On Potential Validity of Document-Centric XML Documents

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

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

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

Abstract

Document-centric XML document creation is a process of marking up textual content rather than typing text in a predefined structure. It turns out that, although the final document has to be valid with respect to the DTD/Schema used for the encoding, the "in-progress" document is almost never valid. At the same time, it is important to ensure that at each moment of time, the editor is working with an XML document that can be enriched with further markup to become valid. In this paper we explain the notion of potential validity of XML documents, which allows us to distinguish between XML documents that are invalid because the encoding is incomplete and XML documents that are invalid and no further encoding will make the document valid. We show that the set of potentially valid XML documents with respect to any DTD is context-free and we give a linear-time algorithm for checking potential validity for documents and document updates.

1 Introduction

The notion of *potential validity* of an XML document introduced in [11] arose from the study of the needs of human editors of document-centric XML documents. As it happens, the editorial process for document-centric XML documents stands in contrast with that of data-centric XML. In majority of applications, data-centric XML is used to represent semistructured information transported between applications or between a database and an application. Individual subcomponents of the data-centric XML documents may be created at different times - by different means, only to be later merged into a single document as a result of executing a query or passing information from one place to another. Even when parts of such XML are authored by humans, this is typically done by filling the blanks in specially prepared forms found in specific applications or data-centric XML editors [5].

```
<!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, e)>
```

Figure 1. A sample DTD.

For document-centric XML, the author is almost invariably a human. In many applications yielding document-centric XML (such as, for example building document collections in digital libraries projects), the content of the document-centric XML document exists long before any markup is introduced. Human editors then use various ad-hoc methods and procedures[1] *to introduce markup on top of this existing text*.

One similarity, however, remains. In most cases, editing a document-centric XML document is guided by a specific XML schema, which specifies the rules and constraints on the occurrence of elements in the XML encoding. But for document-centric XML, the differences between various ways of representing the schema: DTD, XML Schema[7], Relax-NG [13] are less important – typically all content in document-centric XML documents is treated as strings.

The key reason for the notion of potential validity becomes clear when we observe that in the editorial process, in which markup is gradually and manually added over an extended period of time on top of existing content, the intermediate XML documents are **almost never valid**. In [11] we point at two possible sources of such invalidity: (i) *incompleteness* of the encoding and (ii) direct `violation` of a schema rule. This is shown in the following example.

*Work supported, in part, by the NEH grant RZ-20887-02.
†Work supported, in part, by the ITR grant 0325063.
‡Work supported, in part, by the RBRF grant #04-01-00015

[1] In [10] we have reported on a document-centric XML editor which incorporates algorithms described in this paper.

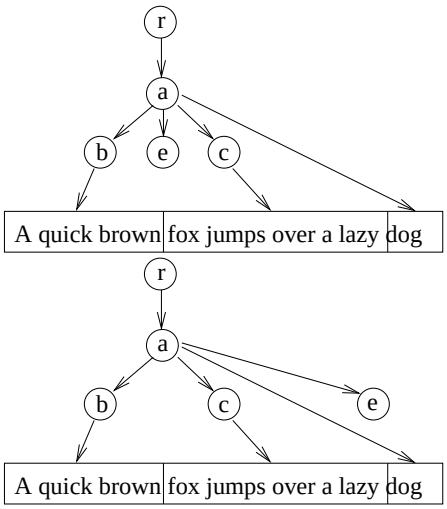

Figure 2. DOM Trees for Example 1

Example 1. *Consider the phrase "*`A quick brown fox jumps over a lazy dog`*" encoded with elements from the DTD in Figure 1 in the following two ways:*

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

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 [4] for both XML fragments. By comparing the encodings with the structure required by the DTD (Figure 3), it is easy to notice that both XML fragments are *not valid* [2] with respect to it. 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 `<d>` to produce the encoding:

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

The documents of the second type, which can be made valid by adding more markup were called *potentially valid* in [11]. In that paper, together with formulating the definition, we have shown that the set of potentially valid XML documents w.r.t. a given DTD [2] is *context-free*. While this result implies immediately, that there exists a polynomial algorithm for determining potential validity[6], we showed in [11] that because the language of potentially valid documents is highly ambiguous, such standard CFG parsing algorithms as Earley's are not practical. In [11] we have began our search for an efficient algorithm for checking potential validity by offering a linear-time algorithm that correctly works on a subclass of DTDs – DTDs *without recursive elements*. While most standard examples of document-centric XML markup rarely have recursive elements, the DTDs often allow them - e.g., in XHTML `<b>` and `<i>` elements can be nested in any way, and thus require appropriate DTD/XSchema structures, despite the fact that we rarely see a `<i><b><i>` combination.

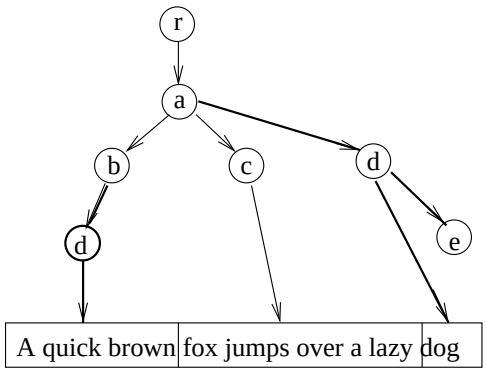

Figure 3. Extending encoding to obtain valid XML.

