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

The process of creation of document-centric XML documents often starts with a prepared textual content, into which the editor introduces markup. In such situations, intermediate XML is almost never valid with respect to the DTD/Schema used for the encoding. 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 enriched with further markup to become valid. 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 give a linear-time algorithm for checking potential validity for documents.

1. INTRODUCTION

Typically, when researchers talk about data-centric and document centric XML documents, the key difference that comes up is in the structure. However, other important differences between these two classes of XML documents exist. The origin of the document is no less important now than its structure. Most data-centric XML documents are generated automatically by software. Those that are created manually, typically are constructed by the user, first, describing the tree structure, and then inserting the text content into the leaf elements. In contrast, document-centric XML documents are usually produced manually, and, *typically* the textual content is available *before* the markup is introduced.

*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 grants ITR-0219924 and ITR-0325063.
‡Work partially sponsored by Russian Fundamental Studies Foundation (Grant 04-01-00565) and by NSF grant CCR 0100040, courtesy of Judy Goldsmith.

Seventh International Workshop on the Web and Databases (WebDB 2004), June 17-18, 2004, Paris, France.

In a nutshell, the work of the human editor of a document-centric document proceeds as follows: the textual content is loaded, and then, step by step, the editor chooses the fragment of the text, and introduces some encoding for it. In most cases, the order in which the markup is introduced has little to do with the DTD structure, and may be dictated by a specific task at hand for the editor: e.g., marking up document sentences without paying attention to paragraph, section and chapter markup. Throughout most of such process, current XML document will not be valid w.r.t. the DTD/XSchema used for the markup.

At the same time, we note that the invalidity of these unfinished document-centric XML documents can have two different causes. First, it may be due to the fact that the markup is simply incomplete. However, it is also possible that while introducing markup, the editor inadvertently but explicitly violated one of the DTD rules. We illustrate these two types of invalidity on the following example.

Consider a DTD shown on figure 1 and two encodings of the phrase “A quick brown fox jumps over a lazy dog”:

```
<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>
```

Is there a difference between these two XML fragments with respect to our DTD? And if yes, then, what is this difference?

Distinguishing between these two cases is important: the former type of invalidity assures the editor that (s)he can continue introducing new markup into the document, while the latter part means that *no* extension of the current XML document will ever be valid. At the same time, we maintain that there is an important difference between these two XML fragments. 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 two `<d>` element to produce the encoding:

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

```
<!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.

