Journal of Universal Computer Science, vol. 15, no. 10 (2009), 2028-2057
submitted: 1/2/09, accepted: 17/4/09, appeared: 28/5/09 © *J.UCS*

Updates, Schema Updates and Validation of XML Documents – Using Abstract State Machines with Automata-Defined States

Klaus-Dieter Schewe
(Information Science Research Centre, Palmerston North, New Zealand
kdschewe@acm.org, isrc@xtra.co.nz)

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

Qing Wang
(University of Otago, Dunedin, New Zealand
qing.wang@otago.ac.nz)

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). In doing so we enable computations on abstract states that are defined by a certain class of automata, in this case VPAs. In this paper we elaborate on this approach by taking also updates of XML documents into account. In this way the ASM-approach combines vertical refinements, which first make states explicit and then instantiate by a specific EDTD, with horizontal refinements, which replace streaming XML documents by stored ones and then add updates. Furthermore, as the EDTD appears as part of the abstract state, updating it is another natural extension by horizontal refinement. In this way we obtain consistently integrated updates and schema updates for XML documents, which can even be extended to become fault-tolerant by taking at most k errors in the document into consideration. It further provides an example of ASM-based computation with automata-defined states.
Key Words: XML, Validation, Abstract State Machines
Category: E.m, H.2

1 Introduction

The eXtensible Markup Language (XML) has become a standard for the data exchange on the world-wide web. In order to prescribe a structure XML documents of interest must adhere to, an (optional) document type definition (DTD) can be used, but the language of DTDs has been often criticised for its limitations such as lack of typing and the use of global element definitions. The typing limitation has been addressed by the more expressive and now commonly used XSchema language, while global element type definitions are addressed by extended document type definitions (EDTDs) [Papakonstantinou and Vianu, 2000].

Whichever schema specification language is used, the problem of XML document validation, i.e. deciding whether a given document adheres to a given schema definition, has to be solved. For this EDTDs as the most general schema definition language provides the advantage to be associated with a specific class of push-down automata, the visibly pushdown automata (VPAs), as the tree languages specified by EDTDs are exactly those that can be recognised by a VPA [Kumar et al., 2007].

On these grounds it is straightforward to model the VPA for the validation of streaming XML documents with a given EDTD by an Abstract State Machine (ASM) [Börger and Stärk, 2003]. In our previous work in [Schewe et al., 2008] we demonstrated that by making states of the VPA explicit, i.e. defining a structure that captures the EDTD, we obtain an ASM specification that deals with all EDTDs in a uniform way, and the validation specification for a specific EDTD can be obtained by a simple instantiation.

This provides an example of ASM-based computations with automata-defined abstract states, i.e. while ASMs in general take first-order structures as abstract states, we are able to deal with restricted classes of such structures that are recognisable by a certain class of automata, in this case VPAs. The idea of automata-defined abstract states was put forward in [Wang et al., 2008]. In this article we build on this insight, and extend our previous work in the direction of updates, i.e. actual computations on these automata-defined states. This requires to relax the request to deal only with streaming XML documents that are read in sequentially. Instead of this we deal with stored XML document, so the storage structure and the explicit definition of the read-operation have to be added, which defines a "horizontal" refinement in the sense that a slightly more general problem is addressed.

Once we deal with stored XML documents it is a natural next step to permit updates to them, which then amount to a model of computation on automata-defined abstract states. As the EDTD itself is part of the state, we can also permit updating it, which leads to integrated schema updates. Still the general approach to validation guarantees that these updates are consistent in the sense that only XML documents will be produced that adhere to the current EDTD.

Finally, we may even pick up the idea from [Thomo et al., 2008] that was handled in our previous work in [Schewe et al., 2008] to permit validation up to k errors, where an error is defined by the need to insert, delete or modify a pair of opening and closing tags. As in Tomo's work we now handle both semantics, i.e. either count each such individual modification as one error, or treat all modifications with respect to the same tag pair as only one error.

