Validation of Streaming XML Documents with Abstract State Machines

Klaus-Dieter Schewe
Information Science Research Centre
Palmerston North, New Zealand
kdschewe@acm.org

Bernhard Thalheim
Institute of Computer Science
University of Kiel, Germany
thalheim@is.informatik.uni-kiel.de

Qing Wang
Information Science Research Centre
Palmerston North, New Zealand
wang0qing@gmail.com

ABSTRACT

The exact validation of streaming XML documents can be realised by using visibly push-down automata (VPA) that are defined by Extended Document Type Definitions (EDTD). It is straightforward to represent such an automaton as an Abstract State Machine (ASM). However, creating a whole VPA is not an efficient validation approach. In this paper it is shown, how the VPA construction can be avoided by using a refined ASM that only requires knowledge of the EDTD. In a second step this approach is extended to approximate validation of streaming XML documents taking at most k updates to the document into consideration.

Categories and Subject Descriptors

E.m [**Data**]: Miscellaneous; H.2 [**Database Management**]: XML

General Terms

theory, algorithms

Keywords

XML, Validation, Abstract State Machines

1. INTRODUCTION

The eXtensible Markup Language (XML) has become a standard for the data exchange on the world-wide web, and XQuery has become the XML querying standard. Most current XQuery implementations require that all XML data reside in memory in one form or another before they start processing the data. This is unacceptable for large XML documents. Typical XQuery processors such as Xalan, Qizz, and Saxon thus fail to handle very large XML documents. Some others, such as Galax take advantage of the query structure by storing in memory only those parts that are needed by the query.

The Document Object Model (DOM) [7, 8] is a platform- and language-neutral API that provides a standard set of interfaces for manipulating an XML document. Scripts can dynamically access and update the content, structure, and style of an XML document. DOM is tree-based and requires that the entire document is represented in memory while processing it. Unfortunately, there are no standard ways to support namespaces in the DOM, nor are there standard ways to create empty DOM documents. The DOM specification does not define how namespaces are supported. Thus, some DOM implementations have defined methods for retrieving various information about the namespace used by a given node.

An alternative approach to the tree-based DOM is provided by the Simple API for XML (SAX) [5, 6, 8, 13] is a non-W3C standard API for streaming document processing. The event-based API reports parsing events directly to the application through callbacks. It typically does not build an internal tree, but handlers deal with the different events. This approach results in simpler parsing and processing of XML documents. It does not keep XML documents and their structure in the memory. Therefore SAX makes it possible to process very large XML documents without exceeding the capacity of memory available for processing.

Event-based techniques such as SAX that do not require the materialisation of all data in memory have influenced the development of theoretical tree transducer models for parsers such as visibly pushdown automata [9].

Extended Document Type Definitions (EDTD), introduced by Papakonstantinou and Vianu [10] provide a general schema formalism based on special context-free grammars that generalises other schema languages such as DTDs and XML Schema, the major extension to DTDs being the introduction of some form of typing. Kumar et al. [9] have shown that the tree languages specified by EDTDs are exactly the visibly pushdown languages (VPL), i.e. those that can be recognised by a specialised class of push-down automata, the visibly pushdown automata (VPA). That is, in order to decide if a given XML document adheres to a specified EDTD, the word representing the XML document must be accepted by the derived non-deterministic or deterministic VPA. In particular, this turns out to be extremely useful for validating streaming XML documents [11, 9], where the document is validated, while it is read. General properties of VPLs such as equivalence of deterministic and non-deterministic VPAs, closure under intersection and union, etc. have been investigated by Alur and Madhusudan [1].

This processing by finite state machines, transducers, or VPA does an excellent job as long as no predicates or complex queries are required. However, these parsers are statically constructed in advance based on the XSchema, DTDs or EDTDs. As XQuery is a functional language and XQuery expressions can appear at any place in a query, a recursive compositional translation of the query is required, but the approaches based on finite state machines require a holistic view of an XPath expression before the automaton is constructed.

In this paper we want to show how the problem of validating

iiWAS 2008 Linz, Austria

streaming XML documents can be approached by using Abstract State Machines (ASM) [3]. This is justified by the claim that dynamic construction of the parser based on a pattern specification of the rule that may parse a sub-expression is the simplest and most effective way of parsing, analysing and processing sets of documents and queries. The rule pattern used for the generation of concrete parsing rules are very flexible and therefore support queries beyond monadic second-order formula.

In a first straightforward approach we model the VPA that is used for the exact validation of streaming XML documents with a given EDTD by an ASM. This corresponds to creating a "ground model" according to the ASM methodology introduced by Börger [2]. However, first creating a rather big VPA out of a given EDTD and then using this VPA for the validation task is not efficient. Therefore, we refine the ASM by an ASM that does not use the VPA at all, but works directly on the structure defined by the EDTD and thus avoids the VPA construction step.

