{G'_{T,\rho}} \epsilon$.*

[5]The set of usable elements is constructed via a fixed-point procedure that starts by marking all left-hand sides of terminal rules as usable, and then marking as usable all left-hand sides of the rules whose righthand sides are usable. Once the set of usable non-terminals is determined, unusable non-terminals are removed from the grammar together with all the rules where they appear in the righthand side[9].

THEOREM 2. *Let $T = \langle \Gamma, \mathcal{T} \rangle$ be a DTD and let **all** elements of T be* usable *for some root element $\rho \in \mathcal{T}$. Then $(\forall a, b \in \mathcal{T})(A \Rightarrow^*_{G'_{T,\rho}} B$ **iff** b is* reachable *from a in R_T) (where A and B are nonterminals of $G'_{T,\rho}$ corresponding to the elements a and b of $\mathcal{T}$).*

Constructing a Simplified Grammar for Potential Validity.

From now on we consider only the DTDs where all XML elements are usable. In Section 2 we have described the construction of an Extended CFG $G'_{T,\rho}$ for recognizing potentially valid documents for a given DTD and a root element. While this grammar can be processed by general CFG parsers, the complex structure of the right-hand sides of the grammar rules, which can contain almost arbitrary regular expressions[6]. In this section, we show how $G'_{T,\rho}$ can be converted into a new, equivalent grammar $G''_{T,\rho}$ that contains only the rules whose right-hand sides can be processed efficiently. We start by showing that we can eliminate all ? operators from consideration.

THEOREM 3. *Let $T = \langle \Gamma, \mathcal{T} \rangle$ be a DTD. Let $T' = \langle \Gamma', \mathcal{T} \rangle$ be a DTD obtained from T by removing all occurrences of the ? operator from Γ. Then, for any root element $\rho \in \mathcal{T}$ and for any XML document w, $w \in D^*(T, \rho, content(w))$ **iff** $w \in D^*(T', \rho, content(w))$.*

Next, we consider two types of the DTD rules involving the $*$ and the $+$ operators. We show that these rules can be processed efficiently.

DEFINITION 7. *A* choice-star element *of a DTD is an element x that has one of the following types of Element Type Definitions:*
1. $<!ELEMENT\ x\ (\#PCDATA|y_1|y_2|\ldots|y_k)* >$, *or*
2. $<!ELEMENT\ x\ (y_1|y_2|\ldots|y_k)* >$, *or*
3. $<!ELEMENT\ x\ (y_1|y_2|\ldots|y_k)+ >, \quad k \geq 1$

DEFINITION 8. *A* sequence-star element *of a DTD is an element x that has one of the following types Element Type Definitions:*
1. $<!ELEMENT\ x\ (y_1, y_2, \ldots, y_k)* >$, *or*
2. $<!ELEMENT\ x\ (y_1, y_2, \ldots, y_k)+ >, \quad k \geq 1$

DEFINITION 9. *An element x of a DTD has a* terminal *element type declaration if it is of one of the following forms:*
1. $<!ELEMENT\ x\ (\#PCDATA) >$, *or*
2. $<!ELEMENT\ x\ EMPTY >$.

We notice that the class of DTDs that have only choice-star elements is an important subset of all DTDs: it includes all *mixed-content* DTDs. The two theorems below show that the potential validity problem for DTDs with only choice-star and/or sequence-star elements can be solved by simply traversing the reachability graphs R_T of such DTDs.

[6]The only restrictions on the syntax of regular expressions found on the right-hand sides of the DTD rules concern combining together XML elements and #PCDATA.

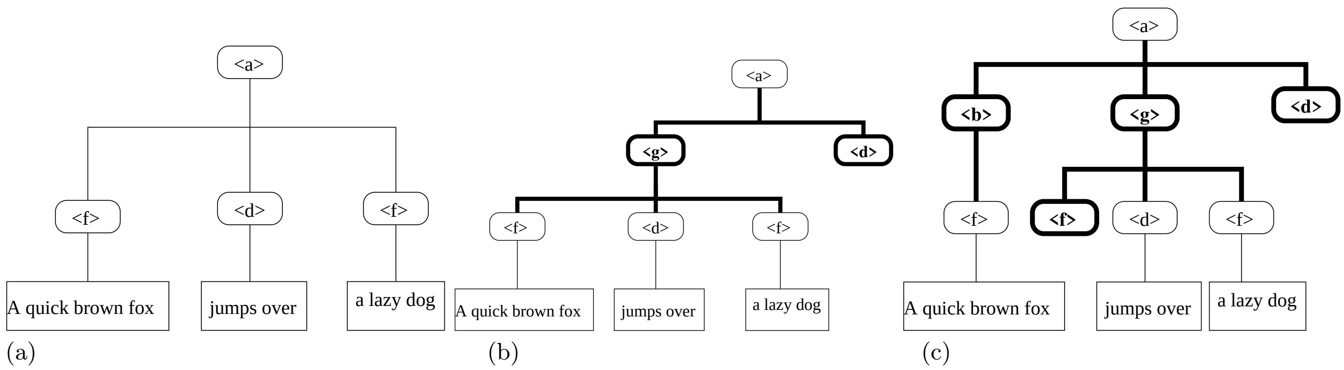

Figure 6: Using Theorem 5 to construct potential validity proofs.