```
a lazy</c><d> dog<e></e></d></a></r>
```

The importance of having fast algorithms for checking validity of XML documents is emphasized in [11]. 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 described above. In this paper, we introduce a notion of *potentially valid* XML documents, as documents that can be turned into valid XML documents only by inserting new XML elements[1]. We then describe an efficient algorithm for testing potential validity.

The rest of the paper is organized as follows. In Section 2 we formally define the notion of potential validity. Our linear-time algorithm for checking potential validity is described in Section 3. We present experimental results for the algorithm implementation and testing in Section 4. Finally, in Section 5, we consider some related work and discuss the importance of this work. We omit the proofs due to space limitation.

2. POTENTIALLY VALID DOCUMENTS

Informally, we can specify *potential validity* of an XML document[2] 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 described in Section 1, we want to decide, given some DTD, whether or not a specific 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 from consideration, however, we note that it is important for human editors to know whether the document they are working on can be made valid only using tag insertions, or tag deletions of already introduced markup are required. We note that text insertions or deletions are allowed, but the class of potentially valid XML documents is closed under text insertion and deletion.

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] [4] 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 $\mathcal{T}$, `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 [4]. 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: (i) $w \in Ext(w, \mathcal{T})$; (ii) an XML string $w^* \in Ext(w, \mathcal{T})$ if $\exists w_1, w_2, w_3 : w_1 w_2 w_3 \in Ext(w, \mathcal{T}) \wedge w^* = w_1 <\alpha> w_2 </\alpha> w_3$; (iii) no other string is an extension string of w.

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 $(\exists w^* \in Ext(w, \mathcal{T}))(w^* \in \mathcal{D}(T, C, r))$.

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.

[1] We note here that the key reason behind our choice of "potentially valid" XML documents as those that can be turned into valid by only additions of markup is that it reflects the realities of the editorial process: for as long as the XML document is potentially valid, the editor is free to add new markup; when the document stops being potentially valid, it is a sign that the editor must revisit existing markup and effect some changes to it.

[2] Through this paper, all XML documents (or simply *documents* unless other meaning is specifically stated) we consider are *well-formed* XML documents [4]. We call *XML string* a string representation of a document.

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

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**.

THEOREM 1. *Given a DTD T and a root element r, the set of XML documents with root r that are potentially valid w.r.t. T is **context-free**.*

Intuitively, to construct the context-free grammar (CFG) recognizing potential validity of XML documents w.r.t. a specific DTD T, we first construct the CFG for simple validity, G_T, and then, for each rule of the form $A \longrightarrow$ `<x>`RHS`</x>` add a new rule $A \longrightarrow RHS$ to the new grammar, G'_T. The details can be found in [10], but are omitted from this paper due to space considerations.

3. ALGORITHMS

Extended context free grammars (or regular right part grammars) were defined a long time ago [9] 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], just to name two). 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 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 but they exhibit poor performances for practical applications of checking potential validity.

As it turns out, however, the family G'_T of grammars for recognizing potential validity, possesses a number of properties, that allow us to develop a fast, linear-time parsing algorithm. To construct a fast linear recognizer for the class G'_T 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[4] is recognized by the grammar G'_T. We start by defining a reachability graph for the DTD elements.

Definition 5. 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 $t_1\} \cup \{(\#PCDATA, EMPTY)\} \cup \{(ANY, t) | t \in V\}$.

The reachability graph of T tells us whether or not 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 there exists a path from t_1 to t_2 in R_T, and $L_T(t_1, t_2) = false$ otherwise. For instance, the potential validity for text insertion[5] as content of an element t is $O(1)$ using L_T: just check whether or not $\#PCDATA$ is reachable from t.

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. It is known that given a CFG, the set of its usable nonterminals can be efficiently constructed [8]. 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 [8] to prune the unusable elements from the DTD. From now on we consider only the DTDs where all XML elements are usable.

The following important property of the usable DTD elements is important for us[6].

THEOREM 2. *Let $T = \langle \Gamma, \mathcal{T} \rangle$ be a DTD and $a \in \mathcal{T}$. Let A be the nonterminal in G'_T that corresponds to a. Then, $A \Rightarrow^*_{G'_T} \epsilon$.*

A first result that allows us to implement an efficient recognizer follows from Theorem 2.

COROLLARY 1. *Let $T = \langle \Gamma, \mathcal{T} \rangle$ be a DTD. Then $(\forall a, b \in \mathcal{T})(A \Rightarrow^*_{G'_T} B$ **iff** b is* reachable *from a in R_T) (where A and B are nonterminals of G'_T corresponding to the elements a and b of $\mathcal{T}$).*

Constructing a Simplified Grammar for Potential Validity.

In Section 2 we have described the construction of an Extended CFG G'_T 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[7]. In the following, for a given $T = \langle \Gamma, \mathcal{T} \rangle$ and a root element $r \in \mathcal{T}$, we give a set of properties of G'_T that allows us to reduce G' complexity without altering the grammar language.

PROPOSITION 1. *Let $T' = \langle \Gamma', \mathcal{T} \rangle$ be a DTD obtained from T by removing all occurrences of the ? operator from Γ. Then $L(G'_T) = L(G'_{T'})$.*

PROPOSITION 2. *Let $T' = \langle \Gamma', \mathcal{T} \rangle$ be a DTD obtained from T by replacing all occurrences of the + operator from Γ with $*$ operator. Then $L(G'_T) = L(G'_{T'})$.*

[4]A token is of type *start tag*, *end tag*, or *text* and we let *type*(X) to return the type of token X.

[5]Text deletion preserves potential validity [10].

[6]Throughout the paper we denote a *derivation* of a CFG G by $\Rightarrow_G$, and *zero or more* derivation steps by $\Rightarrow^*_G$ [1].

[7]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.

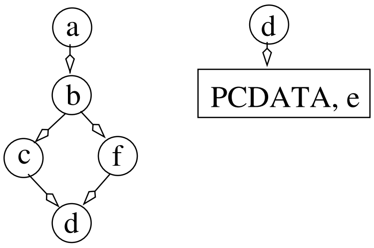

Figure 2: DAGs for Elements Type Declaration of elements a and d in the DTD in Figure 1

The intuition behind the algorithm for parsing the grammar $G'_{T,r}$ is captured in the data structure described next, which we use to model the DTD.

A DAG model for DTD

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

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* nodes. A simple element node corresponds to an element in the right-hand side of the Element Type Declaration for which no $*$ operator 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 $*$ operator is applied. Grouping elements in a choice-star node is to be done in a greedy manner, so that only the outmost $*$ operator is 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 2 (choice-star nodes are represented as boxes).

Algorithm **FastPV**

We can now describe the **Algorithm FastPV** for checking potential validity. Given a DTD T, a root element r and an XML document w, we solve the potential validity problem (**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.

