Visibly Pushdown Transducers for Approximate Validation of Streaming XML

by

Ying Ying (Fay) Ye
B.Sc University of Victoria 2007

A Thesis Submitted in Partial Fullfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Visibly Pushdown Transducers for Approximate Validation of Streaming XML

by

Ying Ying (Fay) Ye
B.Sc University of Victoria 2007

Supervisory Committee

Dr. Venkatesh Srinivasan (Department of Computer Science)
Supervisor

Dr. Alex Thomo (Department of Computer Science)
Supervisor

Dr. Ulrike Stege (Department of Computer Science)
Departmental Member

Dr. Gary MacGillivray (Department of Mathematics and Statistics)
External Examiner

Supervisory Committee

Dr. Venkatesh Srinivasan (Department of Computer Science)
Supervisor

Dr. Alex Thomo (Department of Computer Science)
Supervisor

Dr. Ulrike Stege (Department of Computer Science)
Departmental Member

Dr. Gary MacGillivray (Department of Mathematics and Statistics)
External Examiner

Abstract

Visibly Pushdown Languages (VPLs), recognized by Visibly Pushdown Automata (VPAs), are a nicely behaved family of context-free languages. It has been shown that VPAs are equivalent to Extended Document Type Definitions (EDTDs), and thus, they provide means for elegantly solving various problems on XML. One of the important problems about XML that can be addressed using VPAs is the validation problem in which we need to decide whether an XML document con- forms to the schema specification given by an EDTD. In this thesis, we are interested in solving the approximate version of this problem, which is to decide whether an XML document can be modified by a tolerable number of edit operations to yield a valid one with respect to a given EDTD. For this, we define Edit Visibly Pushdown Transducers (EVPTs) that give us the framework for solving this problem ([23]). We propose two algorithms; the first algorithm solves the approximate validation for any EDTD in PTIME. The second algorithm solves the same problem for an interesting subclass of EDTDs (those rec- ognizable by FSA) in constant space using only a single pass over the document.

Table of Contents

Acknowledgements

I would like to express my deepest gratitude, first and foremost, to my supervisors, Dr. Venkatesh Srinivasan and Dr. Alex Thomo, for their leadership, understanding, and patience throughout my graduate career. I appreciate their expertise in the area, which provided insights that guided and challenged my thinking, also their kind support and assistance in writing this thesis.

I would also like to acknowledge Dr. Ulrike Stege, as well as Dr. Gary MacGillivray from the Math Department for taking time out from their busy schedules to serve as my committee members.

Special thanks go towards my family for the support they provided me through my entire life. Without their love, patience, and encouragement I would never have been able to complete my educational pursuits.

Chapter 1

Introduction

The eXtensible Markup Language (XML) is the lingua franca for data and document exchange on the Web and used in a variety of applications ranging from collaborative commerce to medical databases. An example of an XML document about contact information is given in Fig. 1.1.

```
<contact>
  <address>
    <str>...</str>
    <city>...</city>
  </address>
  <phone>...</phone>
</contact>
```

Fig. 1.1. Example of an XML document.

One of the most important problems on XML is the validation of documents against a schema specification typically given by one of the popular schema languages,

Document Type Definition (DTD), XML Schema ([19]) or Relax NG ([8]). Due to its importance, this problem has received a lot of attention (cf. [18, 22, 4, 6, 5, 21]).

Regarding the schema for XML, a recent development is the use of Visibly Pushdown Automata (VPAs) for representing specifications for XML. VPAs precisely capture the languages of XML documents induced by Extended Document Type Definitions (EDTDs), introduced in [17][1]. EDTDs are essentially extended context free grammars enriched with types and can model all three popular schema formalisms mentioned above: DTD, XML Schema and Relax NG (see [17, 16]). After constructing a VPA for a given EDTD (cf. [18]), the XML validation problem reduces to the one of accepting or rejecting an XML formatted word with the constructed VPA. We note here that the correspondence of VPAs to EDTDs is with respect to the word-encoded derivation trees of EDTDs rather than the set of words they generate. We remark that, when one uses an EDTD as a specification for XML documents, it is its language of derivation trees that is relevant; namely, for an XML document to be valid, when viewed as a tree, it has to correspond to a derivation tree of the given EDTD.

[1] [17] calls this formalism *specialized DTD* as types specialize tags. Similarly with [16], we use the term *extended DTD* to convey that this formalism is more powerful than DTD.

VPAs, which as mentioned, precisely capture XML schema specifications given by EDTDs, are in essence pushdown automata. Their push or pop mode can be determined by looking at the input only (hence their name). VPAs recognize Visibly Pushdown Languages (VPLs), which form a well-behaved and robust family of context-free languages. VPLs enjoy useful closure properties and several important problems for them are decidable. For example, VPLs are closed under intersection and complement, and the containment problem is decidable.

In this thesis, we introduce Edit Visibly Pushdown Transducers (EVPTs), which preserve the VPL family under their transductions. That is, given a VPL L and an EVPT T, the transduction of L through T is again a VPL. Notably, EVPTs give us a framework for solving the approximate validation of XML, where the approximation is in the sense that an XML document might not conform in its current form to a schema but will do so after a few edit operations. Formally, in this thesis, we study the K-validation problem for XML:

> Given a schema and a positive integer K, answer the following question: "Does an XML document fit the schema after at most K edit operations?"

The edit operations on XML generalize the standard edit operations on strings, which are the substitution of a symbol by another, deletion of a symbol, and in-

sertion of a symbol. In the context of XML, these operations are generalized to be substitution, deletion and insertion of pairs of matching open and close tags.

We construct EVPTs for each of the three edit operations on XML, and then superimpose them to produce a combined EVPT for all the operations. Here we remark that our EVPTs do not allow "pure" deletions, but rather implement the deletions by substitutions with special marker symbols which we call "tombstones." This is a technicality that makes the transduction preserve the VPL class.

We use the combined EVPT to design two algorithms that solve the approximate validation problem in two different settings.

The first algorithm shows that the approximate validation problem for a given schema A and an XML document w can be solved in PTIME in the standard RAM model. Let B be the automaton obtained by transducing A through the combined EVPT. Let A_w be the automaton that accepts exactly the words v which yield w after the tombstones (if any) are deleted. The main idea behind this algorithm (and the next) is the observation that, given a schema automaton A and a document w, checking whether w is "close" to a string in $L(A)$ is the same as checking if $L(B)$ has a non-empty intersection with $L(A_w)$. It checks this by first constructing the Cartesian product of B and A_w and then checking for its emptiness.

The second algorithm focuses on solving the K-validation problem in the model of streaming data. It aims at using only constant space and a single pass over the data. Since VPAs in general use a stack and hence require storage space proportional to the depth of the document, this algorithm focuses on solving the approximate validation problem for an interesting subclass of EDTDs, those recognizable using Finite State Automata (FSA).

Here, in line with the streaming XML model presented in [18], we distinguish two phases of the algorithm, which we explicitly call: the *preprocessing phase* and the *querying phase*. The preprocessing phase can be done offline and it has to be such as to facilitate the next phase of querying which in turn has to be done online and in a single pass on the streaming XML.

In the *preprocessing phase*, we transduce the given FSA A through the combined EVPT to get a new FSA B. The preprocessing phase will store this FSA as its final output.

In the *querying phase*, we receive as input a streaming XML document w and check if $L(B) \cap L(A_w)$ is empty or not. For this we present an algorithm which is able to decide the emptiness of this intersection using constant space and performing only a single pass on the document.

The rest of the thesis is organized as follows. In Chapter 2, we discuss related work. Chapter 3 reviews VPAs. In Chapter 4, we introduce EVPTs, their transductions and operations on them. In Chapter 5, we present EVPTs for edit operations separately and glue them together to get a combined EVPT. In Chapter 6, we introduce the alignment distance for XML and show a relation between checking alignment distance and checking emptiness of $L(B) \cap L(A_w)$. In Chapter 7, we give a PTIME algorithm for checking approximate validation in the standard RAM model. In Chapter 8, we present the class of EDTDs which are recognizable by FSAs. In Chapter 9, we present a streaming algorithm solving the approximate validation problem for the subclass of EDTDs recognized by FSAs. Finally, Chapter 10 concludes the thesis.

Chapter 2

Related Work

The XML validation problem has received a lot of attention in the recent years (cf. [18, 21, 7, 20, 9, 4, 6, 5]).

The first two, [18, 21], study the exact validation of XML in a streaming context.

The next three works, [7, 20, 9], consider variants of approximate XML validation, but in a non-streaming setting.

[7] presents a randomized methodology for validating and repairing XML documents. We note that [7] considers edit distance with *moves* and the error is relative rather than absolute. This means that the bigger the document is the bigger the error to be tolerated is. We believe that there are practical cases when an absolute tolerable error must be specified as opposed to a relative one.[1]

[20] presents an exact algorithm, for validating and repairing XML documents.