THEOREM 4. *Let T be a DTD where all elements are usable and have either choice-star, sequence-star or terminal element declarations. Then, an XML document instance w = <a>w' is potentially valid w.r.t. T and a as a root ($w \in D^*(T, a, content(w))$)* ***iff*** *for any token X in $\delta(w')$, $element(X)$ is reachable from a in R_T.*

PROOF. (*sketch*) First we note that the sequence-star declarations 1. and 2. are equivalent to the choice-star declarations 2. and 3. respectively. Without loss of generality, we now assume that all elements in T have either choice-star or terminal declarations.

Let $\delta(w') = X_1, X_2, \ldots, X_n$ be the tokenized version of w'. If $w \in D^*(T, a, content(w))$, then $< a > \delta(w') < /a >$ is derived from a in $G'_{T,a}$ and therefore there is a path from a to each X_i, $i = 1 \ldots n$. This proves the *only if* part of the theorem. Conversely, let all X_i, $i = 1 \ldots n$ be reachable from a and let a be a choice-star element with the element type declaration $<!ELEMENT\ x\ (y_1|y_2|\ldots|y_k)* >$ (other cases are considered in a similar manner). We construct a sequence of elements $z_1, \ldots, z_n$, $z_i \in \{y_1, \ldots, y_k\}$ such that for all $1 \le i \le n$, X_i is reachable from z_i in R_T. We then apply the production rule for $\hat{A}$ n times to produce the sequence $Z_1 Z_2 \ldots Z_n$ of non-terminals, and derive each X_i separately from Z_i. ☐

Theorem 4 suggests a fast way of determining potential validity in choice-star and sequence-star DTDs. However, we cannot reduce a simple choice, or a simple sequence element type declaration to simply checking reachability on the graph R_T (although the only if part of Theorem 4 will obviously remain true). First, we specify how DTD element type declarations of the form $<!ELEMENT\ x\ (y_1, y_2, \ldots, y_k) >$ (otherwise called *sequence declarations*), can also be processed efficiently. Our approach is based on the theorem below.

THEOREM 5. *Let T be a DTD where all elements are usable. Let $w =< a > w' < /a >$ be an XML fragment, and let $\delta(w') = E_1 \ \ldots \ E_n$. Let $\hat{A} \to A_1\ A_2\ \ldots\ A_k$ be the production rule for $\hat{A}$ in $G'_{T,a}$ and let $E_1, \ldots, E_s$, $s \le n$ be the longest prefix of $\delta(w')$ derivable from $\hat{A}_1$. Then $w \in D^*(T, a, content(w))$* ***iff*** *$A_2\ \ldots\ A_k \Rightarrow^* E_{s+1}\ \ldots\ E_n$.*

PROOF. (*Sketch*) The proof is based on the following observation. Suppose A_2 *can only derive* E_{s+1} (as part of the sequence $E_{s+1}\ \ldots\ E_{s+l}$) if it is derived together with the sequence $E_r\ \ldots E_{s-1}\ E_s$, $1 \le r \le s$. Then, we can construct the following derivation: $A_1\ A_2\ \ldots\ A_k \Rightarrow^*_{G'_{T,a}}$ $E_1 \ldots\ E_r\ \ldots\ E_s\ E_r\ \ldots E_s\ E_{s+1}\ \ldots\ E_{s+l}\ A_3\ \ldots\ A_k$: where $E_1 \ldots\ E_r\ \ldots\ E_s$ is derived from A_1 and $E_r\ \ldots E_s\ E_{s+1}\ \ldots\ E_{s+l}$ is derived from A_2. But because **all** elements of the DTD T are usable, and by Lemma 3.2, for each $r \le i \le s$ element $E_i \Rightarrow^*_{G'_{T,a}} \epsilon$, i.e., we can continue the derivation as follows: $A_1\ A_2\ \ldots\ A_k \Rightarrow^*_{G'_{T,a}} E_1 \ldots\ E_r\ \ldots\ E_s\ E_r\ \ldots E_s\ E_{s+1}$ $\ldots\ E_{s+l}\ A_3\ \ldots\ A_k \Rightarrow^*_{G'_{T,a}} E_1 \ldots\ E_r\ \ldots\ E_s\ \epsilon\ \ldots\ \epsilon\ E_{s+1}$ $\ldots\ E_{s+l}\ A_3\ \ldots\ A_k = E_1 \ldots\ E_r\ \ldots\ E_s\ E_{s+1}\ \ldots\ E_{s+l}$ $A_3\ \ldots\ A_k$.