In a second step we generalise this approach to approximate validation of streaming XML documents, i.e. accepting documents that differ from the given EDTD by at most k edit operations – the case $k = 0$ captures the exact validation problem as a special case. Thomo et al. [12] showed that in case insertions, deletions and updates of pairs of opening and closing tags are counted as individual edit operations the approximate validation problem can be solved by using the product of the VPA used for exact validation and a visibly pushdown transducer (VPT) with $2k + 1$ states that captures the count of edit operations[1]. We show that the switch from the exact to the approximate validation problem for streaming XML documents gives rise to another ASM refinement.

With these ASM refinements it does not matter, whether we first switch from a specification exploiting VPAs to one avoiding the VPA use and then take up to k edit operations into account or apply refinements the other way round. In this sense the problem of validating streaming XML documents shows characteristics of horizontal and vertical refinement same as the Java/JVM refinements [4]. That is, we can regard the approximate validation problem as one that is composed of the exact validation problem (the case $k = 0$), and the generalisation by means of a simple VPT. Then refining the corresponding ASMs and eliminating the direct references to VPAs and VPTs amounts to a refinement that has at its core a refinement of the component dealing with the exact validation problem.

2. EXTENDED DOCUMENT TYPE DEFINITIONS

Document Type Definitions (DTD) provide the first and simplest form of adding schema information to XML documents. Abstracting from specific syntax of opening and closing tags and blurring the dinstinction between subelements and attributes we can define a DTD as follows, called *labelled ordered tree object type definition* in [10].

DEFINITION 1. A *document type definition* (DTD) consists of an alphabet Σ, a root $r \in \Sigma$ and a mapping $\ell : \Sigma \to \mathfrak{P}(\Sigma^*)$ assigning to each $a \in \Sigma$ a regular language over Σ.

EXAMPLE 1. The following (adapted from [10]) denotes a DTD with $\Sigma = \{\text{root, dealer, used\_cars, new\_cars, ad, model, year}\}$ and root 'root':

root: dealer dealer: used_cars new_cars used_cars: ad*
new_cars: ad* ad: model year$^?$

The assigned languages are

$$\begin{aligned} \ell(\text{root}) &= \ell(\text{dealer}) \\ \ell(\text{dealer}) &= \text{dealer } \ell(\text{used\_cars}) \; \ell(\text{new\_cars}) \\ \ell(\text{used\_cars}) &= \text{used\_cars } \ell(\text{ad})^* \\ \ell(\text{ad}) &= \text{ad } \ell(\text{model}) \; (\ell(\text{year}) \cup \{\epsilon\}) \\ \ell(\text{new\_cars}) &= \text{new\_cars } \ell(\text{ad})^* \\ \ell(\text{model}) &= \{\text{model}\} \quad \text{and} \\ \ell(\text{year}) &= \{\text{year}\} \end{aligned}$$

This corresponds to the (real) DTD

⟨!DOCTYPE dealer [
  ⟨!ELEMENT dealer (used_cars new_cars)⟩
  ⟨!ELEMENT used_cars (ad*)⟩
  ⟨!ELEMENT new_cars (ad*)⟩
  ⟨!ELEMENT ad ((model year)|(model))⟩
  ⟨!ELEMENT model PCDATA⟩
  ⟨!ELEMENT year PCDATA⟩
]⟩

While such DTDs provide some schema information, they cannot express all desirable properties of XML documents. E.g., the DTD in Example 1 would not allow us to request that the 'year' tag must be present for and only for used cars. This could only be avoided by having two different tags such as 'ad_used' and 'ad_new'. Extended DTDs as introduced in [10] (as *specialised labelled ordered tree object type definition*) take care of this problem.

DEFINITION 2. An *Extended Document Type Definition* (EDTD) consists of a DTD (Σ', r, ℓ) and a mapping $\mu : \Sigma' \to \Sigma$ with another alphabet Σ.

We can use the elements in Σ' to fine-tune the desired structure of XML document adhering to a given EDTD, while $\mu(a)$ defines the actual tag that is to be used. E.g., in our example above we could use 'ad_used' and 'ad_new' as elements of Σ' with both being mapped by μ to 'ad' in Σ – all other elements of Σ' would be mapped to themselves.

We adopt the notational convention to write a^b for element in Σ' with $\mu(a^b) = a \in \Sigma$. The superscript b of a^b is then also called the *type* of the element. If $\mu^{-1}(a)$ contains only one element, we omit the superscript and assume that μ maps a to itself.

EXAMPLE 2. The following (adapted from [10]) denotes an EDTD with $\Sigma = \{\text{root, dealer, used\_cars, new\_cars, ad, model, year}\}$:

root: dealer dealer: used_cars new_cars
used_cars: (ad^u)* new_cars: (ad^n)*
ad^u: model year ad^n: model

In an XML document adhering to this EDTD we would indeed have 'model' and 'year' for each used car, but only 'model' for new cars.

3. VISIBLY PUSHDOWN AUTOMATA

Kumar et al. [9] proved that the tree languages specified by EDTDs are exactly the visibly pushdown languages (VPL), i.e. those that can be recognised by a specialised class of push-down automata, the visibly pushdown automata (VPA). Following [12] we ignore internal actions, as these would just specify the kind of strings associated with leaf elements or attributes. This leads to the following simplified definition of a VPA.