In Section 2 we provide a brief overview of related work in order to place our current work into the literature. In Sections 3 and 4 we introduce preliminaries such as EDTDs and VPAs. Then Section 5 merely repeats our preliminary conference publication in [Schewe et al., 2008], i.e. we describe the ASM approach to streaming XML documents validation thereby outlining the necessary vertical refinements. In Section 6 we outline the concept of automata-defined abstract states and relate it to the validation ASMs. In particular, we relax the request to deal only with streaming XML documents, but instead take stored XML documents into account. In good database tradition it should be understood that storage should include the use of secondary storage devices. Section 7 is then devoted to the extension towards updates, first with respect to stored documents, then even to the stored EDTDs, which defines integrated document and schema updates as requested in [Kirchberg et al., 2005]. In Section 8 we pick up the problem of fault-tolerance by permitting up to k schema violations in a validated document. We first describe the case handled in [Schewe et al., 2008] counting each insertion, deletion or update of a pair of opening and closing tags as one error. Finally, in Section 9 we generalise the approach to the second semantics studied in [Thomo et al., 2008] dealing with the case of counting insertions, deletions and updates of tag pairs with the same name as only one error. This requires the construction of error tables. We conclude with a brief evaluation of our achievements and conclusions.

2 Related Work

The work reported in this article is related to three areas of research. The first one is the validation of XML documents, i.e. parsing the document and deciding, whether it is compatible with a given schema definition, which can be given by a DTD, an XSchema specification or an EDTD. Validation is a necessary step prior to querying with a standard XML query language such as XQuery, and updates.

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, and typical XQuery processors such as Xalan, Qizz, and Saxon 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 most common approaches to XML validation are given by the “Document Object Model” (DOM) [Gupta et al., 2003; Harold, 2002; van Kesteren, 2008; McCormack, 2008], a platform- and language-neutral API, which provides a standard set of interfaces for manipulating an XML document, and the “Simple API for XML” (SAX) [Box et al., 2000; Fegaras, 2004; Garshol, 2002; Harold, 2002; Simeoni et al., 2003; Wilde, 2004], a non-W3C standard API for streaming document processing.

In DOM 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 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.

The alternative SAX approach to DOM is event-based, i.e. parsing events are reported 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 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 (VPAs) [Kumar et al., 2007].

Extended Document Type Definitions (EDTD), introduced by Papakonstantinou and Vianu [Papakonstantinou and Vianu, 2000] 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. [Kumar et al., 2007] have shown that the tree languages specified by EDTDs are exactly the visibly pushdown languages (VPLs), i.e. those that can be recognised by a specialised class of push-down automata, the visibly pushdown automata (VPAs). 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 [Segoufin and Vianu, 2002; Kumar et al., 2007], 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 [Alur and Madhusudan, 2004].

This processing by finite state machines, transducers, or VPAs 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.

The second area of research related to our work is the use of Abstract State Machines as a general approach to produce reliable software [Börger and Stärk, 2003; Schellhorn, 2008]. In our previous work in [Schewe et al., 2008] we showed how the problem of validating streaming XML documents can be approached by using Abstract State Machines (ASMs). This is justified by the claim that dynamic construction of the parser based on a pattern specification of the rule that may parse a subexpression 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 modelled 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 [Börger, 2003]. 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 refined 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 doing so, we actually defined an ASM specification, in which the EDTD is part of the abstract states, and thus an instantiation with a specific EDTD could be used as a further refinement step to produce a validation machine for a specific EDTD. As these refinements did not extend the problem, i.e. the validation of streaming XML documents, we called them *vertical*.

In a second step in [Schewe et al., 2008] we generalised 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. Thomo et al. [Thomo et al., 2008] 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. They 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. We showed that the switch from the exact to the approximate validation problem for streaming XML documents gives rise to further ASM refinements, which we called *horizontal*, as the problem dealt with was enlarged.