This procedure is used inductively to derive $E_{s+l+1}\ \ldots\ E_n$ from $A_3\ \ldots\ A_k$. ☐

To summarize, Theorem 3 allows us to eliminate all ? operators from the consideration. Theorem 4 reduces the problem of parsing star-choice and star-sequence rules to the problem of reachability on the dependency graph R_T of the DTD. Theorem 5 suggests a greedy algorithm for bottom up parsing of the sequence DTD declarations. We are now ready to describe the construction of the grammar $G''_{T,\rho}$ equivalent to $G'_{T,\rho}$ that only contains the choice-star, sequence-star and sequence rules for the DTD nonterminals.

Basically $G''_{T,\rho}$ is constructed from $G'_{T,\rho}$ by replacing productions of type $\hat{A} \to A\_content$ with sets of star-choice, star-sequence or sequence productions for $\hat{A}$, and, possibly, introducing new nonterminals (with corresponding productions) in the process. We build the new set of productions R'' for $G''_{T,\rho}$ iteratively as follows.

Procedure PV-Transform-ECFG.

- Base Case: R''_0 is constructed from R' by removing all occurrences of the ? operator from right-hand sides of all rules in R'. All rules in R''_0 are *unmarked*.
- Inductive Step: Suppose R''_k is constructed. Consider an unmarked rule $r : G \to W \in R''_k$.
  - If $W = PCDATA$, $W = EMPTY$ or $W = ANY$

or $W = (PCDATA|X_1| \ldots |X_n)*$, $W = X$, where X is a single non-terminal, or if W contains terminals, then set $R''_{k+1} = R''_k$, and *mark* r as visited in R''_{k+1}.

– If $W = (W')*$ or $W = (W')+$, let $elements(W') = \{X_1, \ldots, X_m\}$ be the set of all non-terminals, corresponding to the DTD elements found inside W'. Then, $R''_{k+1} = R''_k - \{r\} \cup \{G \to (X_1 \ \ldots \ X_m)*\}$ and the new production is *marked* as visited in R''_{k+1}.

– If $W = W_1 \mid W_2 \mid \ldots \mid W_m$, then, $R''_{k+1} = R''_k - \{r\} \cup \{G \to W_i | 1 \le i \le m\}$ and the new productions are **not** marked in R''_{k+1}.

– If $W = W_1 \ W_2 \ \ldots \ W_m$, then, letting $Y_1, \ldots Y_m$ be new non-terminal symbols $R''_{k+1} = R''_k - \{r\} \cup \{G \to Y_1 \ Y_2 \ \ldots \ Y_m\} \cup \{Y_i \to W_i | 1 \le i \le m\}$, where the rule $G \to Y_1 \ Y_2 \ \ldots \ Y_m$ is *marked* as visited, while all other new productions in R''_{k+1} are **not** marked.

– If R''_k has no unmarked productions, $R'' = R''_k$.

- Termination: If R''_k has no unmarked productions, $R'' = R''_k$.

THEOREM 6. *Let $T = \langle \Gamma, \mathcal{T} \rangle$ be DTD, ρ be a root element from and $G'_{T,\rho}$ be the ECFG checking potential validity of XML documents in T with root ρ. Then, the following holds:*

1. **Procedure PV-Transform-ECFG** *will terminate on input $G'_{T,\rho}$.*
2. *The output grammar $G''_{T,\rho}$ of* **Procedure PV-Transform-ECFG** *is equivalent to $G'_{T,\rho}$.*
3. *The size of grammar $G''_{T,\rho}$ measured as the number of terminals and nonterminals in all the rules $G''_{T,\rho}$ is bounded by some polynomial function of the size of $G'_{T,\rho}$.*

EXAMPLE 6. *Consider the grammar $G'_{T,a}$ constructed in Example 4. Consider the rule*
$\hat{A} \to B?\ (C|G)\ D$,
from it. ***Procedure PV-Transform-ECFG****, will transform the rule as follows:*
• *First, the ? operator will be removed from it, producing the rule $\hat{A} \to B\ (C|G)\ D$,.*
• *Second, this rule will be replaced by the rule $\hat{A} \to Y_1\ Y_2\ Y_3$, where Y_1, Y_2 and Y_3 are new non-terminals, and rules $Y_1 \to B$, $Y_2 \to C|G$ and $Y_3 \to D$ will be introduced.*
• *Third, the rule $Y_2 \to C|G$ will be replaced with two rules $Y_2 \to C$ and $Y_2 \to G$.*

This example shows that **Procedure PV-Transform-ECFG** is not optimal: productions of the form $Y_1 \to B$ generated by this procedure are redundant. A careful implementation of this procedure, however, can avoid creation of such redundant rules by checking the atomicity of the right-hand side of each rule before generating it.