[1]Thomo et al. [12] also provided a solution for the case of counting insertions, deletions and updates of tag pairs with the same name as only one edit operation, which only requires a different VPT.

DEFINITION 3. A *visibly pushdown automaton* (VPA) consists of a finite set Q of states, an start state $q_0 \in Q$, a set of final states $F \subseteq Q$, an (input) alphabel Σ that is partitioned into call symbols in Σ_c and return symbols in Σ_r with a bijection $^- : \Sigma_c \to \Sigma_r$, a stack alphabet Γ containing a special (bottom of stack) symbol $\perp \in \Gamma$, and a transition relation $\tau = \tau_c \cup \tau_r \cup \tau_\epsilon$ with $\tau_c \subseteq Q \times \Sigma_c \times Q \times \Gamma$, $\tau_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$, and $\tau_\epsilon \subseteq Q \times Q$.

Intuitively speaking, a transition $(q_1, a, q_2, \gamma) \in \tau_c$ means that if the automaton is in state q_1 and reads the call symbol a, then it changes the state to q_2 and pushes γ onto the stack. A transition $(q_1, \bar{a}, \gamma, q_2) \in \tau_r$ means that if the automaton in state q_1 reads the return symbol $\bar{a} \in \Sigma_r$ and the symbol γ is on top of the stack, then γ will be popped off the stack and the automaton moves to state q_2. A transition $(q_1, q_2) \in \tau_\epsilon$ just means that in state q_1 the automaton may read nothing, leave the stack unchanged and switch to state q_2.

More formally, a VPA induces a transition relation T on *configurations* $Q \times \Sigma^* \times \Gamma^*$. A start configuration has the form $(q_0, w, \perp)$, and a final configuration has the form $(q_f, \epsilon, \perp)$ with $q_f \in F$. Each transition in τ induces a set of transitions in T.

A transition $(q_1, a, q_2, \gamma) \in \tau_c$ gives rise to configuration pairs $((q_1, aw, v), (q_2, w, \gamma v))$, a transition $(q_1, \bar{a}, \gamma, q_2) \in \tau_r$ gives rise to $((q_1, \bar{a}w, \gamma v), (q_2, w, v))$, and a transition $(q_1, q_2) \in \tau_\epsilon$ gives rise to $((q_1, w, v), (q_2, w, v))$ (with $w \in \Sigma^*$ and $v \in \Gamma^*$). Then a successful *run* is a sequence of configurations $\sigma_0, \ldots, \sigma_f$ with a start configuration σ_0, a final configuration σ_f, and $(\sigma_{i-1}, \sigma_i) \in T$ for all $i = 1, \ldots, f$.

Using VPAs for validating streaming XML documents that are to adhere to a given EDTD, a call transition corresponds to reading an opening tag, for which a corresponding symbol is pushed onto the stack, while a return transition would read the matching closing tag and remove the corresponding symbol from the stack.

EXAMPLE 3. The following VPA (illustrated in Figure 1) can be used to recognise XML documents that adhere to the EDTD in Example 2:

$$
\begin{aligned}
Q &= \{q_0, q_1, \ldots, q_{14}\} \\
\Sigma_c &= \{\text{dealer}, \text{used\_cars}, \text{new\_cars}, \text{ad}, \text{model}, \text{year}\} \\
\Sigma_r &= \{\bar{a} \mid a \in \Sigma_c\} \\
\Gamma &= \{\perp\} \cup \{d, u, a^u, m, y, n, a^n\}
\end{aligned}
$$

with start state q_0, final states $F = \{q_{14}\}$, and the following transitions:

$$
\begin{aligned}
\tau_c = \{&(q_0, \text{dealer}, q_1, d), (q_1, \text{used\_cars}, q_2, u), (q_2, \text{ad}, q_3, a^u), \\
&(q_3, \text{model}, q_4, m), (q_5, \text{year}, q_6, y), (q_8, \text{new\_cars}, q_9, n), \\
&(q_9, \text{ad}, q_{10}, a^n), (q_{10}, \text{model}, q_{11}, m)\} \\
\tau_r = \{&(q_4, \overline{\text{model}}, m, q_5), (q_6, \overline{\text{year}}, y, q_7), (q_7, \overline{\text{ad}}, a^u, q_2), \\
&(q_2, \overline{\text{used\_cars}}, u, q_8), (q_{11}, \overline{\text{model}}, m, q_{12}), (q_{12}, \overline{\text{ad}}, a^n, q_9), \\
&(q_9, \overline{\text{new\_cars}}, n, q_{13}), (q_{13}, \overline{\text{dealer}}, d, q_{14})\}
\end{aligned}
$$