This provided a rather simple example for the interplay of horizontal and vertical ASM refinements following the most prominent example of Java/JVM in [Börger et al., 2001], which has already guided recent research on software product lines [Batory and Börger, 2008]. This method is carried further to XML updates and schema updates in this article following previous work in [Kirchberg et al., 2005].

The third related research area is that of ASM theory. The work in [Wang and

Schewe, 2007; Schewe and Wang, 2008] lays the fundations for a general theory of database transformations in the line of thought of Gurevich's sequential and parallel ASM theses [Gurevich, 2000; Blass and Gurevich, 2003]. One particular idea that was brought up in [Wang et al., 2008] – with possibly significant implications regarding logical foundations [Wang and Schewe, 2008] and classification of computations – is to study computations with abstract states that are recognisable by a class of automata. The question would be how to combine these automata with ASMs. While the research in this area in general is in an infant stage, the present article can be regarded as exemplification of this research idea with VPAs as the automata class of interest. The vertical refinement explored in [Schewe et al., 2008] shows how to relate VPA-recognisable states, i.e. XML documents, to particular ASMs, and the present work integrates the recognition with general computation including updates on such documents. We even laid the grounds for integrated schema updates and fault-tolerant computations.

3 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 [Papakonstantinou and Vianu, 2000].

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 [Papakonstantinou and Vianu, 2000]) denotes a DTD with $\Sigma = \{\text{root}, \text{dealer}, \text{used\_cars}, \text{new\_cars}, \text{ad}, \text{model}, \text{year}\}$ and root 'root':

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

The assigned languages are

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

This corresponds to the (real) DTD

$\langle$!DOCTYPE dealer [
$\langle$!ELEMENT dealer (used_cars new_cars)$\rangle$
$\langle$!ELEMENT used_cars (ad*)$\rangle$
$\langle$!ELEMENT new_cars (ad*)$\rangle$
$\langle$!ELEMENT ad ((model year)|(model))$\rangle$
$\langle$!ELEMENT model PCDATA$\rangle$
$\langle$!ELEMENT year PCDATA$\rangle$
]$\rangle$

While such DTDs provide some schema information, they cannot express all desirable properties of XML documents. For instance, 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 [Papakonstantinou and Vianu, 2000] (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. For instance, 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 elements 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 [Papakonstantinou and Vianu, 2000]) denotes an EDTD with $\Sigma = \{\text{root}, \text{dealer}, \text{used\_cars}, \text{new\_cars}, \text{ad}, \text{model}, \text{year}\}$:

root: dealer	dealer: used_cars new_cars
used_cars: $(\text{ad}^u)^*$	new_cars: $(\text{ad}^n)^*$
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.

4 Visibly Pushdown Automata

Kumar et al. [Kumar et al., 2007] proved that the tree languages specified by EDTDs are exactly the visibly pushdown languages (VPLs), i.e. those that can be recognised by a specialised class of push-down automata, the visibly pushdown automata (VPAs). Following [Thomo et al., 2008] 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.

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) alphabet 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.