We are now ready to describe the algorithm for parsing the $G''_{T,\rho}$ family of grammars. Before we do that, we illustrate the intuition behind the derivations that this algorithm will attempt to construct on the following example.

EXAMPLE 7. *Consider again our DTD from Figure 1 and consider the following XML fragment w (its DOM tree is depicted in Figure 6.(a)):*

```
<a><f>A quick brown fox</f><d>jumps over</d>
<f>a lazy dog</f></a>
```

This fragment is potentially valid. An easy way to verify that is to notice that the sequence `f d f` is part of the righthand side of the DTD rule for the element `c`. Noticing also that element `b` is optional for the content of `a`, and that element `d` can consist of a single empty element `e`, the following valid XML encoding w' (see Figure 6.(b) can be constructed to verify the potential validity of the fragment above:

```
<a><c><f>A quick brown fox</f><d>jumps over</d>
<f>a lazy dog</f></c><d><e></e></d></a>
```

At the same time, ***Algorithm Fast-PV*** *proposed below will construct the derivation that will "shadow" the following valid XML extension w'' of w:*

```
<a><b><f>A quick brown fox</f></b><c><f></f><d>jumps
over</d> <f>a lazy dog</f></c><d><e></e></d></a>
```

The DOM tree of w'', shown in Figure 6.(c), suggests how w'' can be used to prove that w is potentially valid. From Example 6, we know that (a slightly improved version of the grammar) $G''_{T,a}$ will contain rules $\hat{A} \to BY_2D$ and $Y_2 \to G$. The grammar will attempt to derive the longest possible prefix of $< f > \sigma < /f >< d > \sigma < /d >< f > \sigma < /f >$ from B and will succeed in deriving $< f > \sigma < /f >$. It will then proceed to derive $< d > \sigma < /d >< f > \sigma < /f >$ from Y_2, or, in fact, from G. The production for G available in $G''_{T,a}$ will basically be $G \to FDF$, and the derivation of B has just stolen the $< f >$ component. To compensate for this steal, G will derive FDF, and then, the first F will derive ϵ, which DF will derive exactly $< d > \sigma < /d >< f > \sigma < /f >$. The DOM tree of w'', shows clearly how the steal of the element $< f >$ for the content of $< b >$ was compensated by inserting another element $< f >$ with empty content to help validate the content of $< g >$.

A DAG model for DTD

The grammar $G'_{T,r}$ defined in Section 2 gives the theoretical support for checking potential validity. We describe now a Directed Acyclic Graph (DAG) model for a DTD that allows us to efficiently implement a parser for $G'_{T,r}$. More specifically, for an XML string `<a>w</a>` we want to determine whether or not $\hat{A} \Rightarrow^*_{G'_{T,r}} \chi($`<a>w</a>`$)$.

The DTD's DAG model is composed on a DAG for each Element Type Declaration in the DTD (*element DAG*). An element DAG has a *root* node, the element whose the Element Type Declaration is represented by the element DAG, and two types of nodes: *simple element* nodes and *choice-star*

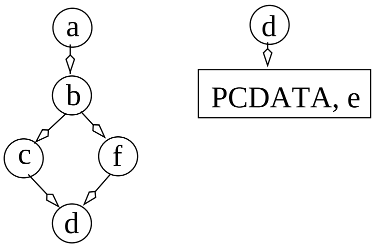

Figure 7: DAGs for Elements Type Declaration of elements *a* and *d* in the DTD in Figure 1

nodes. A simple element node corresponds to an element in the right-hand side of the Element Type Declaration for which no * or + operators are to be applied. A choice-star node corresponds to a set of elements (in the right-hand side of the Element Type Declaration) for which the same * or + operators are applied. Grouping elements in a choice-star node is to be done in a greedy manner, so that only the outmost * or + operators are taken into account in creating a choice-star node (therefore each *mixed content* declarations has two nodes: the root node and one choice-star node grouping all elements plus PCDATA in the mixed content declaration). Each choice-star node keeps a list of elements (*element-list*) grouped in the respective node. Once all nodes are created for an Element Type Declaration, the edges connecting these nodes are created similarly to the *parsing tree* [1] edges of a CFG: a pair of nodes (a, b) are connected by an edge iff b follows directly after a in a production rule. DAGs of two elements of the DTD in Figure 1 are given in Figure 7 (choice-star nodes are represented as boxes).