In this paper, we complete the work started in [11]. We formally show that the class of potentially valid XML documents is closed under markup and content deletions and that checking for potential validity on content updates is $O(1)$ time. We distinguish three classes of DTDs, *non-recursive*, *PV-weak recursive*, and *PV-strong recursive* and we propose a new efficient linear-time complexity algorithm for recognizing potential validity for **any DTD**.

The rest of the paper is organized as follows. In Section 2 we formally define the notion of potential validity. We show how to construct a context-free grammar for checking potential validity in Section 3. Our linear-time algorithm for checking potential validity is described in Section 4.

2 Potentially Valid XML Documents

As in [11], we can specify *potential validity* of an XML document as follows.

Definition 1 (informal potential validity [11]). *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*

[2]Because potential validity concerns only structural properties of the XML document, any schema definition method, DTD, XML Schema or Relax-NG can be used. Without loss of generality we concentrate on the DTDs in this paper as well.

valid by inserting more markup tags, from the given DTD, at some positions.

We make the observation that the potential validity definition above can be straightforward generalized to any other XML schema language (XML Schema [7], Relax NG [13], etc.). Our particular choice for DTD is rather related to the solution we propose in this article for the potential validity problem than how the XML schema language is specified.

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 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[3] [2] 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 element $r \in \mathcal{T}$ 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 [2]. 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 (XML string extension [11]). *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 $\omega \in Ext(w, \mathcal{T})$ if there exists an element $\delta \in \mathcal{T}$ 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) ω can be written as: $\omega = w_1 <\delta> w_2 </\delta> w_3$ and

(c) ω is an XML string.

Intuitively, $Ext(w, \mathcal{T})$ is the set of all possible XML strings obtained by tagging w 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.

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

Definition 3 (potential validity [11]). *Let $T = \langle \Gamma, \mathcal{T} \rangle$ be a DTD and $r \in \mathcal{T}$. An XML string w with $elements(w) \subseteq \mathcal{T}$ and $root(w) = r$ is called* potentially valid *with respect to T and r if $(\exists \omega \in Ext(w, \mathcal{T}))(\omega \in \mathcal{D}(T, r))$.*

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

Example 2. *Consider again the DTD T in Figure 1 and the XML documents instances w and s in Example 1. So $\mathcal{T}$ = {r, a, b, c, d, e, f}, the root element is r, and let a character data string* `"A quick brown fox jumps over a lazy dog"`.
We let
`w'= "<r><a><b><d>A quick brown</d></b><c> fox jumps over a lazy</c><d> dog<e></e></d></a></r>"`.
Since $w' \in D(T, r)$ and $w' \in Ext(w, \mathcal{T})$ it follows that $w \in \mathcal{D}^*(T, r)$.

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

3 Checking potential validity

We define the problem of checking potential validity of XML documents as follows ([11]):

> <u>*Problem PV:*</u> Given a DTD $T = \langle \Gamma, \mathcal{T} \rangle$, and an XML string w with $elements(w) \subseteq \mathcal{T}$ and $root(w) = r \in \mathcal{T}$, output "yes" if $w \in \mathcal{D}^*(T, r)$ and "no" otherwise.

In this section we consider a given DTD $T = \langle \Gamma, \mathcal{T} \rangle$, and a root element $r \in \mathcal{T}$ and we show that the set of all XML strings w for which $PV(T, r, w) =$ "yes" is context-free. We do this by constructing an extended context-free grammar (ECFG)[4] that recognizes potentially valid XML

[3] 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 in any way our consideration of the problem presented in this paper.

[4] 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.

documents with root r. For the rest of the paper we use the CFG grammar notations as in [1]. The empty string is denoted by ϵ and for a grammar G, we denote by $L(G)$ the language accepted by G.

Intuitively the problem solution consists of a set of rules (for accepting strings which represent XML documents) derived from the rules in Γ by relaxing the requirements for the presence of *start tag* and *end tag* markups. That is, the *start tag* and *end tag* of an element might be present or not during the marking up process, however, the element content must be derived according to the rules in Γ.

3.1 The ECFG for checking validity

The extended context-free grammar for recognizing XML strings with root r valid w.r.t. T follows straightforward from Γ[5]. More formally, we let $G_{T,r} = (N, \Sigma, \mathcal{R}, S)$ be the ECFG representation of T, where N is the set of nonterminals, Σ is the set of terminals, $\mathcal{R}$ is the set of grammar rules, and S is the start symbol. The set N of nonterminals contains a nonterminal, $PCDATA$, corresponding to the $\#PCDATA$ keyword in the Element Type Definition and, for each element $x \in T$, there are two corresponding nonterminals, $X, \hat{X} \in N$. In particular, R, and $\hat{R}$ correspond to the root element r.

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

Σ contains a terminal σ, corresponding to any string of non-markup characters of length at least one (a non-empty data character string). For each element $x \in \mathcal{T}$ there are two corresponding terminals, one for its *start tag*, $<x>$ and the other for its *end tag*, $</x>$.