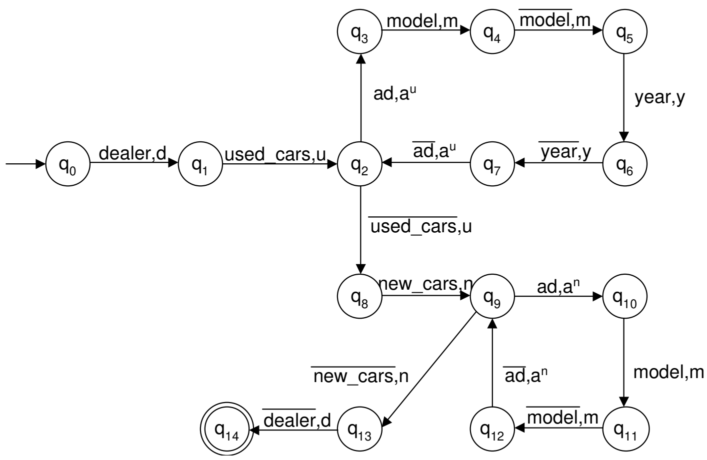

Figure 1: A VPA for validating streaming XML documents

Each transition gives rise to configuration pairs: $(q_1, a, q_2, \gamma) \in \tau_c$ gives rise to $((q_1, aw, v), (q_2, w, \gamma v))$, $(q_1, \bar{a}, \gamma, q_2) \in \tau_r$ gives rise to $((q_1, \bar{a}w, \gamma v), (q_2, w, v))$, and $(q_1, q_2) \in \tau_\epsilon$ 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 $\sigma_0, \dots, \sigma_f$ with a start configuration σ_0, a final configuration σ_f, and $(\sigma_{i-1}, \sigma_i) \in T$ for all $i = 1, \dots, f$.

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:

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

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

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

5 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 non-deterministic) 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 $\neq \perp$
  then choose $k \in \mathbb{N}$ 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. For instance, 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 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 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

sibling (continued)

younger	older	parent
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.

6 Automata-Defined Abstract States

Abstract State Machines (formerly called Evolving Algebras) were introduced by Yuri Gurevich as a means to capture the notion of algorithm in a precise way. The sequential ASM thesis [Gurevich, 2000] shows that sequential algorithms are captured by sequential ASMs, while the parallel ASM thesis [Blass and Gurevich, 2003] shows that ASMs in general capture parallel algorithms.

A decisive characteristic of ASMs is that the notion of state refers to first-order structures (or universal algebras). In [Schewe and Wang, 2008] a tailored ASM thesis for database transformations has been developed, and it has been suggested to investigate states as structures that are recognised by particular classes of automata. In particular, various classes of tree automata should turn out to capture database transformations on classes of XML databases.

In view of this theory the previous section defines an ASM for the class of VPAs. Structures recognised by a VPA and thus accepted by the ASM defined are exactly the XML documents that adhere to the EDTD defined by the element and sibling relations.

In order to turn to XML transformations we have to permit updates to the XML document at hand. So far, however, we only dealt with streaming XML documents, i.e. we tacitly assumed that tag is always bound to the next tag that is read in. In order to enable more than just validation, i.e. checking that we have a valid state, we have to define the read_next rule explicitly.

For this we need to store the XML document, which we do by means of three relations, which we add to the ASM signature as functions

doc_element(3), next(2), parent(2) controlled.

In doing so doc_element$(n, t, i) \neq \perp$ means that we have an element node in the XL document with tag n, type t and unique identifier i. There must always exist exactly one such element node doc_element(root,$\perp$,i_0) $\neq \perp$. The relation parent is used to represent the successor-predecessor relation, i.e. parent$(i, j) \neq \perp$ holds iff the node with identifier i is a direct successor (child) of the node with identifier j. Similarly, the relation next is used for the representation of directly adjacent siblings with the same parent node, i.e. next$(i, j) \neq \perp$ holds iff the nodes with identifiers i and j are children of the same parent node and i is the left neighbour of j.

Example 6. The following three tables represent an XML document that adheres to the EDTD in Example 2:

doc_element

name	type	id
root	$\perp$	0
dealer	$\perp$	1
used_cars	$\perp$	2
new_cars	$\perp$	3
ad	u	4
ad	u	5
ad	u	6
ad	n	7
model	$\perp$	9
year	$\perp$	10
model	$\perp$	11
year	$\perp$	12
model	$\perp$	13
year	$\perp$	14
model	$\perp$	15
model	$\perp$	16
ad	n	8

next

2	3
4	5
5	6
7	8
9	10
11	12
13	14

parent

1	0
2	1
3	1
4	2
5	2
6	2
7	3
8	3
9	4
10	4
11	5
12	5
13	6
14	6
15	7
16	8

It corresponds to the XML tree in Figure 2 modulo the exact values at leaf nodes, which we neglected so far. If leave nodes were to be considered as well, they would give rise to additional records in the doc_element relation such as (text, $\perp$, 17), additional records in the parent relation such as (17, 9), and an additional binary relation value with entries such as (17, "Ford Prefect") assigning a value to a unique leaf node identifier.