More formally, let's consider the (simplified version of) ECFG for the *content specification* of an element type declaration, as given in [3] (note that non terminals are represented between ' characters).

```
contentspec ::= 'EMPTY''ANY' | Mixed | children
children    ::= (choice | seq)('?' | '*' | '+')?
cp     ::= (Name | choice | seq) ('?' |'*' | '+')?
choice ::= '(' cp ( '|' cp )+ ')'
seq    ::= '(' cp (',' cp)* ')'
```

First, we mention that for declarations using `EMPTY` or `ANY`, the element DAG is straight forward: the root node and its only child, the `EMPTY` node or `ANY` node respectively. As mentioned before, *mixed content* declaration correspond as well to a DAG of two nodes.

In the following, we focus on declarations containing (possible nested) *choices* and/or *sequences* of element names. First, ? operators are not taken into account at all for processing and + and * operators are treated identically as *choice-star* operators.

Algorithm FastPV

We can now describe the **Algorithm FastPV** for checking potential validity. Given a DTD T, a root element ρ and an XML document w, we solve **Problem PV** as follows:

1. Usability Check: Determine the subset of elements in T that are *usable.*
2. DTD analysis: Construct the dependency graph R_T, the lookup table L_T representing reachability of usable elements in T and the DTD's DAG_T.
3. Input tokenization: Given w, construct $\delta(w)$.
4. FastPV: Run **Algorithm FastPV** on DAG_T, r, L_T and $\delta(w)$ as inputs.

```
FastPV(DAG_T, L_T, r, X_1, ..., X_n)
    if r ≠ element(X_1)
        return "reject"
    stack = new Stack()
    curRec = new ECRecognizer(ρ)
    foreach k = 2 ... n − 1
        if type(X_k) = end-tag
            curRec = stack.pop()
            continue
        answer = curRec.validate(X_k)
        if answer = "reject"
            return "reject"
        if type(X_k) = start-tag
            stack.push(curRec)
            curRec =
                new ECRecognizer(element(X_k))
    return "accept"
```

Figure 8: The FastPV algorithm

The pseudo-code for **Algorithm FastPV** is shown in Figure 8. **Algorithm FastPV** reads the input string $\delta(w)$ token by token; it starts by instantiating an element content recognizer (**ECRecognizer** object) for the root element and makes it the current element content recognizer. For simplicity, we consider L_T and DAG_T as global variables so they can be accessed for any ECRecognizer object. A stack is used to pile the recognizers as the recognizing process goes deeper in the input XML document structure. Each time a start tag token appears in the input, the current recognizer validates the token; if current recognizer validation fails, **FastPV** rejects the input; else, a new current recognizer object is created using **ECRecognizer** (for the element whose start tag is currently analyzed) and the old one is pushed in a stack. Each time an end tag occurs, the current recognizer is discarded and a new current recognizer is popped from the stack. Each time a σ symbol appears, the current recognizer performs a validation: if it rejects, then **FastPV** rejects. At the end of input, **FastPV** accepts (no intermediate element content recognizer has rejected any input symbol).

Element Content Recognizers

An element content recognizer **ECRecognizer** used in the **Algorithm FastPV** solves the following class of problems:

<u>**Problem ECPV:**</u> **Problem ECPV** is an instance of **Problem PV** on the following inputs: DTD T, an XML string $w =< a > w' < /a >$,

$\delta(w') = X_1, X_2, \ldots, X_n$ such that $\forall 1 \leq i < n$, $type(X_i) = \text{start-tag} \Rightarrow (type(X_{i+1}) = \text{end-tag} \wedge element(X_i) = element(X_{i+1}))$ and root element a.

Informally, the XML documents considered in **Problem ECPV** are of depth one (this is enforced by the requirement that each starting tag is immediately followed by its closing counterpart). The pseudocode for the content recognizer **ECRecognizer** is shown in Figure 9. The idea behind the use of **ECRecognizer** instances in the algorithm **FastPV** is as follows. Consider an XML fragment whose tokenized version is $< a > \sigma < b > \sigma < /b > \sigma < c > \sigma < /c > \sigma < /a >$. When parsing the tokens from left to right, we will first be able to establish the potential validity of the fragment $< b > \sigma < /b >$. Once it is established, we may replace this fragment with $< b >< /b >$, knowing that the content of b has already been derived. Similarly, we will do the same thing for the c element. Once we see token $< /a >$, our current fragment will look $< a > \sigma < b >< /b > \sigma < c >< /c > \sigma < /a >$. At this point the element content recognizer for a will attempt to solve **Problem ECPV** on this content. If successful, it will replace the input fragment with $< a >< /a >$ and pass the control to the recognizer at the upper level.