[1] For example, suppose that there is a schema for XML documents about (people) contact information. Now, following [7], if a contact XML file has a mailing address and a phone number then we would tolerate more structural errors than for some other contact file with only the mailing address. We believe that in this case, one should use an absolute number of tolerable errors in order to not bias the tolerance of validation towards the first file.

Regarding [9], it focuses on validating and repairing XML documents under a set of integrity constraints. The general problem for [9] is undecidable, and thus, it restricts the edit operations to either deletions or insertions only. All [7, 20, 9] consider simple DTDs only, while we consider VPAs which computationally represent EDTDs which in turn can abstract DTD, XML Schema and Relax NG.

The other three works, [4, 6, 5], consider the incremental validation of XML, which is validating documents after updates are being applied on them. The challenge there is to not rescan the document from the scratch, but rather work on the relevant (updated) part of the document. Also, the validation sought is exact rather than approximate. Although these works consider operations that edit (update) documents, the studied problem is very different from the approximate validation of XML.

Our treatment of approximate XML validation bears some similar flavor with [10, 11, 12]. However, these works deal with regular languages only and revolve around a different problem, which is finding paths in graph databases that approximately spell words in a given regular language.

Chapter 3

Visibly Pushdown Automata

VPAs were introduced in [3] and are a special case of pushdown automata. Their alphabet is partitioned into three disjoint sets of call, return and local symbols, and their push or pop behavior is determined by the consumed symbol. Specifically, while scanning the input, when a call symbol is read, the automaton pushes one stack symbol onto the stack; when a return symbol is read, the automaton pops off the top of the stack; and when a local symbol is read, the automaton only moves its control state.

Formally, a *visibly pushdown automaton* (VPA) A is a 6-tuple $(Q, (\Sigma, f), \Gamma, \tau, q_0, F)$, where

1. Q is a finite set of states.
2. – Σ is the alphabet partitioned into the (sub) alphabets Σ_c, Σ_l and Σ_r of call, local and return symbols respectively.

- f is a one-to-one mapping $\Sigma_c \rightarrow \Sigma_r$. We denote $f(a)$, where $a \in \Sigma_c$, by $\bar{a}$, which is in Σ_r.[1]

3. Γ is a finite stack alphabet that (besides other symbols) contains a special "bottom-of-the-stack" symbol $\perp$.
4. q_0 is the initial state.
5. F is the set of final states.
6. $\tau = \tau_c \cup \tau_r \cup \tau_l \cup \tau_\epsilon$ is the transition relation and τ_c, τ_l, τ_r and τ_ϵ are as follows.
   - $\tau_c \subseteq Q \times \Sigma_c \times Q \times \Gamma$
   - $\tau_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$
   - $\tau_l \subseteq Q \times \Sigma_l \times Q$
   - $\tau_\epsilon \subseteq Q \times \{\epsilon\} \times Q$

When reasoning about XML structure and validity, the local symbols are not important, and thus, for simplicity we will not mention local symbols in the rest of the thesis. So, for the above definition, we can consider Σ being partitioned into Σ_c and Σ_r, and τ being partitioned into τ_c, τ_r and τ_ϵ only.

Any transition involves two states (not necessarily distinct). We call the first the *origin state* and the second the *destination state*.

[1] When referring to arbitrary elements of Σ_r, we will use $\bar{a}, \bar{b}, \ldots$ in order to emphasize that these elements correspond to $a, b, \ldots$ elements of Σ_c.

Two transitions are called *consecutive* if the destination state of the first is the same as the origin state of the second. This definition applies regardless of whether the transitions involve a push or a pop.

A sequence of consecutive transitions is an *accepting run* if

1. The origin state of the first transition is q_0,
2. The destination state of the last transition is in F and
3. When starting with an empty stack ($\perp$) and following all the transitions in order,in the end, we get again an empty stack ($\perp$).

A word w is accepted by a VPA if there is an accepting run in the VPA which spells w. A language L is a *visibly pushdown language* (VPL) if there exists a VPA that accepts all and only the words in L. The VPL accepted by a VPA A is denoted by $L(A)$.

Example 1. Suppose that we want to build a VPA accepting XML documents about book collections. Such documents will have a *collection* element nesting any number of *book* elements in them. Each *book* element will nest a *title* element and any number of *author* elements. A VPA accepting well-formed documents of this structure is $A = (Q, (\Sigma, f), \Gamma, \tau, q_0, F)$, where

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\},$$

$$\Sigma = \Sigma_c \cup \Sigma_r =$$

$$\{collection, book, author, title\} \cup \{\overline{collection}, \overline{book}, \overline{author}, \overline{title}\},$$

f maps the Σ_c elements into their "bar"-ed counterparts in Σ_r,

$$\Gamma = \{\gamma_c, \gamma_b, \gamma_a, \gamma_t\} \cup \{\bot\},$$

$$F = \{q_8\},$$

$$\tau = \{(q_0, collection, q_1, \gamma_c), (q_1, book, q_2, \gamma_b), (q_2, author, q_3, \gamma_a),$$

$$(q_3, \overline{author}, \gamma_a, q_4), (q_4, author, q_3, \gamma_a), (q_4, title, q_5, \gamma_t),$$

$$(q_5, \overline{title}, \gamma_t, q_6), (q_6, \overline{book}, \gamma_b, q_7), (q_7, \overline{collection}, \gamma_c, q_8), (q_7, \epsilon, q_1)\}.$$

We show this VPA in Fig. 3.1.

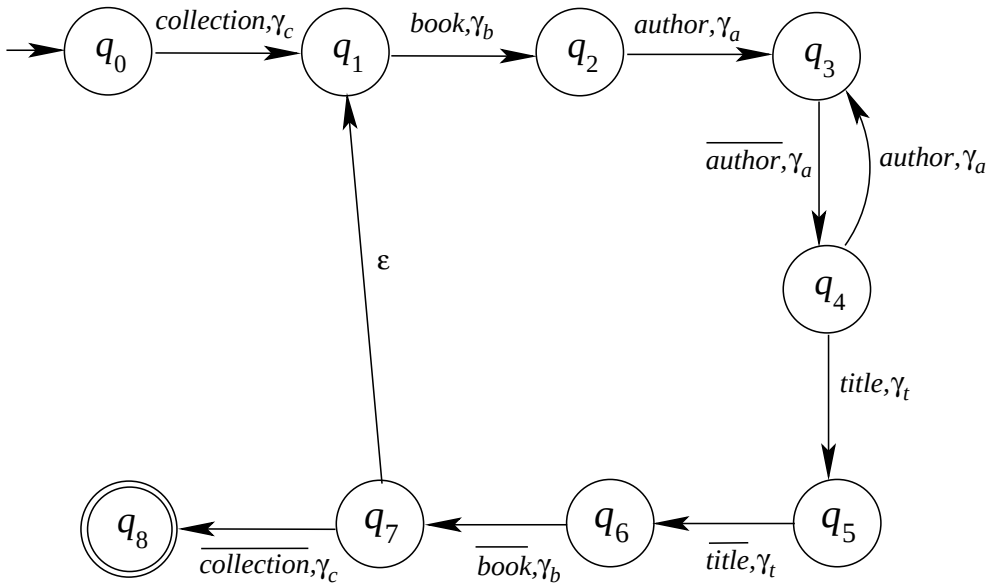

Fig. 3.1. Example of a VPA

Chapter 4

Edit Visibly Pushdown Transducers

An *edit visibly pushdown transducer* (EVPT) T is a 7-tuple

$$(P, (I, f), (O, g), \Gamma, \tau, p_0, F),$$

where

1. P is a finite set of states.
2. – I is the input alphabet partitioned into the (sub) alphabets I_c and I_r of input call and return symbols.
   – f is a one-to-one mapping $I_c \to I_r$. We denote $f(a)$, where $a \in I_c$, by $\bar{a}$.
3. – O is the output alphabet containing special symbols † and $\bar{\dagger}$ (used for deletion).
   – O is partitioned into (sub) alphabets O_c and O_r of output call and return symbols respectively.
   – g is a one-to-one mapping $O_c \to O_r$. We denote $g(b)$, where $b \in O_c$, by $\bar{b}$.
4. Γ is a finite stack alphabet that (besides other symbols) contains a special "bottom-of-the-stack" symbol $\perp$.
5. p_0 is the initial state.

6. F is the set of final states.

7. $\tau = \tau_c \cup \tau_r \cup \tau_\epsilon$, where

   - $\tau_c \subseteq (P \times I_c \times O_c \times P \times \Gamma) \cup (P \times \{\epsilon\} \times O_c \setminus \{\dagger\} \times P \times \Gamma) \cup (P \times I_c \times \{\dagger\} \times P \times \Gamma)$
   - $\tau_r \subseteq (P \times I_r \times O_r \times \Gamma \times P) \cup (P \times \{\epsilon\} \times O_r \setminus \{\bar{\dagger}\} \times \Gamma \times P) \cup (P \times I_r \times \{\bar{\dagger}\} \times \Gamma \times P)$
   - $\tau_\epsilon \subseteq P \times \{\epsilon\} \times \{\epsilon\} \times P$.