4. VALIDATING STREAMING XML DOCUMENTS

Let us now address the exact validation of streaming XML documents using Abstract State Machines. The straightforward idea is to specify an ASM that models a validating (deterministic or nondeterministic) VPA. In this case we only need four 0-ary functions, i.e. variables in the signature of the ASM:

tag(0) monitored
parse(0), state(0), stack(0) controlled

We can assume that tag always contains the next input symbol or the end of input symbol, say $\perp$. Once the ASM reads this symbol, tag will be updated to the next input symbol. The variable parse is used for the result of the validation. It is set to 1, if the XML document adheres to the EDTD, and to 0 otherwise. The variables state and stack contain the values of the current state and the content of the stack, i.e. a list of symbols. Furthermore, assume that rules pop and push(x) for popping values from and pushing them onto the stack, respectively, are defined elsewhere. Then we obtain the following simple main rule for a validating ASM:

main = (state := q_0 ‖ stack := $\perp$ ‖ parse := 0) ; check

The check rule then has to read the next input symbol and depending on the state and the stack either terminate with an error or continue checking until there is no more input symbol. As we may have to deal with a non-deterministic VPA we must also provide a choice of a case number – for deterministic VPAs this is not needed. Thus, we obtain the following general form for the check rule:

```
check =
    read_next(tag) ;
    if tag ≠ ⊥
    then choose k ∈ ℕ do
        case ...
        case k = i and state = q_i and tag = a_i
            then (push(γ_i) ‖ state := q'_i) ; check endcase
        case ...
        case k = j and state = q_j and tag = a_j
                                  and top(stack) = γ_j
            then (pop ‖ state := q'_j) ; check endcase
        case ...
        enddo
    else parse := 1
    endif
```

Here the two highlighted cases $k = i$ and $k = j$ correspond to transitions in Σ_c and Σ_r, respectively. E.g., the i'th case for the VPA in Example 3 could be

```
case k = 3 and state = q_2 and tag = 'ad'
                then (push(a^u) ‖ state := q_3) ; check
endcase
```

This approach to specifying the validating VPA by an ASM is straightforward, the only advantage being that there is no need to switch from a non-deterministic to a deterministic VPA. In order to obtain a more suitable ASM specification we refine the ASM that we obtained from the VPA by first making the concepts of state and stack more explicit. Both together merely represent where in the parsing of an XML tree we are actually located, which can be as well represented explicitly by using relations for elements and siblings. More precisely, let the ASM signature contain functions

sibling(3) static and element(3) controlled

Then element$(n, t, i) = \perp$ means that there is no element with name n, type t and identifier i in the EDTD, while element$(n, t, i) = k$ with $k \in \{0, 1, 2\}$ means that there is an element with name n, type t and identifier i in the EDTD, which is inactive, active, or one of its children is active, respectively. Once we receive an opening tag it will become active and remain so as long as its children are processed, and become inactive after receiving the matching closing tag. Furthermore, sibling$(i_1, i_2, i) := 1$ means that an element with identifier i_1 may be the left neighbour of an element with identifier i_2, both under the parent element with identifier i. The fact

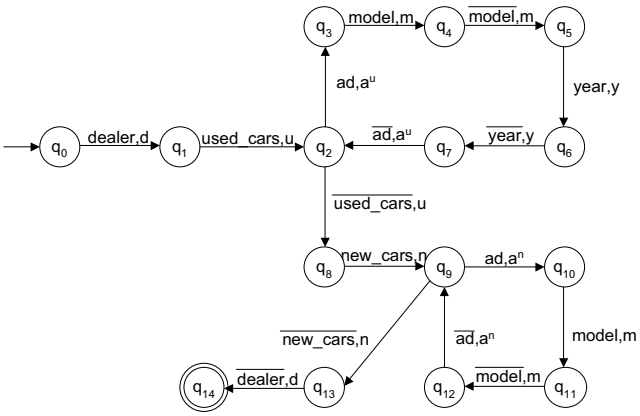

Figure 1: A VPA for validating streaming XML documents

that there is no left or right neighbour will be modelled by letting $i_1 = \perp$ or $i_2 = \perp$, respectively.

As auxiliary controlled functions we further need depth(0), previous(1) with depth taking the actual depth in the XML tree as value, while previous(d) will be set to the identifier of the last child on depth d that has been processed.

EXAMPLE 4. For our EDTD in Example 2 we initialise the functions as previous(d) $= \perp$ for all d, depth = 0, and the values for element and sibling are defined by tables:

element

name	type	id	state
root	$\perp$	0	1
dealer	$\perp$	1	0
used_cars	$\perp$	2	0
new_cars	$\perp$	3	0
ad	u	4	0
ad	n	5	0
model	$\perp$	6	0
year	$\perp$	7	0

sibling

younger	older	parent
$\perp$	1	0
$\perp$	2	1
2	3	1
$\perp$	4	2
4	4	2
$\perp$	5	3
5	5	3
$\perp$	6	4
6	7	4
$\perp$	6	5
1	$\perp$	0
3	$\perp$	1
4	$\perp$	2
5	$\perp$	3
7	$\perp$	4
6	$\perp$	5