Each instance of **ECRecognizer** for an element x uses the element DAG, $DAG_T(x)$, to validate the element content. The validation proceeds in the greedy manner that follows the result of Theorem 5 to validate sequential rules. The alternate rules for the same token are processed in parallel. The result of the Theorem 4 is used to do quickly validate the star-choice and star-sequence productions using the element reachability information contained in the lookup table L_T.

Complexity of Algorithm FastPV

In general, the time it takes to solve an instance of **Problem PV** depends on the size (number of tokens) in the input XML document and on the properties of the input DTD. Let $n = size(w)$ be the number of tokens in $\delta(w)$, m be the number of XML elements defined in the input DTD, and let k be the number of tokens (occurrences of DTD elements in the left- and right-hand sides of the DTD rules). By Theorem 6 the size of the grammar $G''_{R,T}$ is $p(k)$ where p is some polynomial. Construction of the dependency graph and the lookup table for the DTD take $O(m^3)$ time, which cannot be more than $O(k^3)$.

The running time of the **Algorithm FastPV** is determined by the fact that it performs one call to the element content recognizer **ECRecognizer** per each input token. Each input token is considered inside exactly one instance of **ECRecognizer** — the instance generated for the parent XML element of the token. At the same time, the number of passes taken by that **ECRecognizer** over each token is bounded by an exponent on the branching factor in the DTD (i.e. the largest number of alternate productions for an XML element), which in turn is bounded by 2^k. Combined together, we have the following running time bound on **Algorithm FastPV**

```
class ECRecognizer

active − nodes = empty

ECRecognizer(Element e)
    r = root(DAG_T(e))
    append children(r) to active−nodes

validate(Element x)
    result = "reject"
    foreach node n in active−nodes
        if type(n) = choice-star
            matched = false
            foreach element y in n.elements−list
                if x = y or lookup(x, y) = true
                    matched = true
                    break
            if matched
                result = "accept"
            else
                remove n from active−nodes
                append children(n) to active−nodes
        else
            if element(n) = x
                result = "accept"
                remove n from active−nodes
                pre-pend children(n) to active−nodes
                continue
            if lookup(x, element(n)) = true
                if n.recognizer = null
                    n.recognizer = new ECRecognizer(element(n))
                if n.recognizer.validate(x) = "accept"
                    result = "accept"
                    continue
            remove n from active−nodes
            append children(n) to active−nodes
    return result
```

Figure 9: The ECRecognizer algorithm

THEOREM 7. *The FastPV algorithm decides* ***Problem PV*** *for an input instance w, DTD T and root element ρ in $O(n \cdot 2^k)$ time.*

We note, that for a fixed DTD, k will be a constant, and therefore **Algorithm FastPV** will run in linear time of the size of the input XML file. The constant factor 2^k is also a very conservative estimate, as in practice, the branching factor for DTDs is much smaller than the total size of the DTD.

4. EVALUATION

We have implemented two potential validity checkers: one based on Earley's algorithm as described in Section 3.1 and the other – the implementation of the **Algorithm FastPV** described in 3.2. Below we give a brief description of the implementation and the tests.

Both checkers were implemented in Java. In addition, we have created programs for analyzing input DTDs and generating DTD information: dependency graphs, lookup tables and grammar rules and programs for parsing and tokenizing input XML files.

In the tests described here, we have evaluated only the performance of the potential validity checkers, for the whole documents, without including the times for DTD analysis and parsing. We have tested the two methods on a generated dataset based on five XML DTDs, which we will call DTD1, DTD2, DTD3, DTD4 and DTD5. DTDs 1 through 4 were generated artificially, while DTD5 was generated by Pizza Chef [7], the Text Encoding Initiative (TEI) [17] software for constructing TEI-compliant DTDs. DTD1 and DTD2 have 25 elements each, with DTD2 having twice the branching factor (number of alternate productions for DTD elements) of DTD1. Similarly, DTD3 and DTD4 have 50 elements each, and the branching factor of DTD4 is twice that of DTD3. DTD5 has 193 elements, a relatively high branching factor, but unlike other DTDs, most of its rules are star-choice.