Observe that in the above definition, we do not allow transitions with ϵ as output. When such transducers "delete" a symbol, they in fact substitute the "deleted" symbol by $\dagger$ or $\bar{\dagger}$ depending on whether the symbol is a call or return one.

We define an *accepting run* for T similarly as for VPAs. Now, given a word $u \in I^*$, we say that a word $w \in O^*$ is an *output of T for u* if there exists an accepting run in T spelling u as input and w as output.[1]

A transducer T might produce more than one output for a given word u. We denote the set of all outputs of T for u by $T(u)$. For a language $L \subseteq I^*$, we define the *image of L through T* as

$$T(L) = \bigcup_{u \in L} T(u).$$

[1] In other words, we get u and w when concatenating the transitions' input and output components respectively.

If language L is a VPL, then we show that $T(L)$ is a VPL as well. To show this, let

$$A = (Q, (\Sigma^A, f^A), \Gamma^A, \tau^A, q_0, F^A)$$

be a VPA accepting L, and

$$T = (P, (I, f^T), (O, g^T), \Gamma^T, \tau^T, p_0, F^T)$$

be an EVPT as above, where $I \supseteq \Sigma^A$ and f^T is an extension of f^A. Then, we present a construction to obtain a VPA B, whose accepting language is $T(L)$, showing thus that the image of L through T is again a VPL.

The construction is a Cartesian product of A and T and similar in spirit to the construction of [3] for showing the closure of VPLs under intersection.

Specifically,

$$B = (R, (\Sigma^B, g^B), \Gamma^B, \tau^B, r_0, F^B),$$

where

1. $R = Q \times P$,
2. $\Sigma^B \subseteq O$, and g^B is a refinement of g^T
3. $\Gamma^B \subseteq (\Gamma^A \cup \{\diamond\}) \times \Gamma^T$, where $\diamond$ is a special symbol not in Γ^A and Γ^T.
4. $r_0 = (q_0, p_0)$,
5. $F^B = F^A \times F^T$,

6. $\tau^B = \tau_c^B \cup \tau_r^B$, where

$$\begin{aligned}
\tau_c^B = &\{(q,p), b, (q',p'), (\gamma^A, \gamma^T)) : (q, a, q', \gamma^A) \in \tau_c^A \text{ and } (p, a, b, p', \gamma^T) \in \tau_c^T\} \cup \\
&\{((q,p), \dagger, (q',p')), (\gamma^A, \gamma^T) : (q, a, q', \gamma^A) \in \tau_c^A, a \neq \epsilon \text{ and } (p, a, \dagger, p', \gamma^T) \in \tau_c^T\} \cup \\
&\{((q,p), b, (q,p'), (\diamond, \gamma^T)) : q \in Q \text{ and } (p, \epsilon, b, p', \gamma^T) \in \tau_c^T\} \\
\tau_r^B = &\{((q,p), \bar{b}, (\gamma^A, \gamma^T), (q',p')) : (q, \bar{a}, \gamma^A, q') \in \tau_r^A \text{ and } (p, \bar{a}, \bar{b}, \gamma^T, p') \in \tau_r^T\} \cup \\
&\{((q,p), \bar{\dagger}, (\gamma^A, \gamma^T), (q',p')) : (q, \bar{a}, \gamma^A, q') \in \tau_r^A, a \neq \epsilon \text{ and } (p, \bar{a}, \bar{\dagger}, \gamma^T, p') \in \tau_r^T\} \cup \\
&\{((q,p), \bar{b}, (\diamond, \gamma^T), (q,p')) : q \in Q \text{ and } (p, \epsilon, \bar{b}, \gamma^T, p') \in \tau_r^T\}
\end{aligned}$$

Clearly, B is a VPA, and we can show that

Theorem 1. *The language accepted by B is the image of L through T, i.e. $L(B) = T(L)$.*

Proof. An EVPT T can be considered as two VPAs; the *input* VPA A_{T_I} and the output *output* VPA A_{T_O}. A_{T_I} and A_{T_O} can be obtained from T by ignoring the output and input parts, respectively, of the transitions of T. A_{T_I} and A_{T_O} have the same structure; each transition path in A_{T_I} has some corresponding transition path in A_{T_O} and vice versa.

Now, the construction of VPA B computes the Cartesian product of VPA A with VPA A_{T_I}, but instead of keeping the matched transitions, it replaces them by the corresponding transitions in A_{T_O}.

Thus, if B accepts a word w, it means that there exists a corresponding word u accepted by A and A_{T_I}, such that $w \in T(u)$. As $u \in L$, we have that $T(u) \subseteq T(L)$ and $w \in T(L)$.

On the other hand, for a word u in $T(L)$, there exists some accepting transition path in the Cartesian product of A with A_{T_I}. By the construction of B, this accepting path induces an accepting path in B as well. Let w be the word spelled out by such a path in B. We have that $w \in L(B)$, and this concludes our proof. □

Superimposition of EVPTs.

Given two edit EVPTs,

$$T_1 = (P, (I, f), (O, g), \Gamma_1, \tau_1, p_0, F)$$

and

$$T_2 = (P, (I, f), (O, g), \Gamma_2, \tau_2, p_0, F),$$

which are the same except for the stack alphabet and transition relation, their superimposition EVPT is

$$T = (P, (I, f), (O, g), \Gamma_1 \cup \Gamma_2, \tau_1 \cup \tau_2, p_0, F).$$

In the rest of the thesis, we will work on building transducers for preprocessing a given schema specification A, transducing it into a "wider" schema B, which captures all

the words obtainable by applying at most K edit operations on the words captured by A.

Chapter 5

EVPTs for Edit Operations on XML

Since XML documents are nested, when we edit one call element, we also need to edit the corresponding return element. Thus, we consider an (XML) edit operation to consist of two single-symbol operations.

We want to build an EVPT, which given an input word u produces as output all the words v obtainable by applying not more than a certain number (say K) of edit operations on u. We define the edit operations as substitutions, deletions with tombstones, and insertions of call-return matches, and computationally represent them by using EVPTs.

Let $\Sigma = \Sigma_c \cup \Sigma_r$ be the underlying alphabet of XML documents. For the following subsections, we set

- $I = \Sigma$, $O = \Sigma \cup \{\dagger, \bar{\dagger}\}$,
- $I_c = \Sigma_c$, $O_c = \Sigma_c \cup \{\dagger\}$,
- $I_r = \Sigma_r$, $O_r = \Sigma_r \cup \{\bar{\dagger}\}$,
- $g|_{\Sigma_c} = f$ and $g(\dagger) = \bar{\dagger}$.

5.1 Substitution

A *call-return match substitution* replaces in an input word a call-return match $a, \bar{a}$ by another call-return match $b, \bar{b}$.

For example, consider the XML document given in Fig. 5.1 [left]. By substituting `<phone>`, `</phone>` by `<tel>`, `</tel>`, we obtain the document shown in Fig. 5.1 [right].

```
<contact>
  <address>
    <str>...</str>
    <city>...</city>
  </address>
  <phone>...</phone>
</contact>
```

```
<contact>
  <address>
    <str>...</str>
    <city>...</city>
  </address>
  <tel>...</tel>
</contact>
```

Fig. 5.1. Illustration of substitution.

In the following, given a non-negative integer K, we build an EVPT which for any word u produces as output the set of all the words w obtainable from u by applying at most K substitutions. We denote this transducer by $T_{\sigma}^{\leq K}$ and formally define it as an EVPT with

– $Q = \{q_0, q_1, q_2, \ldots, q_{2K}\}$,

- $\Gamma = \{\gamma_a : a \in \Sigma_c\} \cup \{\sigma_{ab} : a, b \in \Sigma_c, a \neq b, \} \cup \{\perp\}$,
- $F = \{q_{2i} : 0 \leq i \leq K\}$,
- $\tau = \tau_c \cup \tau_r$, where

$$\tau_c = \{(q_i, a, a, q_i, \gamma_a) : 0 \leq i \leq 2K \text{ and } a \in \Sigma_c\} \cup$$
$$\{(q_i, a, b, q_{i+1}, \sigma_{ab}) : 0 \leq i \leq 2K-1, a, b \in \Sigma_c \text{ and } a \neq b\},$$
$$\tau_r = \{(q_i, \bar{a}, \bar{a}, \gamma_a, q_i) : 0 \leq i \leq 2K \text{ and } \bar{a} \in \Sigma_r\} \cup$$
$$\{(q_i, \bar{a}, \bar{b}, \sigma_{ab}, q_{i+1}) : 1 \leq i \leq 2K-1, \bar{a}, \bar{b} \in \Sigma_r \text{ and } \bar{a} \neq \bar{b}\}.$$

For illustration, in Fig. 5.2, we show $T_{\sigma}^{\leq 2}$, for alphabet $\{a, b\} \cup \{\bar{a}, \bar{b}\}$.

Intuitively, the transitions in the first set of τ_c and in the first set of τ_r leave the consumed call and return symbols unchanged.