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

For each element $x \in \mathcal{T}$, we denote by r_x the right-hand side of the Element Type Definition in Γ for x and we denote by r_X the transcription of r_x where every element y in r_x is replaced by its corresponding nonterminal Y. For an element $z \in \mathcal{R}$ of which the Element Type Definition rule in Γ contains the keyword "ANY" [2], the rule r_Z is rewritten as:

$(Z_1|Z_2|\ldots|Z_n|PCDATA)*$,

where $Z_1, Z_2, \ldots, Z_n$ are the nonterminals for all elements in $\mathcal{T}$.

Then:

$$\mathcal{R} = \{S \to R, PCDATA \to \sigma, PCDATA \to \epsilon\} \cup \{X \to <x>\hat{X}</x>,\ \hat{X} \to r_X | x \in \mathcal{T}\}.$$

Example 3. *For the DTD T in Figure 1, let $G_{T,r} = (N, \Sigma, \mathcal{R}, S)$:*

$N = \{S, PCDATA, R, \hat{R}, A, \hat{A}, \ldots, F, \hat{F}\}$

$\Sigma = \{\sigma, <r>, </r>, <a>, , \ldots, <f>, </f>\}$

$\mathcal{R} = \{S \to R, PCDATA \to \sigma \mid \epsilon\} \cup$

$\{R \to <r>\hat{R}</r>, \hat{R} \to A+,$

$A \to <a>\hat{A}, \hat{A} \to B?, (C|F), D,$

$B \to \hat{B}, \hat{B} \to (D|F),$

$C \to <c>\hat{C}</c>, \hat{C} \to PCDATA,$

$D \to <d>\hat{D}</d>, \hat{D} \to (PCDATA \mid E)*,$

$E \to <e>\hat{E}</e>, \hat{E} \to \epsilon,$

$F \to <f>\hat{F}</f>, \hat{F} \to C, B, E\}$

For the purpose of checking validity, we need to convert XML strings into strings recognized by the ECFGs described above. We introduce an operator,

$$\delta_T : \bigcup\{w | w \text{ is an XML string}, elements(w) \subseteq \mathcal{T}\} \to \Sigma^*$$

defined recursively as follows:

• For any (possibly empty) character data content C,

$$\delta_T(C) = \begin{cases} \sigma, & \text{if } C \text{ is not the empty string} \\ \epsilon, & \text{otherwise} \end{cases}$$

. • For any $a \in \mathcal{T}$, $\delta_T(< a/ >) = <a>$.

• Let w be an XML string with $elements(w) \subseteq \mathcal{T}$ and $w = w_1<a>w_2w_3$, where w_2 is an XML string. Let $\delta_T(w_1) = d_1, \delta_T(w_2) = d_2$, and $\delta_T(w_3) = d_3$. Then $\delta_T(w) = d_1<a>d_2d_3$.

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: $\delta_T(<a>$A quick brown$<c>$ fox jumps over a lazy$</c><d>$ dog$<e></e></d>) =$
$<a>\sigma<c>\sigma</c><d>\sigma<e></e></d>$

3.2 The ECFG for checking potential validity

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

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

$\mathcal{R}' = \mathcal{R} \cup \{X \to \hat{X} | x \in \mathcal{T}\}$.

The intuition behind this extension of $G_{T,r}$ is as follows. Given a valid (w.r.t. DTD T) XML document w with root r, we can construct potentially valid XML documents from it by selecting one or more tags in w and removing them to produce document $\omega \in \mathcal{D}^*(T, r)$. In the derivation of $S \Rightarrow_G^* \delta_T(w)$, the "open tag" and "close tag" terminals are derived via the rules of the form $X \to <x>\hat{X}</x>$. By inserting the new rule $X \to \hat{X}$ for each XML element $x \in \mathcal{T}$, we are allowing the grammar $G'_{T,r}$ to mimic the derivation of $\delta_T(w)$, and convert it into the derivation

[5] The DTD's Element Type Definitions are ECFG productions [2].

of $\delta_T(\omega)$ by electing not to derive the XML tag terminals where needed.

We are now ready to state the main result of this section.

Theorem 1. *Given an XML string w, $w \in \mathcal{D}^*(T,r) \Leftrightarrow \delta_T(w) \in L(G'_{T,r})$.*

The following result shows that character data updates and markup deletions preserve potential validity. Markup insertion or deletion means insertion or deletion of pairs of start and end tags so that the XML document remains well-formed. We say we have a character data update when it refers to a change (deletion or insertion of characters) in an *existent text node* of the XML document. Character data insertion refers only on *text node creation*. For instance, let us consider the XML document string: `<a>XML<space></space>string</a>`. Then the XML string `<a>an XML<space></space>string</a>` is obtained by a character data update whereas the XML string `<a>XML<space>-</space>string</a>` represents a character data insertion.

Theorem 2. *The class of potentially valid documents $\mathcal{D}^*(T,r)$ is closed under character data updates and markup deletions.*

3.3 Properties of the ECFG for checking potential validity

Extended context free grammars (or regular right part grammars) were defined a long time ago [15, 9] and many parsers and recognizers have been proposed for them ([14, 3], just to name two). It is important to note that, however, some parsers work not for general ECFGs but for certain restricted cases.

Most of the grammars in the family of grammars $G'_{T,r}$ we construct 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 but they exhibit poor performances for practical applications of checking potential validity.