An XML generator program had been written to construct potentially valid XML files given a DTD. The input parameters for the XML generator were (in addition to the DTD), the number of tokens to be generated and the *depth* parameter, regulating the depth of the DOM trees for the generated XML. For each DTD we have generated XML instance documents for six sizes: 1000, 10,000, 100,000, 500,000, 1,000,000 and 5,000,000 tokens (a token is a *start tag*, *end tag*, or a uninterrupted string of character data) and for six depth parameters: 3,4,5,6,7,8. The actual sizes of documents on disk range from 5kB to 40MB. For each triple (DTD, Size, Depth) we have generated four XML instance documents. Due to the requirement that the generated documents are potentially valid, the exact size in number of tokens for each generated instant document varied slightly from the desired size.

Both Earley's algorithm implementation and **FastPV** implementation have been run on all instance XML documents on a Dell Dimension 4100 PC with 1Gb of main memory running Linux. **FastPV** successfully terminated on all instances, whereas Earley's algorithm was able to consistently terminate only on instances from DTDs 1 through 4 of size 100,000 and less. For each (DTD, Size, Depth) triple we averaged the performance of the respective method and the token size over the four instance documents.

The results of the tests are shown in Figure 4. It displays the running times of the tests for Earley's (left) and **FastPV** (right) for the XML instance documents generated with the depth parameter of 8. As seen from the comparison of the running times, **FastPV** performs orders of magnitude better than Earley's and keeps the running time under 25 seconds for the documents with 5,000,000 tokens (for comparison, Earley's algorithm did not scale well at all, running out of memory on 1,000,000 and 5,000,000 tests and taking hours for even smaller tests on documents with 500,000 tokens). The running time of **FastPV** algorithm shows linear dependence on the number of tokens for each DTD considered. It is interesting to note that the fastest processing time was achieved for DTD5, generated from TEI, despite the DTD being much larger. This is due to the fact that most of the rules in DTD5 were star-choice and thus allowed for faster processing. All other DTDs were artificially created with significant branching outside of star-choice rules.

5. RELATED WORK

In [6] XML editors are classified as text editors and structure editors. While the latter type is dealing mostly with the XML tree model and associated operation (node insertions and deletions), the former models better the textual editing of an XML document (with markup insertion over a textual content). Document-centric XML documents with irregular structure are likely to be more appropriately edited in a text editor environment. Due to user interaction time constraints, fast algorithms for various levels of validity checking are required [6].

Potential validity checking is a relaxed schema constraint enforcing for XML documents. Incremental validation [14, 2] of XML documents is a computing time saving solution for document updates validation, an efficient alternative to whole document validation. However, both approaches [14, 2] are considering an initially *valid* document together with a set of update operations and deciding the validity of the resulting document. As pointed out earlier, for editing document-centric XML documents, it is rarely the case that the document is valid until the tagging work is close to be completed. Moreover, a markup insertion operation can be simply described in the textual representation of the XML document as "markup *from* start position *to* end position". In a tree model representation of the XML document, however, such an insertion is a composite operation, as it requires both node insertion and some parent-child relationships changes in the document structure. These make incremental validation more useful in a structure editor environment.

On the other hand, a potentially valid document is not, in most cases, a valid document. Further checking is necessarily to decide the document validity.

A parser for XML DTDs extended context free grammars is

[7] http://www.tei-c.org/pizza.html

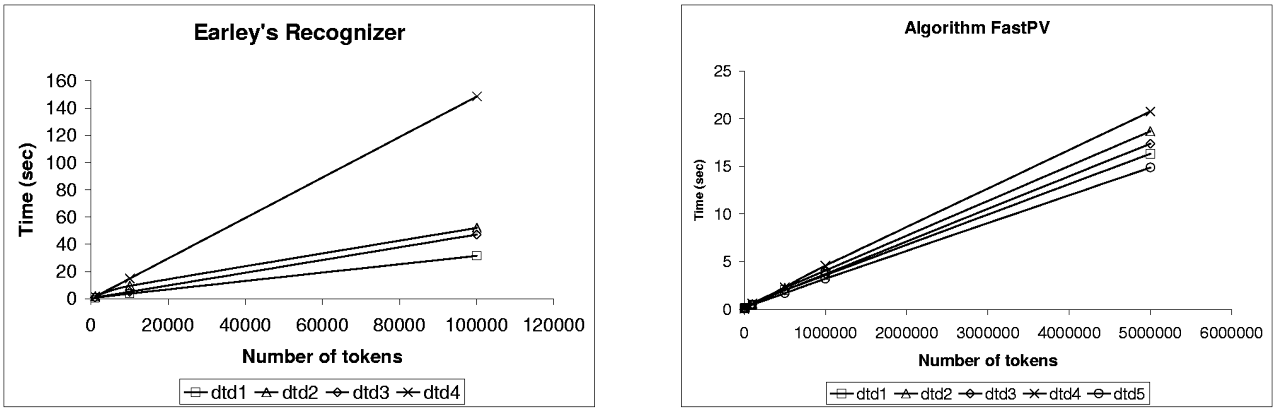

Figure 10: Test Results.

given in [4]. However, that work is based on the *unambiguity* property of the XML right-hand-side grammar productions [3], and excludes from start empty productions for element types. Our solution is based on exploiting the fact that *any* element type derives an empty string.