Regarding the transitions in the second set of τ_c, they substitute a call symbol, say a, by another call symbol, say b. A substitution marking symbol σ_{ab} is pushed onto the stack. Symbol σ_{ab} in the stack is crucial in determining which occurrence of $\bar{a}$ has to be replaced by $\bar{b}$ using a transition in the second set of τ_r.

Finally, since we want to substitute 0, 1, ..., K symbols, we need $K+1$ different final states for the $K+1$ different cases.

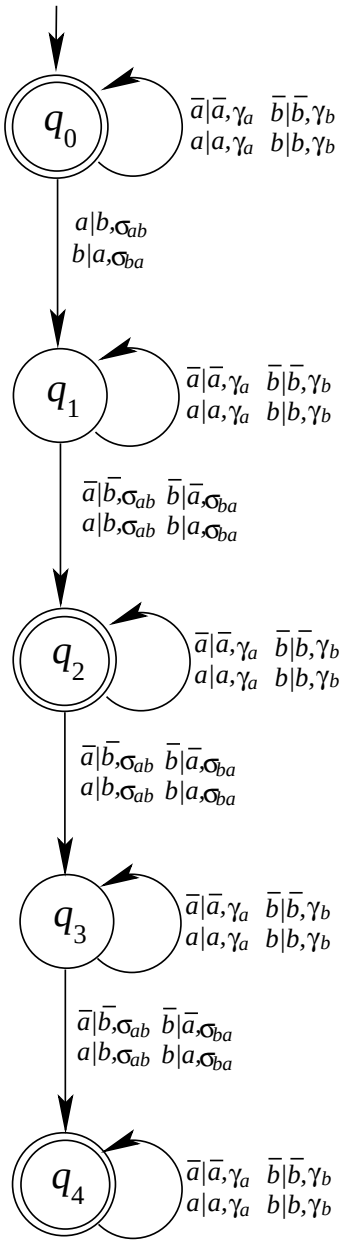

Fig. 5.2. EVPT $T_\sigma^{\leq 2}$

5.2 Deletion

A *call-return match deletion* removes in an input word a call-return match $a, \bar{a}$ by substituting them with tombstones † and $\bar{\dagger}$, respectively.

For example, consider the XML document given in Fig. 5.3 [left]. By deleting `<address>`, `</address>`, we obtain the document shown in Fig. 5.3 [right].

```
<contact>                    <contact>
  <address>                    <†>
    <str>...</str>               <str>...</str>
    <city>...</city>             <city>...</city>
  </address>                   <†̄>
  <phone>...</phone>           <phone>...</phone>
</contact>                   </contact>
```

Fig. 5.3. Illustration of deletion under the first semantics.

In the following, given a non-negative integer K, we build an EVPT which for any word u produces as output the set of all the words w obtainable from u by applying at most K deletions with tombstones. We denote this transducer by $T_{\bar{\delta}}^{\leq K}$ and formally define it as an EVPT with

- $Q = \{q_0, q_1, q_2, \ldots, q_{2K}\}$,
- $\Gamma = \{\gamma_a : a \in \Sigma_c\} \cup \{\delta_a : a \in \Sigma_c\} \cup \{\bot\}$,

– $F = \{q_{2i} : 0 \leq i \leq K\}$,

– $\tau = \tau_c \cup \tau_r$, where

$$\tau_c = \{(q_i, a, a, q_i, \gamma_a) : 0 \leq i \leq 2K \text{ and } a \in \Sigma_c\} \cup$$
$$\{(q_i, a, \dagger, q_{i+1}, \delta_a) : 0 \leq i \leq 2K-1, \text{ and } a \in \Sigma_c\},$$
$$\tau_r = \{(q_i, \bar{a}, \bar{a}, \gamma_a, q_i) : 0 \leq i \leq 2K \text{ and } \bar{a} \in \Sigma_r\} \cup$$
$$\{(q_i, \bar{a}, \bar{\dagger}, \delta_a, q_{i+1}) : 1 \leq i \leq 2K-1, \text{ and } \bar{a} \in \Sigma_r\}.$$

For illustration, in Fig. 5.4, we show $T_{\delta}^{\leq 2}$, for alphabet $\{a, b\} \cup \{\bar{a}, \bar{b}\}$.

Similarly with the substitution, the transitions in the first set of τ_c and in the first set of τ_r leave the consumed call and return symbols unchanged.

Regarding the transitions in the second set of τ_c, they delete a call symbol, say a, by substituting with tombstone †. A deletion marking symbol δ_a is pushed onto the stack. Symbol δ_a in the stack is crucial in determining which occurrence of $\bar{a}$ has to be deleted (substituting with tombstone $\bar{\dagger}$) by using a transition in the second set of τ_r.

Since we want to perform 0, 1, ..., K deletions, we need $K + 1$ different final states for the $K + 1$ different cases.

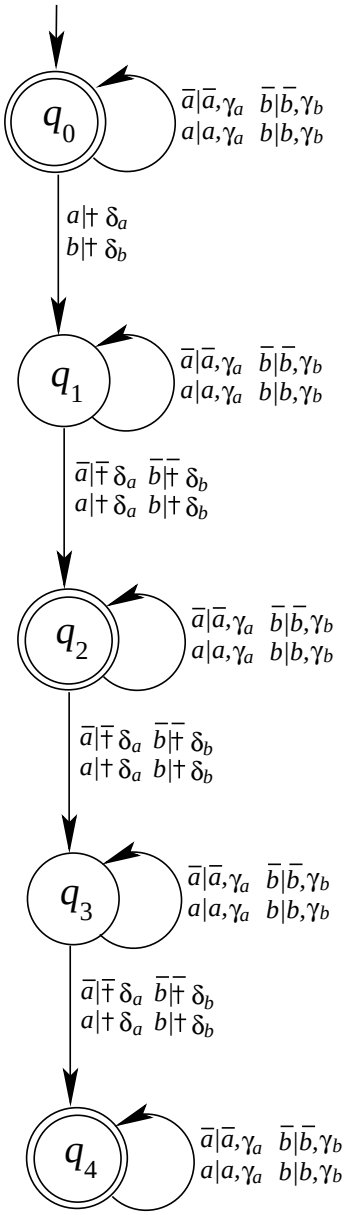

Fig. 5.4. EVPT $T_{\delta}^{\leq 2}$.

5.3 Insertion

A *call-return match insertion* inserts in an input word a call symbol a and a corresponding return symbol $\bar{a}$ while maintaining the well-formedness of the XML document. Thus, insertion is a "structure creation" operator.

For example, consider the XML document given in Fig. 5.5 [left]. By inserting `<address>`, `</address>`, surrounding the street and city elements, we obtain the document in Fig. 5.5 [right].

```
<contact>
  <str>...</str>
  <city>...</city>
  <phone>...</phone>
</contact>
```

```
<contact>
  <address>
    <str>...</str>
    <city>...</city>
  </address>
  <phone>...</phone>
</contact>
```

Fig. 5.5. Illustration of insertion under the first semantics.

In the following, given a non-negative integer K, we build an EVPT which for any word u produces as output the set of all the words w obtainable from u by applying at most K insertions. We denote this transducer by $T_{\eta}^{\leq K}$ and formally define it as an EVPT with

- $Q = \{q_0, q_1, q_2, \ldots, q_{2K}\}$,
- $\Gamma = \{\gamma_a : a \in \Sigma_c\} \cup \{\eta_a : a \in \Sigma_c\} \cup \{\perp\}$,
- $F = \{q_{2i} : 0 \leq i \leq K\}$,
- $\tau = \tau_c \cup \tau_r$, where

$$\begin{aligned}
\tau_c = &\{(q_i, a, a, q_i, \gamma_a) : 0 \leq i \leq 2K \text{ and } a \in \Sigma_c\} \cup \\
&\{(q_i, \epsilon, a, q_{i+1}, \eta_a) : 0 \leq i \leq 2K-1, \text{ and } a \in \Sigma_c\}, \\
\tau_r = &\{(q_i, \bar{a}, \bar{a}, \gamma_a, q_i) : 0 \leq i \leq 2K \text{ and } \bar{a} \in \Sigma_r\} \cup \\
&\{(q_i, \epsilon, \bar{a}, \eta_a, q_{i+1}) : 1 \leq i \leq 2K-1, \text{ and } \bar{a} \in \Sigma_r\}.
\end{aligned}$$

For illustration, in Fig. 5.6, we show $T_\eta^{\leq 2}$, for alphabet $\{a, b\} \cup \{\bar{a}, \bar{b}\}$.

Again, the transitions in the first set of τ_c, and in the first set of τ_r leave the consumed call and return (respectively) symbols unchanged.

Regarding the transitions in the second set of τ_c, they insert a call symbol, say a. An insertion marking symbol η_a is inserted on the stack. Symbol η_a in the stack is crucial in determining when to insert $\bar{a}$ by using a transition in the second set of τ_r.