```
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 3: The FastPV algorithm

3. Input tokenization: Given w, construct $\delta(w)$.
4. FastPV: Run **Algorithm FastPV** on DAG_T, r, L_T and $\delta(w)$ as inputs.

The pseudo-code for **Algorithm FastPV** is shown in Figure 3. **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

The problem of *element content potential validity* (**Problem ECPV**) is defined as follows: given a DTD T and an XML string $w = <a>w'$, $\delta(w') = X_1, X_2, \ldots, X_n$, we want to determine whether $A \Rightarrow^*_{G'} X_1, X_2, \ldots, X_n$, where A is the non-terminal in G' corresponding to the tag $< a >$.

An element content recognizer **ECRecognizer** used in the **Algorithm FastPV** solves the **Problem ECPV**. 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 4.

```
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 4: The ECRecognizer algorithm

The idea behind using **ECRecognizer** instances in the algorithm **FastPV** is as follows. Consider an XML fragment with tokenized version $<a>\sigma\sigma\sigma<c>\sigma</c>\sigma$. When parsing the tokens from left to right, we will first be able to establish the potential validity of the fragment σ. Once it is established, we may replace this fragment with $$, knowing that the content of b has already been derived. Similarly, we will do the same thing for the c element. Once we see token $$, our current fragment will look $<a>\sigma\sigma<c></c>\sigma$. 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 $<a>$ 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 by keeping a node active until the input token is not validated by the active node.

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). The size of the grammar G'_T recognizing potential validity is $p(k)$ where p is some polynomial[8]. 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**:

THEOREM 3. *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 in Java the algorithms described in Section 3.

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 tested 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 [9]. 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. For each DTD we have generated XML instance documents for six sizes: 1000,

[8] This statement is given without proof here, as the grammar construction had been omitted from the paper.

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

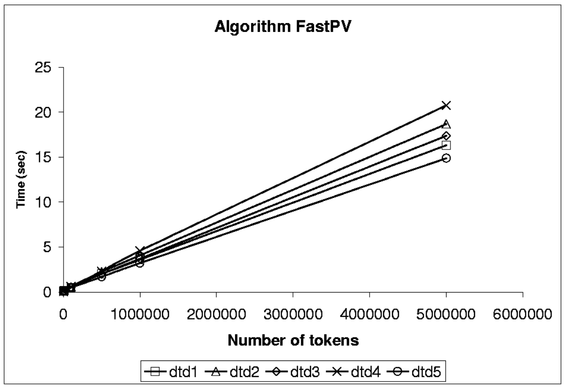

Figure 5: Test Results.

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.
The tests 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 and 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. For comparison, we implemented Earley's algorithm [7]. It did not scale well at all, running out of memory on 1,000,000 tokens and taking hours for documents with 500,000 tokens.

5. CONCLUSIONS

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.
Potential validity checking enforces relaxed schema constraints for XML documents. Incremental validation [11, 3, 2] of XML documents is a computational time-saving solution for validation of document updates, an efficient alternative to whole document validation. However, incremental validation considers an initially *valid* document together with a set of update operations and decides 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. 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 given in [5]. However, that work is based on the *unambiguity* property of the XML right-hand-side grammar productions [4], 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.

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.

6. REFERENCES

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

[2] D. Barbosa, A. O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient Incremental Validation of XML Documents. In *ICDE*, pages 671–682, April 2004.

[3] B. Bouchou and M. H. F. Alves. Updates and Incremental Validation of XML Documents. In *DBPL*, 2003.

[4] 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.

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

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

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

[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, A. Dekhtyar, and M. Dekhtyar. Checking Potential Validity of XML Documents. Technical Report TR 403-04, May 2004. http://www.cs.uky.edu/~dekhtyar/publications/TR403-04.pvalidity.ps.

[11] Y. Papakonstantinou and V. Vianu. Incremental validation of xml documents. In *ICDT*, pages 47–63. Springer-Verlag, 2002.

[12] 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.