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

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 $x \in \mathcal{T}$ is called *usable* if $\exists z \in L(G)$ and a derivation $S \Rightarrow_G^* z$ that contains the nonterminal X [1]. It is known that given a CFG, the set of its usable nonterminals can be efficiently constructed [8]. From now on we consider that all XML elements in the DTD T are usable.

The following property of grammar $G'_{T,r}$ is important for us.

Theorem 3. *For any $X \in N$, $X \Rightarrow^*_{G'_{T,r}} \epsilon$.*

Immediate consequences of Theorem 3 allow us to simplify the grammar $G'_{T,r}$.

Corollary 3.1. *Let $T' = \langle \Gamma', \mathcal{T} \rangle$ be a DTD obtained from T by removing all occurrences of the "?" and replacing "+" operators by "*" operators in Γ. Then $L(G'_{T,r}) = L(G'_{T',r})$.*

While the grammar $G'_{T,r}$ 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], makes general parsing algorithms too inefficient. Corollary 3.1 allows us to reduce $G'_{T,r}$ complexity without altering the grammar language.

For the rest of this article we consider that the DTD T has no "?" operators and that "+" operators were replaced by "*" operators.

Definition 4 (star-group). *For any element $x \in \mathcal{T}$ a* star-group *is a subexpression of r_x so that all of the following apply:*
(i) each expression of form $a*$ or $(\ldots)*$ is either a star-group or a subexpression of a star-group ($a \in \mathcal{T}$ and the notation $(\ldots)$ corresponds to any parenthesized expression);
(ii) no star-group is a subexpression of a star-group.
We call the elements contained by a star-group the star-group elements.

For instance, in $r_x = (a, (b * |(c, d*, e)*))$, the expressions $b*$ and $(c, d*, e)*$ are star-groups, but $d*$ is not a star-group.

The following result states the independence of the grammar language from the expression in a star-group.

Proposition 1. *Let $T' = \langle \Gamma', \mathcal{T} \rangle$ be a DTD obtained from T by replacing each star-group containing elements $a_1, \ldots, a_n \in \mathcal{T}$ with a star-group of form $(a_1, \ldots, a_n)*$. Then $L(G'_{T,r}) = L(G'_{T',r})$.*

In other words, Proposition 1 establishes that a star-group is matched by rules depending on star-group elements but independent on the star-group expression. Given that the #PCDATA keyword appears alone or in a mixed content of an element type declaration ([2]), it follows that checking

[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.

for potential validity on character data insertion reduces to checking whether or not an element type declaration contains #PCDATA keyword (hence, $O(1)$ time complexity).

4 Algorithms for checking potential validity

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. As pointed out earlier, a general ECFG parser algorithm is not the most appropriate solution for recognizing strings in $L(G'_{T,r})$.

We make the observation that we can solve the potential validity problem incrementally, for each document node, by considering only node's children. To formalize this we introduce a new operator, Δ_T, that transforms an XML string, rooted at some node a, into a sequence of symbols corresponding to the children of node a (in the *document order* [4]). We define

$$\Delta_T : \bigcup\{w | w \text{ is an XML string}, elements(w) \subseteq \mathcal{T}\} \rightarrow \Sigma^*$$

as follows:

• For any (possibly empty) character data content C, $\Delta_T(C) = \delta_T(C)$

• Let $<r>w</r>$ be any XML string and let $<r>w'</r>$ be the XML string obtained from $<r>w</r>$ by removing all descendants but children of its root node. Then $\Delta_T(<r>w</r>) = \delta_T(<r>w'</r>)$.

For instance, for the XML string `w = "<a><b>A quick brown</b><e></e><c> fox jumps over a lazy</c> dog</a>"` we have $\Delta_T(w)$ = `<a><b></b><e></e><c></c>`σ`</a>`.

We emphasize at this point that, although we employ the document tree model, the operator Δ_T (or δ_T) can be equally implemented for a document string model as well as for a document tree model.

We formally define the problem of checking potential validity for an XML node as follows.

> *Element Content Potential Validity (Problem ECPV):*
> Given an XML string w with $elements(w) \subseteq \mathcal{T}$ and $root(w) = a \in \mathcal{T}$, output "yes" if $\Delta_T(w) \in L(G'_{T,a})$ and "no" otherwise.

We observe that checking potential validity for markup insertion into a potentially valid document reduces to solving twice Problem ECPV: for the node inserted and for its parent. In this section we develop an efficient algorithm for solving Problem ECPV. Solving Problem PV can be then performed by checking the potential validity of every node in the document (Problem ECPV) and the algorithm for this is given in [11]. We also make the observation that for any DTD element of which the Element Type Declaration contains the keyword "ANY" [2] the ECPV problem presents no practical interest. For all XML documents instances the content of a such element is potentially valid.

4.1 Element Content Recognizer

We start by observing that, for any terminal symbol $x \in \Sigma$ in an XML string and any nonterminal $Y \in N'$ the derivation $Y \Rightarrow^*_{G'_{T,r}} x$ depends only on the grammar G'. Therefore, given the grammar $G'_{T,R}$ (i.e. the DTD), we can pre-compute whether or not a given symbol $x \in \Sigma$ can be derived from a given nonterminal $Y \in N'$.

Symbols reachability pre-computation gives immediately a "no" answer to Problem ECPV of an element x if some symbol in the input string is not reachable from X.