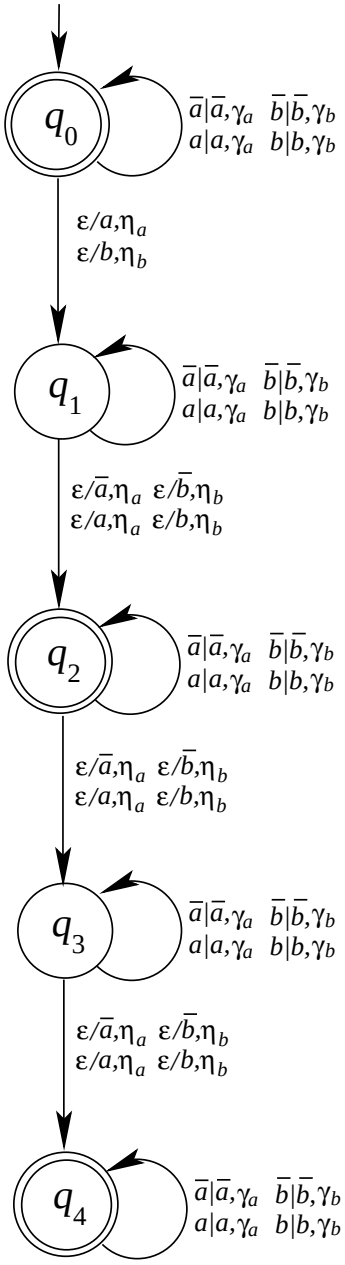

Fig. 5.6. EVPT $T_{\eta}^{\leq 2}$.

5.4 An EVPTs for All Operations

Here, for a given a non-negative integer K, we will construct an EVPT which for any word u produces as output a set of words w obtainable from u by applying at most K edit operations, which can be substitutions, deletions with tombstones or insertions. We will show that these words, when stripped off tombstones, constitute *all* the words of an "alignment distance" (to be defined soon) at most K from word u.

As can be observed from the previous section, sets Q, I, O and F are the same for all the transducers constructed so far. Notably, an EVPT $T^{\leq K}$ for at most K "non-overlapping" edit operations can be simply obtained by superimposing $T_{\sigma}^{\leq K}$, $T_{\delta}^{\leq K}$ and $T_{\eta}^{\leq K}$.

Transducer $T^{\leq K}$ has $2K + 1$ states and $O(KR^2)$ transitions, where R is the size of the underlying alphabet.

Chapter 6

Alignment Distance

Definition 1. *An* alignment *of two nested words u and w is obtained by first inserting call and return tombstones, either into or at the ends of u and w, preserving proper nesting, and then placing the two resulting words one above the other so that every call (return) symbol in one string is opposite a unique call (return) symbol in the other string.*

Definition 2. *The* cost *of an alignment is the number of positions where the corresponding call symbols differ.*

Definition 3. *The* alignment distance *between two words u and w, denoted by $d(u, w)$, is the cost of a cheapest alignment of u and w.*

Given, a language $L \subseteq \Sigma^*$ and a non-negative integer K, we define

$$L^{(K)} = \{u : \exists w \in L \text{ and } d(u, w) \leq K\}.$$

Now, we relate language $T^{\leq K}(L)$ with language $L^{(K)}$ for any $L \subseteq \Sigma$.

Let L' be a language on $O = \Sigma \cup \{\dagger, \bar{\dagger}\}$. We denote by $L'|_\Sigma$ the language we obtain after substituting $\dagger$ and $\bar{\dagger}$ with ϵ in the words of L'.

We can show that for transducer $T^{\leq K}$ obtained at the end of the previous chapter,

Theorem 1. *$T^{\leq K}(L)|_{\Sigma} = L^{(K)}$ for any language $L \subseteq \Sigma$.*

Proof. To show the "$\subseteq$" direction of the Theorem, we prove the following lemma.

Lemma 1. *If $(u, v) \in \Sigma^* \times \Sigma^*$ and there exists a string $v' \in O^*$ such that (1) $v = v'|_{\Sigma}$ and (2) $(u, v') \in R_{T^{\leq K}}$, then $d(u, v) \leq K$.*

Proof. Since $(u, v') \in R_{T^{\leq K}}$, there is an accepting run of the transducer $T^{\leq K}$ spelling u as input and v' as output. In other words, we get u and v' by concatenating the input and the output component of the transitions leading from the start to accept state. In the accepting run, some of the transitions might have ϵ as the input component. Construct a string $u' \in O^*$ by first replacing an ϵ by $\dagger$ if the transition belongs to τ_c and by $\bar{\dagger}$ if the transition belongs to τ_r and then concatenating all the input components of the accepting run. Then it is easy to check that $u'|_{\Sigma} = u$ and (u', v') is an alignment with cost at most K. This follows from the fact that any accepting run in $T^{\leq K}$ can have at most K call symbol mismatches. Hence we can conclude that $d(u'|_{\Sigma}, v'_{\Sigma}) \leq K$. That is, $d(u, v) \leq K$. □

To show the "$\supseteq$" direction of the Theorem, we show the following lemma.

Lemma 2. *Let $(u, v) \in \Sigma^* \times \Sigma^*$ and $d(u, v) \leq K$. Furthermore, let $u' \in O^*$ and $v' \in O^*$ be the strings corresponding to u and v respectively in the cheapest alignment of u and v. Then $(u, v') \in R_{T^{\leq K}}$.*

Proof. Consider the cheapest alignment (u', v') of u and v and let $|u'| = |v'| = m$. As noted above, u' and v' are well-nested and every call (return) symbol of u' is opposite a call (return) symbol of v' in this alignment. Moreover, the number of call symbol mismatches is at most K.

For any $1 \leq i \leq m$, let u'_i and v'_i denote the i-th symbol of u' and v', respectively. Now, we scan the (u', v') alignment from left to right and incrementally construct the accepting run of $T^{\leq K}$ as follows. Initially, $T^{\leq K}$ is in the start state q_0 before the first input symbol. As we perform the scan, we distinguish four cases with respect to the pairs of symbols in the alignment.

1. $u'_i = v'_i$ and $u'_i, v'_i \in \Sigma$:

   Let $T^{\leq K}$ be in state q_h before the ith symbol. Then add the transition $(q_h, u'_i, v'_i, q_h, \gamma_{u'_i})$ to the accepting run if u'_i and v'_i are call symbols and the transition $(q_h, u'_i, v'_i, \gamma_{u'_i}, q_h)$ to the accepting run if u'_i and v'_i are return symbols. Such (self-loop) transitions exist by the definition of $T^{\leq K}$.

2. $u'_i \neq v'_i$ and $u'_i, v'_i \in \Sigma$:

   Let $T^{\leq K}$ be in state q_h before the ith symbol. Then add the transition $(q_h, u'_i, v'_i, q_{h+1}, \sigma_{u'_i v'_i})$ to the accepting run if u'_i and v'_i are call symbols and the transition $(q_h, u'_i, v'_i, \sigma_{u'_i v'_i}, q_{h+1})$ to the accepting run if u'_i and v'_i are return symbols. Such (substitution) transitions exist by the definition of $T^{\leq K}$.

3. $u'_i = \dagger$ and $v'_i \in \Sigma_c$ or $u'_i = \bar{\dagger}$ and $v'_i \in \Sigma_r$:

   Let $T^{\leq K}$ be in state q_h before the ith symbol. Then add the transition $(q_h, \epsilon, v'_i, q_{h+1}, \eta_{u'_i})$ to the accepting run if $u'_i = \dagger$ and $v'_i \in \Sigma_c$ and the transition $(q_h, \epsilon, v'_i, \eta_{u'_i}, q_{h+1})$ to the accepting run if $u'_i = \bar{\dagger}$ and $v'_i \in \Sigma_r$. Such (insertion) transitions exist by the definition of $T^{\leq K}$.

4. $u'_i \in \Sigma_c$ and $v'_i = \dagger$ or $u'_i \in \Sigma_r$ and $v'_i = \bar{\dagger}$:

   Let $T^{\leq K}$ be in state q_h before the ith symbol. Then add the transition $(q_h, u'_i, \dagger, q_{h+1}, \delta_{u'_i})$ to the accepting run if $u'_i \in \Sigma_c$ and $v'_i = \dagger$ and the transition $(q_h, u'_i, \bar{\dagger}, \delta_{u'_i}, q_{h+1})$ to the accepting run if $u'_i \in \Sigma_r$ and $v'_i = \bar{\dagger}$. Such (deletion) transitions exist by the definition of $T^{\leq K}$.