Similar as before the main ASM rule then takes the form main = (initialise $\|$ parse := 0) ; check, so we can concentrate on the check rule, which can be defined as follows:

check =
  read_next(tag) ;
  if tag $\neq \perp$
  then if $\exists n, t_1, i_1, i_2, t_2$ with element(n,t_1,i_1) = 1 $\wedge$
        element(tag,t_2,i_2) $\neq \perp \wedge$
        sibling(previous(depth),i_2,i_1) = 1
      then (element(n,t_1,i_1) := 2 $\|$ element(tag,t_2,i_2) := 1 $\|$
        previous(depth) := i_2) ; depth := depth + 1 ; check
      elsif $\exists n, t_1, i_1, n_2, i_2, t_2$ with element(n_1,t_1,i_1) = 1 $\wedge$
        element(n_2,t_2,i_2) = 2 $\wedge$ tag = $\bar{n_1}$
      then (element(n_1,t_1,i_1) := 0 $\|$ element(n_2,t_2,i_2) := 1 $\|$
          depth := depth - 1) ;
        previous(depth) := i_1 ;
        if sibling(i_1,$\perp$,i_2) = 1
        then previous(depth+1) := $\perp$
        endif)) ; check
  else if element(root,$\perp$,0)=1
      then parse := 1
      endif
  endif

Note that this ASM specification captures any EDTD, the difference being each time only the values for the functions sibling and element. However, while this ASM makes the notion of state and stack explicit – both used for characterisation of the position within the ordered XML tree – it does not appear to look much simpler than the ASM specification that was based on the recognising VPA. In order to simplify the ASM specification we apply a further refinement step by instantiating the ASM specification. That is, we exploit the fact that our EDTD is finite, so the tables for element and sibling will be finite. By substituting all possible cases for the value of 'tag' in the check rule we blow up the size of the ASM specification, but at the same time manage to get rid of element and sibling. Furthermore, we eliminate the use of identifiers, as tag name and type will be sufficient. Thus, we only require a unary controlled function 'state', initialised with state(root) = 1.

EXAMPLE 5. For the EDTD in Example 2 and tag = dealer we obtain the simplified case

case tag = dealer $\wedge$ previous(0) = $\perp \wedge$ state(root) = 1
then (state(root) := 2 $\|$ state(dealer) := 1 $\|$
  previous(0) := dealer $\|$ depth := 1) ; check

Similarly, for tag = $\overline{\text{model}}$ we obtain the case

case tag = $\overline{\text{model}} \wedge$ state(model) = 1 $\wedge$ state(ad^u) = 2
then (state(model) := 0 $\|$ state(ad^u) := 1 $\|$
previous(3) := model $\|$ depth := 3) ; check

The resulting ASM covers the various cases resulting from the EDTD, but avoids the creation of the VPA. Furthermore, the EDTD is not explicitly stored anymore.

5. APPROXIMATE VALIDATION OF STREAMING XML DOCUMENTS

The approximate validation of streaming XML documents works in principle in the same way as the exact validation. The difference is that we permit up to k edit operations, which can be the change of a tag name, the omission of a tag, or the insertion of an additional tag. As shown by Thomo et al. [12] the approximate solution (in case all changes to pairs of opening/closing tags are counted as one edit operation each) can be achieved by a VPA that is the product of the VPA used for the exact validation problem and a visibly pushdown transducer (VPT) with $2k+1$ states. The VPT just increments the state count by one for every change of tag. So we need also up to k additional call symbols that can be used as new or replacement symbols and additional stack symbols indicating insertion, deletion and update of tags. We omit the details of the VPT and the product construction (see [12]), but instead illustrate the resulting VPA for the EDTD in Example 2.

EXAMPLE 6. The following VPA can be used to recognise XML documents that adhere to the EDTD in Example 2 up to k edit operations (for simplicity let us assume that insertions and replacements of tags always used new tags):

$$\begin{aligned}
Q &= \{q_{i,j} \mid 0 \le i \le 14, 0 \le j \le 2k\}\\
\Sigma_c &= \{\text{dealer}, \text{used_cars}, \text{new_cars}, \text{ad}, \text{model}, \text{year}\}\\
&\qquad \cup \{\text{new}_i \mid 1 \le i \le k\}\\
\Sigma_r &= \{\bar{a} \mid a \in \Sigma_c\}\\
\Gamma &= \{\bot\} \cup \{d, u, a^u, m, y, n, a^n\} \cup \{\iota_i \mid 1 \le i \le k\}\\
&\qquad \cup \{\delta_x \mid x \in \{d, u, a^u, m, y, n, a^n\}\}\\
&\qquad \cup \{\sigma_{x,j} \mid x \in \{d, u, a^u, m, y, n, a^n\}, 1 \le j \le k\}
\end{aligned}$$

with start state $q_{0,0}$, final states $F = \{q_{14,2j} \mid 0 \le j \le k\}$. The following transitions (with $0 \le j \le 2k$) are those from Example 3 capturing the processing of tags without edit – note that the only change is the replacement of state q_i by $q_{i,j}$:

$$\begin{aligned}
\tau_c \supseteq \{&(q_{0,j}, \text{dealer}, q_{1,j}, d), (q_{1,j}, \text{used_cars}, q_{2,j}, u),\\
&(q_{2,j}, \text{ad}, q_{3,j}, a^u), (q_{3,j}, \text{model}, q_{4,j}, m),\\
&(q_{5,j}, \text{year}, q_{6,j}, y), (q_{8,j}, \text{new_cars}, q_{9,j}, n),\\
&(q_{9,j}, \text{ad}, q_{10,j}, a^n), (q_{10,j}, \text{model}, q_{11,j}, m)\}\\
\tau_r \supseteq \{&(q_{4,j}, \overline{\text{model}}, m, q_{5,j}), (q_{6,j}, \overline{\text{year}}, y, q_{7,j}),\\
&(q_{7,j}, \overline{\text{ad}}, a^u, q_{2,j}), (q_{2,j}, \overline{\text{used_cars}}, u, q_{8,j}),\\
&(q_{11,j}, \overline{\text{model}}, m, q_{12,j}), (q_{12,j}, \overline{\text{ad}}, a^n, q_{9,j}),\\
&(q_{9,j}, \overline{\text{new_cars}}, n, q_{13,j}), (q_{13,j}, \overline{\text{dealer}}, m, q_{14,j})\}
\end{aligned}$$

Similarly, we obtain transitions for the deletion of tags, i.e. instead of the tag x expected as specified by the EDTD we read ϵ, but nevertheless treat this as if x were read, and use δ_x as the corresponding stack symbol:

$$\begin{aligned}
\tau_c \supseteq \{&(q_{0,j}, \epsilon, q_{1,j+1}, \delta_d), (q_{1,j}, \epsilon, q_{2,j+1}, \delta_u),\\
&(q_{2,j}, \epsilon, q_{3,j+1}, \delta_{a^u}), (q_{3,j}, \epsilon, q_{4,j+1}, \delta_m),\\
&(q_{5,j}, \epsilon, q_{6,j+1}, \delta_y), (q_{8,j}, \epsilon, q_{9,j+1}, \delta_n),\\
&(q_{9,j}, \epsilon, q_{10,j+1}, \delta_{a^n}), (q_{10,j}, \epsilon, q_{11,j+1}, \delta_m)\}\\
\tau_r \supseteq \{&(q_{4,j}, \epsilon, \delta_m, q_{5,j+1}), (q_{6,j}, \epsilon, \delta_y, q_{7,j+1}),\\
&(q_{7,j}, \epsilon, \delta_{a^u}, q_{2,j+1}), (q_{2,j}, \epsilon, \delta_u, q_{8,j+1}),\\
&(q_{11,j}, \epsilon, \delta_m, q_{12,j+1}), (q_{12,j}, \epsilon, \delta_{a^n}, q_{9,j+1}),\\
&(q_{9,j}, \epsilon, \delta_n, q_{13,j+1}), (q_{13,j}, \epsilon, \delta_d, q_{14,j+1})\}
\end{aligned}$$

For insertions of tags we simply allow to read an additional new symbol, i.e. we obtain transitions

$$\begin{aligned}
\tau_c \supseteq \{&(q_{i,j}, \text{new}_h, q_{i,j+1}, \iota_h) \mid 0 \le i \le 14,\\
&\qquad 1 \le h \le k, 0 \le j \le 2k\}\\
\tau_r \supseteq \{&(q_{i,j}, \overline{\text{new}}_h, \iota_h, q_{i,j+1}) \mid 0 \le i \le 14,\\
&\qquad 1 \le h \le k, 0 \le j \le 2k\}
\end{aligned}$$

Finally, for replacements we obtain transitions similar to the case of deletions, but reading a new symbol new$_h$ instead of the one expected according to the definition of the EDTD. In this case we use $\sigma_{x,h}$ as the stack symbol:

$$\begin{aligned}
\tau_c \supseteq \{&(q_{0,j}, \text{new}_h, q_{1,j+1}, \sigma_{d,h}), (q_{1,j}, \text{new}_h, q_{2,j+1}, \sigma_{u,h}),\\
&(q_{2,j}, \text{new}_h, q_{3,j+1}, \sigma_{da^u,h}), (q_{3,j}, \text{new}_h, q_{4,j+1}, \sigma_{m,h}),\\
&(q_{5,j}, \text{new}_h, q_{6,j+1}, \sigma_{y,h}), (q_{8,j}, \text{new}_h, q_{9,j+1}, \sigma_{n,h}),\\
&(q_{9,j}, \text{new}_h, q_{10,j+1}, \sigma_{a^n,h}), (q_{10,j}, \text{new}_h, q_{11,j+1}, \sigma_{m,h})\}\\
\tau_r \supseteq \{&(q_{4,j}, \overline{\text{new}}_h, \sigma_{m,h}, q_{5,j+1}), (q_{6,j}, \overline{\text{new}}_h, \sigma_{y,h}, q_{7,j+1}),\\
&(q_{7,j}, \overline{\text{new}}_h, \sigma_{a^u,h}, q_{2,j+1}), (q_{2,j}, \overline{\text{new}}_h, \sigma_{u,h}, q_{8,j+1}),\\
&(q_{11,j}, \overline{\text{new}}_h, \sigma_{m,h}, q_{12,j+1}), (q_{12,j}, \overline{\text{new}}_h, \sigma_{a^n,h}, q_{9,j+1}),\\
&(q_{9,j}, \overline{\text{new}}_h, \sigma_{n,h}, q_{13,j+1}), (q_{13,j}, \overline{\text{new}}_h, \sigma_{d,h}, q_{14,j+1})\}
\end{aligned}$$