6. CONCLUSIONS

Fast potential validity checking of XML documents is a useful operation for XML documents updates. It can circumvent unnecessarily validation operations and help making decisions of further document updates.

The problem of potential validity of XML documents often occurs while XML markup is introduced on top of existing content. Any XML editor designed to allow the user to load a text file and mark it up by selecting sequences of characters and then choosing the markup element to tag has to be able to tell if the new XML document can be completed to a valid document. In this paper, we have shown that the set of potentially valid XML documents for a DTD is context-free and can be recognized by an efficient, linear-time (in the size of XML encoding) algorithm. Our algorithm for checking potential validity can be equally easily implemented using two most common models of the XML documents: the tree model and plain text model. The experimental evaluation of this algorithm showed that it is robust and scales well.

Our future work lies in implementing potential validity efficiently as a part of above-mentioned document-centric XML editing software. Also, we want to improve our algorithm in order to be able to decide both validity and potential validity. Furthermore, we want to extend this work to checking potential validity of XML documents with respect to an XML Schema.

Acknowledgments

The work of the first author was supported, in part, by a Collaborative Research Award from the National Endowment for the Humanities (NEH grant RZ-20887-02) and the Andrew W.Mellon Foundation, and sponsored by The British Library and the Bodleian Library, Oxford. The second author's work was supported, in part, by the National Science Foundation under Grant No. 0219924. The work of the third author had been sponsored by Russian Fundamental Studies Foundation (Grant 04-01-00565).

7. REFERENCES

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] Béatrice Bouchou and Mirian Halfeld Ferrari Alves. Updates and Incremental Validation of XML Documents. In Georg Lausen and Dan Suciu, editors, *Proceedings of DBPL 2003*, volume 2921 of *Lecture Notes in Computer Science*. Springer, 2004.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/REC-xml, Oct 2000. W3C, REC-xml-20001006.

[4] Anne Brüggemann-Klein and Derick Wood. On predictive parsing and extended context-free grammars, 2003.

[5] N. P. Chapman. LALR(1,1) parser generation for regular right part grammars. *Acta Informatica*, 21:29–45, 1984.

[6] James Clark. Incremental XML Parsing and Validation in a Text Editor, December 2003. Presentation at XML 2003, Philadelphia.

[7] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM (CACM)*, 13(2):94–102, February 1970.

[8] Jay Earley. An Efficient Context-Free Parsing Algorithm (Reprint). *Communications of the ACM (CACM)*, 26(1):57–61, January 1983.

[9] S. Ginsburg. *The mathematical theory of context-free languages.* McGraw-Hill, 1966.

[10] R. Heckmann. An efficient ELL(1) parser generator. *Acta Informatica*, 23:127–148, 1986.

[11] S. Heilbrunner. On the definition of ELR(k) and ELL(k) grammars. *Acta Informatica*, 11:169–176, 1979.

[12] W. R. LaLonde. Constructing LR parsers for regular right part grammars. *Acta Informatica*, 11:177–193, 1979.

[13] I. Nakata and M. Sassa. Generation of Efficient LALR Parsers for Regular Right Part Grammars. *Acta Informatica*, 23:149–162, 1986.

[14] Yannis Papakonstantinou and Victor Vianu. Incremental validation of xml documents. In *Proceedings of the 9th International Conference on Database Theory*, pages 47–63. Springer-Verlag, 2002.

[15] P. W. Purdom, Jr. and C. A. Brown. Parsing extended LR(k) grammars. *Acta Informatica*, 15:115–127, 1981.

[16] H.-C. Shin and K.-M. Choe. An improved LALR(k) parser generation for regular right part grammars. *Information Processing Letters*, 47(3):123–129, September 1993.

[17] C. M. Sperberg-McQueen and L. Burnard(Eds.). Guidelines for Text Encoding and Interchange (P4). http://www.tei-c.org/P4X/index.html, 2001. The TEI Consortium.

[18] J. W. Thatcher. Characterizing derivation trees of context-free grammars through a generalization of nite automata theory. *Journal of Computer and System Science*, 1(4):317–322, December 1967.