We remark that the case $u'_i = v'_i = \dagger/\bar{\dagger}$ need not be considered as any alignment containing this can be shrunk to a one without it keeping the number of mismatches the same. Also note that the value of h above is at most $2K$ during the construction of the accepting run as there are at most $2K$ (call and return) symbol mismatches between u' and v' and hence a transition can always be added at every stage. □

Given an input document (word) w, we consider the following regular language:

$$L_w = \{w' \in (\Sigma \cup \{\dagger, \bar{\dagger}\})^* : w'|_\Sigma = w\}.$$

It easy to construct an FSA A_w recognizing L_w. An example is given in Fig. 6.1 for word $w = ab\bar{b}\bar{a}$. Formally,

$$A_w = (P, \{a, \bar{a}, b, \bar{b}, \dagger, \bar{\dagger}\}, \tau, p_0, F),$$

where

- $P = \{p_0, p_1, p_2, p_3, p_4\}$,
- $F = \{p_4\}$,
- $\tau = \tau_w \cup \tau_\dagger$, where

$$\tau_w = \{(p_0, a, p_1), (p_1, b, p_2), (p_2, \bar{b}, p_3), (p_3, \bar{a}, p_4)\},$$

$$\tau_\dagger = \{(p_i, \dagger, p_i), (p_i, \bar{\dagger}, p_i) \text{ such that } 0 \leq i \leq 4\}.$$

Then, we show that

Corollary 1. *Checking that a document w belongs to $L^{(K)}$ is equivalent to checking that $L_w \cap T^{\leq K}(L) \neq \emptyset$.*

Proof. Using the theorem above, checking that w belongs to $L^{(K)}$ is the same as checking that there is a string $w' \in (\Sigma \cup \{\dagger, \bar{\dagger}\})^*$ such that $w'|_\Sigma = w$ and $w' \in T^{\leq K}(L)$. The latter is the same as checking that $L_w \cap T^{\leq K}(L) \neq \emptyset$ by the definition of L_w. □

Based on the corollary above, it is sufficient for the query scheme to decide the non-emptiness of $L_w \cap T^{\leq K}(L)$ on the document w.

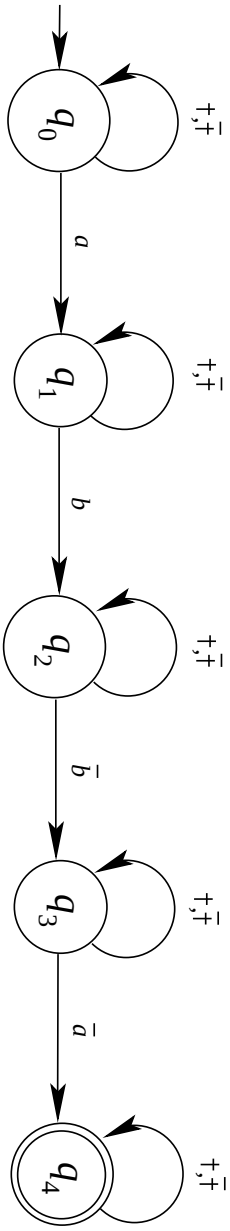

Fig. 6.1. A_w for $w = ab\bar{b}\bar{a}$.

Chapter 7

Validating XML in PTIME

If L is a VPL, $T^{\leq K}(L)$ is computable as we show in Chapter 4, and it is captured by a VPA B. The emptiness of $L(B) \cap L(A_w)$ can be decided in time polynomial in the size of w and B.

Thus, our contribution here is showing that the K-validation using alignment distance for XML is indeed possible in PTIME. Our algorithm starts by constructing the Cartesian product C of B and A_w. To check the emptiness of $L(C)$, it incrementally builds a set G that contains all pairs (p_i, p_j) such that there exists a properly nested word taking C from p_i to p_j starting and ending with empty stack. It finally checks if (p_0, p_f), for some final state p_f, is in G.

Algorithm 1

Input: A DFA A_w recognizing L_w and a VPA B recognizing $T^{\leq K}(L)$.

Output: "Yes" if $L(A_w) \cap L(B) \neq \emptyset$.

1. Construct Cartesian product $C = A_w \times B$.
2. $G = [(p_1, p_1), \ldots, (p_m, p_m)]$, where p_i, for $i \in [1, m]$ is a state in C.

3. Repeat the following steps until G does not change anymore.
   (a) Add a new pair (p_i, p_k) to G whenever one the following conditions is true:
      i. There exist transitions (p_i, a, p_j, γ) and $(p_l, \bar{a}, \gamma, p_k)$ in C, and $(p_j, p_l) \in G$,
      ii. There exist (p_i, p_j) and $(p_j, p_k) \in G$.
4. Output "yes" if there exists $(p_0, p_f) \in G$, where p_f is a final state in C, and "no" otherwise.

In the next chapter, we tackle the problem of K-validation in a streaming context, where we allowed to perform only a single pass on the document and use constant space not depended on the document. We remark that these requirements imply that the validation (even the exact version) *cannot* be performed by VPAs as they use stack space which depends on the document depth, i.e. not constant. Instead, a validation satisfying the streaming requirements needs to be performed by finite state automata (FSAs).

Chapter 8

DTDs and EDTDs Representable by FSAs

We wish to validate streaming XML in a single pass and using constant space. The latter means that the space needed for the validation should not depend on the streaming XML document.

For this to be possible, the language of the XML documents induced by an EDTD schema should be recognizable by a finite state automaton (FSA) rather than a general VPA which would need stack space proportional to the depth of the scanned document.

Thus, the question becomes:

Which schemas can be captured using FSAs?

This question is elegantly answered in [18] for EDTDs. In this chapter we present the necessary definitions for presenting the result of [18]. Then, we will make the connection with our EVPTs.

In order to talk about DTDs and then EDTDs let us first abstract XML documents by trees capturing the nesting structure of elements in the documents. For example,

the nested word

$$rabc\bar{c}\bar{b}bc\bar{c}\bar{a}ab\bar{b}bc\bar{c}\bar{a}\bar{r}$$

can be represented by the tree in Fig. 8.1.

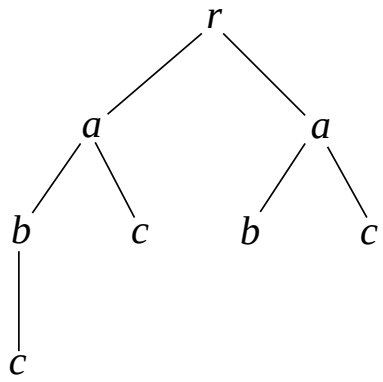

Fig. 8.1. An XML tree.

Going from trees to nested words is also easy. Let t be a node labeled tree. We denote by $[t]$ the corresponding nested word obtained inductively as follows.

1. If t is a single root node labeled by a, then $[t] = a\bar{a}$.
2. If t has a root and subtrees $t_1, \ldots, t_n$, then $[t] = a[t_1] \ldots [t_n]\bar{a}$.

If Tr is a tree set, we denote by $L(Tr)$ the language of nested words corresponding to the trees in Tr.

8.1 DTDs

In essence, DTDs are extended context-free grammars and formally defined as follows.

Definition 1. *Let Σ_c be the (finite) tag (call symbol) alphabet of a given XML collection. Then, a DTD D is a pair (R, r), where R is a function mapping Σ_c-symbols to regular expressions on Σ_c and r is the root symbol (cf. [15]).*

We denote by $R(a)$ both the regular expression and the corresponding regular language for a symbol $a \in \Sigma_c$.

Definition 2. *A* valid XML tree *complying to a DTD $D = (R, r)$ has a root which is labeled by r and every node labeled, say by a, has a sequence of children whose label concatenation, say $bc \dots x$, is in $R(a)$.*

The set of trees complying to a DTD D is denoted by $Tr(D)$.

Now, we define the language of the nested words corresponding to a DTD.

Definition 3. *Let D be a DTD. Then we define the associated language of properly nested words as*

$$L(D) = \{[t] : t \in \mathrm{Tr}(D)\}.$$

A simple example of a DTD defining the structure of some book collections is the following:

$$\text{collection} \longrightarrow book^*$$

$$\text{book} \longrightarrow author^{+}\ title$$

where "+" implies "one or more" and "∗" implies "zero or more" occurrences of the child element. In essence, a DTD D is an extended context-free grammar, and a valid XML document with respect to D is a parse tree for D.

8.2 EDTDs

Here we consider extended DTDs (EDTDs). They are motivated because DTDs have the following severe limitation: their definition of the type of a given tag depends only on the tag itself and not on the context in which it occurs.

For example, this means that, if we consider the set $Tr^{(1)}$ consisting only of the tree given in Fig. 8.1, then we cannot find a DTD describing it! This is because the type (language of children) of the first b differs from the type of the second b. Observe that, we cannot use the rule

$$b \longrightarrow c|\epsilon$$

because this would imply that $Tr^{(1)}$ has more than one tree.

The above leads naturally to an extension of DTDs with *specializations*. Intuitively, we want to allow defining the type of a tag by a number of cases depending on the context.

Definition 4. *An extended DTD over Σ_c is a quadruple*

$$D = (\Sigma_c, \Sigma_c', D', \mu)$$

where

1. *Σ_c and Σ_c' are finite alphabets,*
2. *D' is a DTD over Σ_c', and*
3. *μ is a mapping from Σ_c' to Σ_c.*

Intuitively, Σ_c' provides a set of specializations for the elements of Σ_c. Specifically, an element $a' \in \Sigma_c'$ is a specialization for $a \in \Sigma_c$, if $\mu(a') = a$.