As for the exact validation of streaming XML documents it is straightforward to specify the VPA by means of an ASM. Each transition gives rise to a case as before. We omit the details. Let us instead refine the ASM dealing with the exact validation of streaming XML documents in general to one that permits at most k edit operations. This also arises as refinement of the ASM specification based on the VPA for approximate XML document validation by making again the state and stack explicit. In doing so, we change the definition of the check rule to

check = choose $x \in \{0, i, d, u\}$ do check'(x) enddo

letting the values $0, i, d, u$ capture the normal case and the cases of insertion, delete and update, respectively. We then need two more functions in the ASM signature

change(3) controlled count(0) controlled

Initially, count will be set to 0, while change is completely undefined. Later, change$(n, t, n') = \ell$ will indicate that the tag name n with type t has been changed to n' – the value $\bot$ for n and n' covering insertions and deletions, respectively – and ℓ gives a count for this change.

Let us now look at the four cases in the new check rule. Obviously, check'(0) is specified in the same way, as check was specified before the refinement (keeping the call of the check rule). Now, look at the other cases.

for insertion:

check'(i) =
read_next(tag) ;
if count $< k \wedge \exists h$.tag = new$_h$
then choose n, t_1, i_1 with element(n,t_1,i_1) = 1 do
(element(n,t_1,i_1) := 2 $\|$ count := count + 1 $\|$
change($\perp$,$\perp$,new$_h$) := count + 1 $\|$
choose x with $\forall n', t', x$.element$(n', t', x) = \perp$ do
element(new$_h$, $\perp$, x) := 1 enddo) ;
depth := depth + 1 ; check
elsif $\exists h$ with tag = $\overline{\text{new}}_h$
then choose n, t, i, i_1 with element$(n, t, i) = 2 \wedge$
element(new$_h$, $\perp$, i_1) = 1 do
(element(new$_h$, , $\perp$, i_1) := $\perp$ $\|$ element(n, t, i) := 1 $\|$
depth := depth - 1) enddo ; check
endif

for deletion:

check'(d) =
if count $< k$
then if $\exists n, t, i, n_2, t_2, i_2$ with element$(n, t, i) = 1 \wedge$
element(n_2,t_2,i_2) $\neq \perp \wedge$
sibling(previous(depth),i_2,i_1) = 1
then (element(n, t, i) := 2 $\|$ element(n_2,t_2,i_2) := 1 $\|$
previous(depth) := i_2 $\|$ depth := depth + 1 $\|$
count := count + 1 $\|$
change$(n_2, t_2, \perp)$:= count + 1) ; check
elsif $\exists n, t, i, n_2, i_2, t_2$ with element$(n, t, i) = 1 \wedge$
element(n_2,t_2,i_2) = 2 $\wedge$
$\exists x \neq \perp$.change$(n, t, \perp) = x \wedge \forall n', t', i', y.$
(change$(n', t', i') = y \Rightarrow y \leq x$)
then (element(n_1,t_1,i_1) := 0 $\|$ element(n_2,t_2,i_2) := 1 $\|$
depth := depth - 1 $\|$ change$(n, t, \perp)$:= $\perp$) ;
previous(depth) := i_1 ;
if sibling(i_1,$\perp$,i_2) = 1
then previous(depth+1) := $\perp$
endif)) ; check
endif

for update:

check'(u) =
read_next(tag) ;
if count $< k$
then if $\exists h$.tag = new$_h \wedge$
$\exists n, t_1, i_1, n_2, t_2, i_2$ with element$(n, t_1, i_1) = 1 \wedge$
element$(n_2, t_2, i_2) \neq \perp \wedge$
sibling(previous(depth)),i_2, i_1) = 1
then (element(n, t_1, i_1) := 2 $\|$
element(new$_h, t_2, i_2$) := 1 $\|$
previous(depth) := i_2 $\|$
change$(n_2, t_2,$ new$_h$) := count + 1 $\|$
count := count + 1) ; depth := depth + 1 ; check
elsif $\exists h$.tag = $\overline{\text{new}}_h \wedge$
$\exists n_1, t_1, i_1, n_2, t_2, i_2$ with element(new$_h, t_1, i_1$) = 1
$\wedge$ element$(n_2, t_2, i_2) = 2 \wedge$
$\exists x \neq \perp$.change$(n_1, t_1,$ new$_h) = x \wedge \forall n', t', i', y.$
(change$(n', t', i') = y \Rightarrow y \leq x$)
then (element(n_1,t_1,i_1) := 0 $\|$ element(n_2,t_2,i_2) := 1 $\|$
depth := depth - 1 $\|$
change$(n_1, t_1,$ new$_h)$:= $\perp$) ;
previous(depth) := i_1 ;
if sibling(i_1,$\perp$,i_2) = 1
then previous(depth+1) := $\perp$
endif)) ; check
endif
endif