Definition 5 (reachability graph). *The reachability graph of T is the directed graph $R_T = (V, E)$, $V = \mathcal{T}$, $E = \{(t_1, t_2) : t_1, t_2 \in \mathcal{T}, t_2$ appears in $r_{t_1}\}$.*

We say that an element x or $PCDATA$ is *reachable* from another element y, denoted $y \rightsquigarrow x$, if there is a path from y to x in R_T.

The reachability graph of T tells us whether or not the markup of a given element $t_2 \in \mathcal{T}$ may be found in the markup content of another element t_1. With R_T constructed, the reachability relation between its nodes can be pre-computed in a form of a *lookup table*, L_T, such that $L_T(t_1, t_2) = true$ if $t_1 \rightsquigarrow t_2$ in R_T, and $L_T(t_1, t_2) = false$ otherwise.

The following results allow us to implement an efficient recognizer for $G'_{T,r}$.

Proposition 2. *1. For any $a, b \in \mathcal{T}$, $A \Rightarrow^*_{G'_{T,r}} B$ iff $b \rightsquigarrow a$ in R_T.*

2. For any star-group expression g_x in T and for any XML string w we have $g_X \Rightarrow^* \Delta_T(w)$ if:

(a) $\forall <a>, \in \Delta_T(w) : \exists y$ an element of g_x such that $y \rightsquigarrow a$ in R_T.

(b) $\sigma \in \Delta_T(w) : \exists y$ an element of g_x such that $y \rightsquigarrow PCDATA$ in R_T.

Due to DTD syntax, namely the keyword PCDATA appears alone in an Element Type Declaration or in a *mixed-content* [2], the following property is immediate:

Proposition 3. *Let $w \in \mathcal{D}^*(T, r)$ and let w' the XML string obtained from w by inserting character data as text node for some element node x. Then $w' \in \mathcal{D}^*(T, r)$ iff $x \rightsquigarrow PCDATA$.*

It follows that checking potential validity for character data insertion into a potentially valid document is $O(1)$ using a lookup table.

4.2 A DAG model for DTD

For the DTD T we describe now a Directed Acyclic Graph model (DAG_T) that allows us to efficiently solve Problem ECPV. DAG_T model is a collection of directed acyclic graph components. For each element $x \in \mathcal{T}$ an *element DAG* (DAG_x) is constructed, and $DAG_T = \cup_{x \in \mathcal{T}} DAG_x$.

An element DAG_x has a *root* node, labelled x. Its other nodes are of two types: *simple element* nodes and *star-group* nodes. A simple element node n_y of DAG_x corresponds to each element y in r_x, y not in any star-group of r_x. The node is labelled y and we denote $element(n_y) = y$. For each star-group expression in r_x there is a corresponding star-group node g. The node is labelled as the list of all elements in the star-group expression. We denote by $elements(g)$ the set of all elements in g's corresponding star-group expression. A graph edge connects a node corresponding to an element or star-group expression to the node corresponding to the adjacent element or expression (i.e., comma separated). An "or" operator ("|") introduces branching. As a result, any path in the graph of an element x, from root to a leaf, completely describes a production alternative from $X \rightarrow \hat{X}$.

The DAGs of two elements of the DTD in Figure 1 are given in Figure 4. In the figure, star-group nodes are represented as boxes and the root and simple nodes as circles. The nodes are labeled as described above. We observe that, for instance, all paths in DAG_a ($a \rightarrow b \rightarrow c \rightarrow d$ and $a \rightarrow b \rightarrow f \rightarrow d$) correspond to production alternatives $A \rightarrow BCD$ and $A \rightarrow BFD$ respectively.

The reason of having a graph for each element instead of unique, bigger graph for the whole DTD is one of storage requirements: the bigger graph might contain multiple element graph copies as an element can appear in many productions. Instead, we store a small graph for each element and a bigger graph is constructed as needed for a specific element content recognizer.

4.3 Algorithm ECRecognizer

Checking potential validity for an element content requires checking all possible derivations of the corresponding element nonterminal in the grammar G'. Intuitively, this operation is expensive due to backtracking: once a derivation fails to produce the element content, other derivations need to be checked. In order to produce fast algorithms for checking potential validity, we need to reduce backtracking at a minimum. We do this by using a greedy approach:

For a given DTD there are two possible causes for backtracking: (i) the star-groups and (ii) recursive elements (elements for which the corresponding nonterminals derive themselves). Proposition 2 (1) resolves the star-group case:

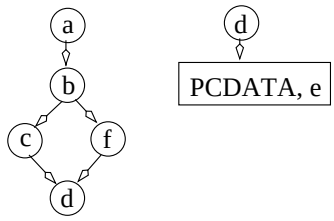

Figure 4. DAGs for Elements Type Declaration of elements a and d in the DTD in Figure 1

it is enough to check for reachability (which can be precomputed).

The case of DTDs with recursive elements require a more careful examination. Although recursive elements can be eliminated in many cases by carefully defining the DTD, they might occur in practice. We start the discussion by formally defining the recursive elements.