For example, an EDTD for $Tr^{(1)}$ is

$$D = (\{r, a, b, c\}, \{r', a_1', a_2', b_1', b_2', c'\}, D', \mu),$$

where D' is

$$r' \longrightarrow a_1' a_2'$$

$$a_1' \longrightarrow b_1' c'$$

$$a_2' \longrightarrow b_2' c'$$

$$b_1' \longrightarrow c'$$

$$b_2' \longrightarrow \epsilon$$

$$c' \longrightarrow \epsilon,$$

and $\mu(r') = r$, $\mu(a'_1) = \mu(a'_2) = a$, $\mu(b'_1) = \mu(b'_2) = b$, and $\mu(c') = c$.

A tree t over Σ_c satisfies an *EDTD* D, if $t \in \mu(\mathit{Tr}(D'))$.

Now, we define the language of the nested words corresponding to an EDTD.

Definition 5. *Let D be a DTD or EDTD. Then we define the associated language of properly nested words as*

$$L(D) = \{[t] : t \in \mu(\mathrm{Tr}(D))\}.$$

8.3 Recognizable DTDs and EDTDs

Let D be a DTD (or EDTD) over Σ_c, and consider the associated language of nested words $L(D)$. Here we will present the result of [18] for characterizing the DTDs (or EDTDs) D for which $L(D)$ can be recognized by an FSA, i.e. $L(D)$ is regular.

Such DTDs (or EDTDs) are called *recognizable*.

We first illustrate the problem with the following examples.

Example 1. Consider the DTD D

$$r \longrightarrow a$$

$$a \longrightarrow a|\epsilon,$$

which defines the trees with root r and containing a single path of arbitrary length of nodes labeled a. Thus, $L(D) = \{ra^n\bar{a}^n\bar{r}|n \in \mathbb{N}\}$ which is not regular.

Example 2. Consider the DTD D

$$r \longrightarrow a^*$$

$$a \longrightarrow b|c$$

$$b \longrightarrow \epsilon$$

$$c \longrightarrow \epsilon.$$

It is easy to see that $L(D) = r(a(b\bar{b}|c\bar{c})\bar{a})^*\bar{r}$ which is regular. So, D is recognizable.

Now, we provide the complete characterization of [18] regarding the recognizable DTDs and EDTDs.

Definition 6. *Let D be a DTD over Σ_c and G_D the graph constructed as follows:*

1. *its set of vertices is Σ_c, and*
2. *for each rule $a \longrightarrow R(a)$ in D there is an edge from a to b for each b occurring in some word in $R(a)$.*

Call G_D the dependency graph of D. Now we define:

1. *Two symbols a and b are* mutually recursive *if they belong to some cycle of G_D.*
2. *A symbol a is* recursive *if it is mutually recursive with itself.*
3. *A DTD D is* non-recursive *iff G_D is acyclic.*
4. *An EDTD $D = (\Sigma_c, \Sigma'_c, D', \mu)$ is* non-recursive *iff DTD D' (over Σ'_c) is non-recursive.*

Now, from [18], we have that

Theorem 1. *An EDTD is recognizable iff it is non-recursive.*

Specifically, the construction of [18] to obtain an FSA from a non-recursive $EDTD$ is as follows.

Let $D = (\Sigma_c, \Sigma'_c, D', \mu)$ be a non-recursive EDTD. Without loss of generality we can assume that $\Sigma_c \cap \Sigma'_c = \emptyset$. For each $a' \in \Sigma'_c$ construct an FSA A_a recognizing $\mu(a')R_{a'}\mu(\bar{a'})$, where $R_{a'}$ is the regular expression associated to a' in D'.

An FSA A recognizing $L(D)$ is constructed inductively as follows. Let A_0 be $A_{r'}$, where r' is the root label. For $i \geq 0$, A_{i+1} is obtained by modifying A_i as follows. For each transition $e = (p, a', q)$ of A_i, where $a' \in \Sigma_c$

- add a copy A_e of $A_{a'}$,
- add the transitions (p, ϵ, s_e), where s_e is the start state of A_e, and (f_e, ϵ, q) for each accepting state f_e of A_e,
- remove e.

Because D is non-recursive this process is sure to terminate. Note that the resulting FSA is over the alphabet $\Sigma_c \cup \Sigma_r$. The above built FSA recognizes $L(D)$.

8.4 VPAs and EPVTs

As mentioned in the Introduction EDTDs can be precisely captured by VPAs. Namely, one can transform in PTIME an EDTD into a VPA and vice versa. Thus for an EDTD D, $L(D)$ is a VPL (see [3]).

Denote by $n_a(w)$, where $a \in \Sigma_c$ and $w \in \Sigma^* = (\Sigma_c \cup \Sigma_r)^*$, the greatest number of a's in subwords v of w, such that the first and last symbol of v is a, and there is no $\bar{a}$ in between.

Then, we show that,

Theorem 2. *An EDTD $D = (\Sigma, \Sigma', D', \mu)$ is non-recursive if and only if for each $a \in \Sigma_c$ there exist $n_a^{max} \in \mathbb{N}$, such that $n_a(w) \leq n_a^{max}$ for any $w \in L(D)$.*

Proof.

"if." For the sake of contradiction, suppose that D is recursive and let $a \in \Sigma'$ be a recursive symbol in D'. Hence there exists a tree t in $Tr(D')$ where a repeats along one path. The string $[t]$ is of the form $ru_1av_1aw\bar{a}v_2\bar{a}u_2\bar{r}$, where u_1u_2 and v_1v_2 are properly nested words, and a and $\bar{a}$ are not in v_1 and v_2, respectively. By iterating the recursive part of the derivation from a to a, we have that $ru_1(av_1)^naw\bar{a}(v_2\bar{a})^nu_2\bar{r}$ is also in $L(D')$ for each $n > 0$. This implies that there does not exist a number bounding $n_a(w)$, and this is a contradiction.

"only if." Since D is non-recursive, we can build an FSA A accepting $L(D)$. Let n be the number of states in A. For the sake of contradiction, suppose that there exists a symbol a such that there is no bound on $n_a(w)$ for $w \in L(D)$, i.e. for any number $m > 0$ there exist a word $w_m \in L(D)$ such that $n_a(w_m) > m$. Let $m = n + 1$. Word w_m can be written as $ru_1v_awv'_au_2\bar{r}$, where u_1u_2 and $v_av'_a$ are properly nested words, and v_a contains $n + 1$ occurrences of a and no occurrence of $\bar{a}$.

Now, since FSA A has n states, when reading v_a, there exists a state s in A, and a subword v in v_a containing at least one occurrence of a, such that v labels a cycle starting and ending in s. Clearly, we can remove subword v from v_a in w_m and obtain another word w'_m which is accepted by FSA A. However, w'_m is not properly nested, and thus, it should not be accepted by A. Contradiction. □

Since Σ_c is finite, we have that

Corollary 1. *An EDTD D is non-recursive if and only if there exists $n^{max} \in \mathbb{N}$ such that $n_a(w) \leq n^{max}$ for each $a \in \Sigma_c$ and $w \in L(D)$.*

We can show that

Theorem 3. *Let D be a non-recursive EDTD. Then $T^{\leq K}(L(D))$ is regular.*

Proof. From the above corollary, we have that there exists $n^{max} \in \mathbb{N}$ such that $n_a(w) \leq n^{max}$ for each $a \in \Sigma_c$ and $w \in L(D)$.

Since $T^{\leq K}$ is an EVPT, it preserves VPLs, and thus, $T^{\leq K}(L(D))$ is a VPL for which we build a VPA B. As mentioned earlier, we can construct an equivalent EDTD D_B for VPA B, such that $L(D_B) = T^{\leq K}(L(D))$.

Now EVPT $T^{\leq K}$ performs only up to K operations on the words of $L(D)$, and thus, we have that $n_a(w) \leq n^{max} + K$ for each $a \in \Sigma_c$ and $w \in L(D_B)$, and by the same corollary, we have that D_B is non-recursive, i.e. $L(D_B) = T^{\leq K}(L(D))$ is regular. □

From the above we conclude that given a non-recursive EDTD D, we can effectively obtain an FSA recognizing $T^{\leq K}(L(D))$.

Chapter 9

A Streaming Algorithm for K-Validation

In this Chapter, we give an algorithm for the K-validation problem for streaming XML when the schema is a non-recursive EDTD D. To do so, we start with some definitions.

For this, let

$$B = (Q, O, \tau_B, q_0, F)$$

be the deterministic FSA for $T^{\leq K}(L(D))$. Also, let A_w be the FSA built as in Chapter 6. We consider the states of A_w to be numbered in the usual way (see Fig.6.1).