While this ASM handles again any EDTD specified by means of the 'element' and 'sibling' functions, we can further refine it to obtain an ASM for approximate validation of streaming XML documents under a specific EDTD. As before, we substitute for all cases in the check/check' rules the possible values for tag, i.e. dealer, new_cars, etc., and eliminate identifiers. For this we would again require the unary controlled function 'state' in the signature. Furthermore, we would still use the functions count and change.

EXAMPLE 7. For the EDTD in Example 2 and tag = dealer we obtain the simplified case dealing with an update to the new tag name new$_h$:

case tag = new$_h \wedge$ previous(0) = $\perp \wedge$ state(root) = 1
then (state(root) := 2 $\|$ state(dealer) := 1 $\|$ depth := 1 $\|$
previous(0) := dealer $\|$ count := count + 1 $\|$
change(dealer,$\perp$,new$_h$) := count + 1) ; check

Similarly, for the deletion of tag = $\overline{\text{model}}$ we obtain the case

case state(model) = 1 $\wedge$ state(ad^u) = 2 $\wedge$
$\exists x$. change(model,$\perp$,$\perp$) = $x \wedge \forall n', t', i', y.$
(change$(n', t', i') = y \Rightarrow y \leq x$)
then (state(model) := 0 $\|$ state(ad^u) := 1 $\|$
change(model,$\perp$,$\perp$) := $\perp$ $\|$
previous(3) := model $\|$ depth := 3 $\|$) ; check

6. CONCLUSION

In this paper we showed how ASMs can be used for the exact and the approximate validation problem of streaming XML doucuments on the basis of EDTDs and corresponding VPAs. In particular, we showed that the two problems can be related by means of ASM refinements. On one hand it is straightforward to see that a VPA for XML document validation can be represent as an ASM – naturally, ASMs provide the much more expressive computational model. However, the advantage of the ASM approach is that it provides a single specification dealing with any kind of EDTD – only an encoding of the EDTD in two input relations 'element' and 'sibling' is required. This specification can then be refined to result in a specification of a parser that is specific for a given EDTD.

In particular, there is no need to store the complete XML document as in tree-based parsing approaches such as DOM. In this sense our solution follows the line of event-based approaches to XML such as SAX. Due to the expressiveness of ASMs it even supports the recursive compositional translation of queries, which is a requirement resulting from the nesting within XQuery that cannot be easily accommodated in automata-based approaches.

The elimination of automata in the approach can also be seen as ASM refinements. As this shows ressemblance to the Java/JVM study, in which horizontal and vertical refinements co-exist, it gives rise to the question, whether decomposition of ASM specifications and refinements of components can be defined as a general add-on to the ASM development methodology.

7. REFERENCES

[1] R. Alur and P. Madhusudan. Visibly pushdown languages. In L. Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC 2004)*, pages 202–211. ACM, 2004.

[2] E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.

[3] E. Börger and R. Stärk. *Abstract State Machines*. Springer-Verlag, Berlin Heidelberg New York, 2003.

[4] E. Börger, R. Stärk, and J. Schmid. *Java and the Java Virtual Machine: Definition, Verification and Validation*. Springer-Verlag, Berlin Heidelberg New York, 2001.

[5] D. Box, A. Skonnard, and J. Lam. *Essential XML: Beyond Markup*. Addison Wesley, 2000.

[6] L. M. Garshol. *Definitive XML Application Development*. Prentice-Hall, 2002.

[7] S. Gupta, G. Kaiser, D. Neistadt, and P. Grimm. DOM-based content extraction of HTML documents. In *Proc. 12th WWW conf.*, pages 207–214. ACM Press, 2003.

[8] E. R. Harold. *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX*. Addison Wesley, 2002.

[9] V. Kumar, P. Madhusudan, and M. Viswanathan. Visibly pushdown automata for streaming XML. In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web (WWW 2007)*, pages 1053–1062. ACM, 2007.

[10] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2000)*, pages 35–46. ACM, 2000.

[11] L. Segoufin and V. Vianu. Validating streaming XML documents. In L. Popa, editor, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2002)*, pages 53–64. ACM, 2002.

[12] A. Thomo, S. Venkatesh, and Y. Y. Ye. Visibly pushdown transducers for approximate validation of streaming XML. In S. Hartmann and G. Kern-Isberner, editors, *Foundations of Information and Knowledge Systems – Proc. 5th International Symposium, FoIKS 2008*, volume 4932 of *LNCS*, pages 219–238. Springer-Verlag, 2008.

[13] E. Wilde. *Advanced XML Technologies*. CRC Press, 2004.