Definition 6 (recursive). *An element $x \in \mathcal{T}$ is called a* recursive element *if there exist a derivation $X \Rightarrow^*_{G'_{T,r}} X$. A DTD with recursive elements is said to be recursive, otherwise it is said to be non-recursive.*

We observe that, in some situations, recursion may occur through the elements of a star-group. In this situation, the Proposition 2 (1) is still applicable, so we need to distinguish these cases.

Definition 7 (PV-strong recursive). *A recursive element $x \in \mathcal{T}$ is called a* PV-strong recursive element *if there exist a derivation $X \Rightarrow^*_{G'_{T,r}} X$ such that each non-empty employed production corresponds to a non star-group element. A DTD with at least one PV-strong recursive element is said to be PV-strong recursive.*

A trivial example of a strong recursive element would be the element a with the following Element Type Definition:

```
<!ELEMENT a ((a | c), b*)>
```

Definition 8 (PV-weak recursive). *A recursive element $x \in \mathcal{T}$ which is not PV-strong recursive is called a* PV-weak recursive element. *A recursive DTD with no PV-strong recursive elements is said to be PV-weak recursive.*

In the following, we give an algorithm for solving the *Problem ECPV* any DTD, which extends the algorithm in [11] for solving the *Problem ECPV* for non recursive DTDs and PV-weak recursive DTDs.

4.3.1 PV-strong recursive DTDs

The element content recognizer algorithm (*ECRecognizer*), presented in Figure 5, solves the *Problem ECPV* for any DTD. The algorithm builds an *ECRecognizer object* from

```
class ECRecognizer
(1)  DAG DAG_T, integer depth
(2)  lookup table L_T
(3)  Set activeNodesSet = empty

(4)  ECRecognizer(DAG D, lookup table L, Element e, integer d){
(5)    DAG_T = D, depth = d
(6)    L_T = L
(7)    r = root(DAG_T(e))
(8)    append children(r) to activeNodesSet
(9)  }

(10) method: validate(Element x) {
(11)   result = "reject"
(12)   foreach node n in activeNodesSet
(13)     if type(n) = "star-group"
(14)       matched = false
(15)       foreach element y in elements(n)
(16)         if x = y or lookup(x, y) = true
(17)           matched = true
(18)           break

(19)       if matched
(20)         result = "accept"
(21)         continue
(22)     else
(23)       if lookup(x, element(n)) = true
(24)         if n.recognizer = null
(25)           n.recognizer =
               new ECRecognizer(DAG_T, L_T,element(n), depth-1)
(26)         if n.recognizer.depth > 0
             and n.recognizer.validate(x) = "accept"
(27)           result = "accept"
(28)           continue
(29)       if element(n) = x
(30)         result = "accept"
(31)         remove n from activeNodesSet
(32)         pre-pend children(n) to activeNodesSet
(33)         continue

(34)     remove n from activeNodesSet
(35)     append children(n) to activeNodesSet

(36)   return result
(37) }

(38) method: recognize(Element x_1, x_2, ..., x_n) {
(39)   foreach x in {x_1, x_2, ..., x_n}
(40)     if validate(x) = "reject"
(41)       return "reject"
(42)   return "accept"
(43) }
```

Figure 5. The ECRecognizer **algorithm for any DTD**

DAG_T, lookup table L_T, and the element e whose content is to be recognized. To solve Problem ECPV for a given markup node t, the method $recognize()$ is used on all children nodes of t (transformed into a sequence of elements and PCDATA using Δ_T operator). The algorithm's core is the method $validate()$ that is able to recognize the element content as the input symbols are read. Starting with DAG_e, the $validate()$ method executes a greedy search, on all branches, of each symbol in the tokenized input. The search for the current symbol corresponding to an element (or PCDATA) x performs only if $L_T(element(e), x) = true$ and stops in one of the following two situations:
(i) When the simple element node n, $label(n) = x$, is encountered. However, if $element(n) \neq x$ but