Now, we will lazily build a Cartesian product of A_w and B and remember only the frontier of this product. In fact we build the Cartesian product (which could be view as Layered Cartesian product) in an unfolded fashion in the form of a rooted tree. The root of that tree is the start state pair (p_0, q_0), and the child of any node is the state pair which can be reach by its parent by only one transition. With each state-state node in the product we maintain a counter for the number of † symbols in the path from the root to the node.

Algorithm 2

Input: FSAs A_w and B.

Output: "Yes" if $L(A_w) \cap L(B) \neq \emptyset$. "No" otherwise.

Method:

1. Initialize a frontier queue $F = \{[(p_0, q_0), 0]\}$.
2. Repeat until F becomes empty
   (a) Dequeue a pair $[(p_i, q_j), m]$ from F.
   (b) If p_i and q_j are final states in A_w and B, respectively, then return "Yes" and terminate.
   (c) If $m \leq K$ then do

   For each transition of the form (p_i, a, p_h) in τ_{A_w}, and (q_j, a, q_k) in τ_B, check and if not present, add $[(p_h, q_k), m]$ to F if $a \neq \dagger$ and $[(p_h, q_k), m + 1]$ otherwise.

   For each transition of the form $(p_i, \bar{a}, p_h)$ in τ_{A_w}, and $(q_j, \bar{a}, q_k)$ in τ_B, check and if not present, add $[(p_h, q_k), m]$ to F if $\bar{a} \neq \bar{\dagger}$ and $[(p_h, q_k), m + 1]$ otherwise.
3. Return "No."

We now analyze the space requirements of our scheme. We can show that

Theorem 1. *The space for the query phase is constant.*

Proof. In order to prove the space bound, we show the following lemma.

Lemma 1. *Consider the frontier queue F at any intermediate stage of the algorithm and let $[(p_i, q_j), h]$ be any tuple in F. Then, there exists an integer d such that $i+h = d$ or $i + h = d + 1$.*

Proof. The algorithm can be visualized as constructing a layered Cartesian product graph using a breadth-first-search starting from node $[(p_0, q_0), 0]\}$. Hence, at any stage of the algorithm, F contains nodes only at depth d and $d+1$ for some integer d. We now observe that all the nodes of the form $[(p_i, \_), h]$ in F at depth d have the property that $i+h = d$. This fact is true for the root node $[(p_0, q_0), 0]\}$. In addition, we observe that a child of a node $[(p_i, \_), h]$ is labeled either by $[(p_{i+1}, \_), h]$ or by $[(p_i, \_), h + 1]$ depending on whether the corresponding transition uses a symbol from Σ or a †. □

From the proof of the above lemma, at any moment in frontier F, we can find nodes of depth d and $d + 1$, for some integer d. Now, fix a depth, say d. Then, we show that the number of nodes of depth d in F is at most $(K + 1)|B|$, where $|B|$ is the number of states in B. For this, observe that there are at most $K + 1$ choices for the counter value $h \in \{0, 1, \ldots, K\}$ and by the lemma above, the counter value h automatically fixes the first state component, p_i to belong to $\{p_{d-K}, \ldots, p_d\}$. There

are $|B|$ choices for the second state component, q_j. Therefore, the size of the frontier queue at any stage is at most $2(K+1)|B|$. This size is a constant as it is independent of A_w. □

Chapter 10

Concluding Remarks

In this work, we have investigated the problem of approximate XML validation, an important problem in XML processing. Contributions of this thesis include:

- Introduction of EVPTs
- Building EVPTS for the various edit operations
- Their application to building two algorithms for checking approximate XML validity.

There are many interesting open questions related to this work that we would like to address in the future. We mention a few of them here.

- Can we use VPAs and VPTs to address other interesting questions about XML like XML integration?
- Is it possible to build a streaming algorithm for approximate validation of XML specified by broader classes of EDTDs?
- Is it possible to speed up our PTIME algorithm further by exploiting the structure of A_w?

Bibliography

[1] Aho, V., A., Peterson, G., T. A Minimum Distance Error-Correcting Parser for Context-Free Languages. *SIAM J. Comput.* 1(4): 1972, pp. 305–312.

[2] Alur, R., Kumar. V., Madhusudan, P., and Viswanathan, M. Congruences for Visibly Pushdown Languages. In *Proc.* 32*nd International Colloquium of Automata, Languages and Programming (ICALP)* (Lisboa, Portugal, 11–15 July 2005), pp. 1102–1114.

[3] Alur, R., and Madhusudan, P. Visibly Pushdown Languages. In *Proc.* 36*th ACM Symp. on Theory of Computing* (Chicago, Illinois, 13–15 June 2004), pp. 202–211.

[4] Balmin, A., Papakonstantinou, Y., and Vianu, V. Incremental Validation of XML Documents. *ACM Trans. Database Syst.* 29(4): 2004, pp. 710–754.

[5] Barbosa, D., Leighton, G., and Smith, A. Efficient Incremental Validation of XML Documents After Composite Updates. In *Proc. of* 2*nd Int. XML Database Symp.* (Seoul Korea, 10–11 September 2006), pp. 107–121.

[6] Barbosa, D., Mendelzon, A. O., Libkin, L., Mignet, L., and Arenas, M. Efficient Incremental Validation of XML Documents. In *Proc. of* 20*th Int. Conf. on Data Engineering* (Boston, USA, 30 March–2 April 2004), pp. 671–682.

[7] Boobna, U., and de Rougemont, M. Correctors for XML Data. In *Proc.* 2*nd International XML Database Symposium* (Toronto, Canada, 29–30 August 2004), pp. 97–111.

[8] Clark, J., and M. Murata, M. RELAX NG Specification. OASIS, December 2001.

[9] Flesca, S., Furfaro, F., Greco, S., and Zumpano, E. Querying and Repairing Inconsistent XML Data. In *Proc. 6th International Conference on Web Information Systems Engineering* (New York, USA, 20–22 November 2005), pp. 175–188.

[10] Grahne, G., and Thomo, A. Approximate Reasoning in Semistructured Data. In *Proc. of the* 8*th International Workshop on Knowledge Representation meets Databases* (Rome, Italy, 15 September 2001).

[11] Grahne, G., and Thomo, A. Query Answering and Containment for Regular Path Queries under Distortions. In *Proc. of* 3*rd International*

Symposium on Foundations of Information and Knowledge Systems (Wilhelmminenburg Castle, Austria, 17–20 February 2004), pp. 98–115.

[12] Grahne, G., and Thomo, A. Regular Path Queries under Approximate Semantics. *Ann. Math. Artif. Intell.* 46(1-2): 2006, pp. 165–190.

[13] Green, T. J., Gupta, A., Miklau, G., Onizuka, M., and Suciu, A. Processing XML Streams with Deterministic Automata and Stream Indexes. *ACM Trans. Database Syst.* 29(4): 2004, pp. 752–788.

[14] Kumar, V., Madhusudan, P. and Viswanathan, M. Visibly Pushdown Automata for Streaming XML. In *Proc. of Int. Conf. on World Wide Web* (Alberta, Canada, 8–12 May, 2007), pp. 1053–1062.

[15] Neven, F. Automata Theory for XML Researchers. *SIGMOD Record* 31(3): 2002 pp. 39–46.

[16] Martens, W., Neven, F., Schwentick, T., Bex, G., J. Expressiveness and complexity of XML Schema. *ACM Trans. Database Syst.* 31(3): 2006, pp. 770–813.

[17] Papakonstantinou, Y., and Vianu, V. DTD Inference for Views of XML Data. In *Proc.* 19*th ACM Symp. on Principles of Database Systems* (Dallas, Texas, 15–17 May 2000), pp. 35–46.

[18] Segoufin, L., and Vianu, V. Validating Streaming XML Documents. In *Proc.* 21*st ACM Symp. on Principles of Database Systems* (Madison, Wisconsin, 3–5 June 2002), pp. 53–64.

[19] Sperberg-McQueen, C., M., and Thomson, H. XML Schema 1.0. http://www.w3.org/XML/Schema, 2005.

[20] Staworko, S., and Chomicki, J. Validity-Sensitive Querying of XML Data-bases. In *Proc. of* 2*nd International Workshop on Database Technologies for Handling XML Information on the Web, EDBT Workshops* (Munich, Germany, 26–31 March 2006), pp. 164–177.

[21] Segoufin, L., and Sirangelo, C. Constant-Memory Validation of Streaming XML Documents Against DTDs. In *Proc.* 11*th International Conference on Database Theory* (Barcelona, Spain, 10–12 January 2007), pp. 299–313.

[22] Suciu, D. The XML Typechecking Problem. In *SIGMOD record* 31(1), 2002, pp. 89–96.

[23] Thomo, A., Venkatesh, S., Ye, Y., Y. Visibly Pushdown Transducers for Approximate Validation of Streaming XML. In *Proc.* 5*th International Symposium on Foundations of Information and Knowledge Systems* (Pisa, Italy, 11–15 February 2008), pp. 219–238.

Alex Thomo, Srinivasan Venkatesh, Ying Ying Ye: Visibly Pushdown Transducers for Approximate Validation of Streaming XML. FoIKS 2008: 219-238