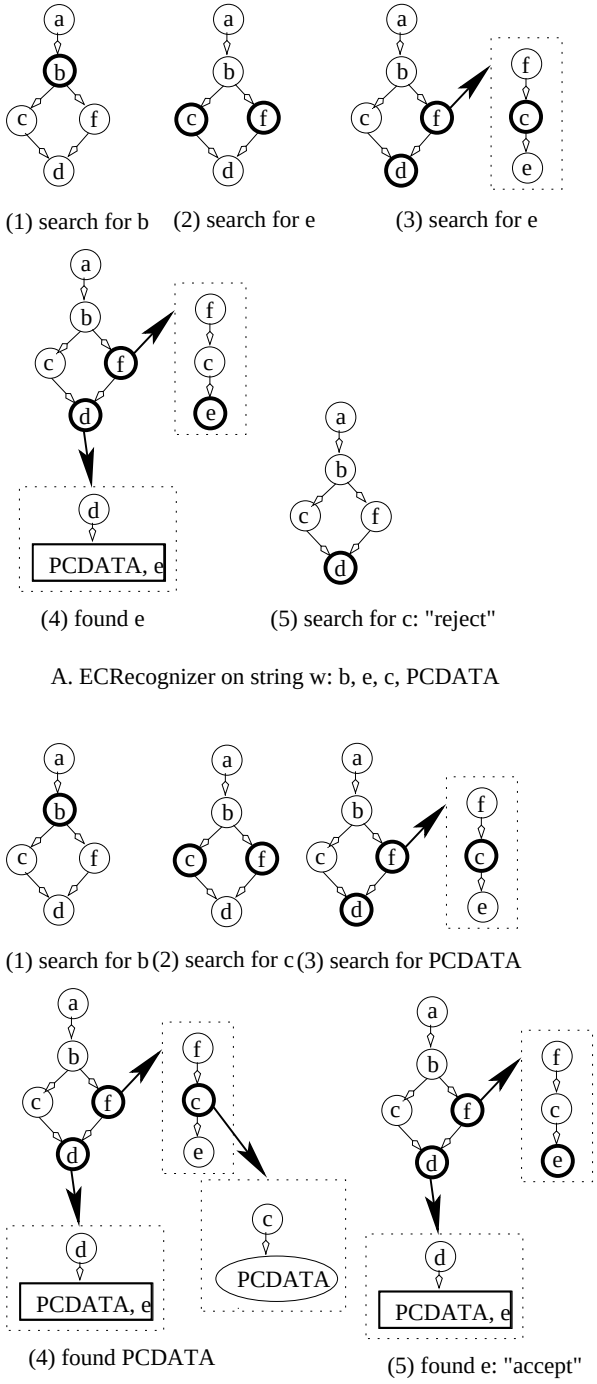

Figure 6. ECRecognizer on content of $<a>$ in XML strings w (A) and s (B) from Example 1

$L_T(element(n), x) = true$ then the current graph is augmented by plugging in $DAG_{element(n)}$ and the search continues based on the new graph configuration (greediness). The graph augmentation corresponds to using a grammar production $X \rightarrow \hat{X}$. Otherwise, the search continues with the next node in the current graph configuration.
(ii) When a star-group node c occurs, such that either $x \in elements(c)$ or $\exists y \in elements(c),\ L_T(y, x) = true$.

The nodes where the search for the current input symbol stops are saved in an *active nodes set* in order to be used as starting nodes of searching for the next input symbol.

The correctness of the greedy approach (i) is ensured by Theorem 3: if there are other unmatched nonterminals and the input sequence ends, they can derive empty strings. In addition, greediness ensures that a current input symbol cannot match "behind" the current searching point. For situation (ii), correctness follows directly from Proposition 2.

We explain first how the algorithm performs for a non-PV-strong recursive DTD, then we discuss potential problems for PV-strong recursive DTDs and how the algorithm in Figure 5 solves those problems.

Example 4. *Figure 6 (A. and B.) shows how ECRecognizer algorithm performs on the content of markup* $<a>$ *in the XML strings* w *and* s *in Example 1. In the figure, the* active nodes *(nodes in the* activeNodesSet *of ECRecognizer algorithm) are represented with solid thick line; the dotted rectangles represent additional ECRecognizer object created for deep search in an element DAG (see lines 25, 26 of the algorithm in Figure 5; the corresponding element points through an arrow to the ECRecognizer object).*
For the first input (A), the element b *is the only active node (as initialized in line 8 of ECRecognizer algorithm). So searching for* b *is successful (line 29 of the algorithm) and* c*,* f *are the current active nodes and the search continues for the second input symbol,* e*. Note that* b *is not found in the* $lookup\ table$ *of* b *in line 23, neither for this DTD instance nor for other non PV-strong recursive DTDs. This condition is important here to avoid an infinite loop in the algorithm. Since* e *is not reachable from* c*,* c *is removed from the active node set and the next node,* d *is added (lines 34 and 35). New ECRecognizer objects are created for elements* d *and* f*, then* e *is found by these recognizers (steps 3, 4). At this point,* f *is removed from the active node set as its last element was matched. The algorithm rejects in step 5, as from the active node* d *no element* c *can be reached.*
For the second input (B), searching for each input symbol is successful and the algorithm returns "accept".

In the example above we left out of discussion the parameter $depth$. The novelty of the algorithm in Figure 5 consists in solving recursive elements related problems by taking the document depth into account. We can understand better the impact of having recursive elements by analyzing

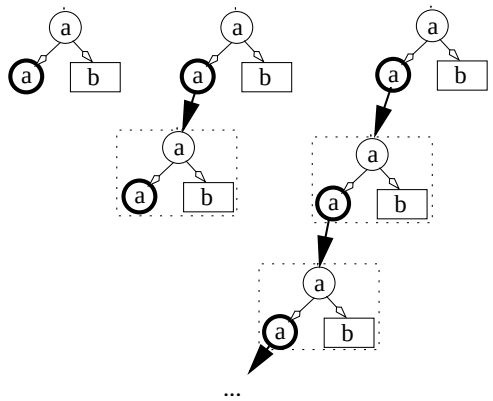

Figure 7. ECRecognizer on content of $<a>$ **in the XML string** $<a>$ **and DTD** T_1 **from Example 5**

the following example.

Example 5. *Let us consider the following DTD* T_1*:*

```
<!ELEMENT a (a | b*)>
<!ELEMENT b EMPTY>
```

The element a *is clearly a PV-strong recursive element.*

Then let us consider the following valid XML instance (with respect to the DTD T_1 *above):* `<a><b></b><b></b></a>`

Let us use the ECRecognizer algorithm on the input above and to ignore any bound for the document depth (that is, $depth$ *is set to infinity): few steps of the algorithm are given in Figure 7 (we use the same representation as in Example 4 and Figure 6). We observe that the algorithm enters in an infinite loop (*a *is the first node in the* $activeNodesSet$ *therefore an infinite number of recognizers are created in line 25).*

The infinite loops in the ECRecognizer algorithm [11] are a consequence of the greedy approach: the algorithm matches as many input symbols as possible before moving to the next node, in order to avoid backtracking. Loop detection and breaking can be done, however, this may affect the algorithm performances. For instance, in Example 5 we can detect the PV-strong recursive element a and force the algorithm to match on the right branch (star group node b) of a's DAG (Figure 7) while keeping a pointer on node a (for backtracking). In that case no backtracking occurs: all input symbols match in the star group node b. However, this is not always the case.

Example 6. *Let us consider the PV-strong recursive DTD* T_2*:*

```
<!ELEMENT a ((a | b), b)>
<!ELEMENT b EMPTY>
```

Then let us consider the XML instance: `<a><b></b><b></b></a>`. *This is a potentially valid XML string since it can be obtained from the valid XML string (with respect to T_2)* `<a><a><b></b></a><b></b></a>` *by removing the inner* `<a>` *tags. It is clear that, in this case, taking one recursive step (line 25) is absolutely necessary.*

We observe that each execution of line 25 of the algorithm is equivalent to using a production $A \rightarrow \hat{A}$ of grammar G'. This means that the possibility of missing tag $<a>$ is taken into account, hence considering the corresponding valid XML string having the depth one unit more than the actual input XML string.

To overcome the problem of infinite loops for some XML instances in the case of PV-strong recursive DTDs we impose an *upper bound* on the depth of the input document to be checked for potential validity. This approach has a strong practical motivation: it is known that, in practice, most XML documents' depths are of one digit magnitude [12]. For flexibility, the document depth upper bound is a parameter of ECRecognizer algorithm: the fist instance of an ECRecognizer object takes the upper bound value as input, whereas the subsequent ECRecognizer objects created in line 25 (Figure 5) take as input the parent's depth upper bound minus one. The major changes of the algorithm in [11] are in lines 25 and 26. The depth decreases one unit each time the algorithm generates a new recognizer (line 25), and the maximum depth (that is, whether or not the depth decreased from maximum depth up to zero) is checked in line 26.

4.4 Complexity of ECRecognizer

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 w and on the properties of the input DTD T. Let n be the number of symbols in $\delta_T(w)$, $m = |\mathcal{T}|$ be the number of XML elements in T, and let k be the number of element occurrences in all r_x expressions, $x \in \mathcal{T}$. We observe that k is an appropriate measure for T since $k \geq m$ and it takes $O(k)$ steps to read all DTD's rules.

The essential step in solving the Problem PV is using the algorithm ECRecognizer for checking potential validity for each node of the input XML document. The following result establishes the time complexity of ECRecognizer algorithm for non PV-strong recursive DTDs.

Theorem 4. *For any DTD T and a maximum acceptable documents depth D, the method $recognize()$ of ECRecognizer algorithm in Figure 5 decides Problem ECPV in $O(k^D \cdot n)$ time, where n is the number of input tokens.*

We note, that for a fixed DTD, k, D, and m are constants, and therefore Algorithm ECRecoginizer runs in linear time of the size of the input XML file. The constant factors k^D and k^m are also very conservative estimates, as in practice, the branching factor for DTDs is much smaller.

5 Conclusions

In this paper, we complete the work started in [11]. We formally show that the class of potentially valid XML documents is closed under markup and content deletions and that checking for potential validity on content updates is $O(1)$ time. We distinguish three classes of DTDs, *non-recursive*, *PV-weak recursive*, and *PV-strong recursive*, and we propose a new efficient linear-time complexity algorithm for recognizing potential validity for **any DTD**.

References

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

[2] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau(Eds.). Extensible Markup Language (XML) 1.0 (Third Edition). http://www.w3.org/TR/2004/REC-xml-20040204/, Feb 2004. W3C Recommendation.

[3] A. Brüggemann-Klein and D. Wood. On predictive parsing and extended context-free grammars, 2003.

[4] M. Champion, S. Byrne, G. Nicol, and L. Wood(Eds.). Document Object Model (DOM) Level 1 Specification. http://www.w3.org/TR/REC-DOM-Level-1/, Oct 1998. W3C Recommendation, REC-DOM-Level-1-19981001.

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

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

[7] D. C. Fallside and P. Walmsley(Eds.). XML Schema (Second Edition). http://www.w3.org/TR/xmlschema-0/, Oct 2004. W3C Recommendation.

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

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

[10] I. E. Iacob and A. Dekhtyar. xTagger: a new approach to authoring document-centric XML. In *JCDL '05*, pages 44–45, New York, NY, USA, 2005. ACM Press.

[11] I. E. Iacob, A. Dekhtyar, and M. I. Dekhtyar. Checking Potential Validity of XML Documents. In *WebDB*, pages 91–96, 2004.

[12] L. Mignet, D. Barbosa, and P. Veltri. The XML web: a first study. In *WWW '03*, pages 500–510, New York, NY, USA, 2003. ACM Press.

[13] OASIS. Relax NG. http://www.relaxng.org/, Sep 2003.

[14] 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.

[15] 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.