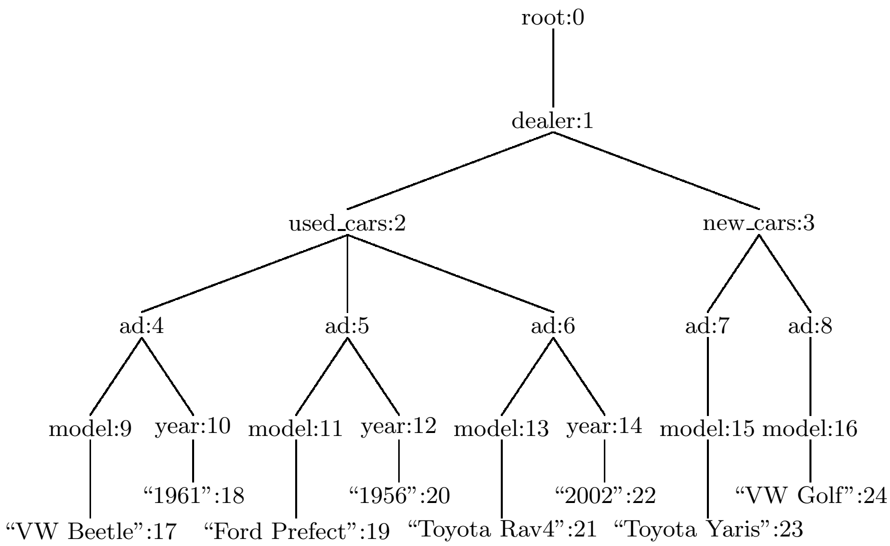

Figure 2: Tree representation of an XML document

In addition we will need five auxiliary variables, which will be added to the signature by

mode(0), tag(0), next_tag(0), active_id(0), tag_list(0) controlled

Note that tag(0) has now become a controlled function, while it was monitored, i.e. only controlled by the environment, before. We have to extend the initialisation by

mode := 0 ‖ next_tag := root ‖ tag_list := [] ‖
    choose i with doc_element(root,$\perp$,i) $\neq \perp$
        do active_id := i enddo

The crucial bit is the refinement of the rule next_tag(tag), which now becomes

if $\quad$ mode $= 0$
then tag := next_tag $\|$ tag_list = [next_tag]$^\frown$tag_list $\|$
 if $\quad \exists j$.parent(j, active_id) $\neq \bot \wedge \forall k$.next($k, j$) $= \bot$
 then active_id := j
 else search_id(active_id)
 endif ;
 choose n, t with doc_element(n, t, active_id) $\neq \bot$
  do next_tag := n enddo
else tag := $\overline{first(\text{tag\_list})}$ $\|$ tag_list := $rest$(tag_list) $\|$
  mode := mode $- 1$
endif

using the auxiliary rule

search_id(i) =
 if $\quad \exists k$.next(i, k) $\neq \bot$
 then active_id := k
 elsif $\exists i'$.parent(i, i') $\neq \bot$
 then search_id(i')
 else active_id := $\bot$
 endif $\|$
 mode := mode + 1

With this refinement the check rule now validates an XML document that is part of the structure by means of the controlled functions doc_element, next and parent, while the EDTD the document has to adhere to is represented by using the functions element and sibling. As in the previous section it is now no problem to eliminate the EDTD and to refine the check rule by instantiating it with the given fixed EDTD. This does not affect the refinement of the next_tag rule described above, i.e. no matter whether the XML document is streaming and read in tag by tag or stored as part of the state, the check rule remains the same. Example 5 shows how this will look like for the fixed EDTD from Example 2.

7 Integrated Updates and Schema Updates for XML

In the previous section we presented a refined check rule that will parse stored XML documents and only terminate successfully, if the document adheres to a given EDTD. In the general ASM for this problem the EDTD itself is also stored as part of the state, but this can be refined to obtain a check rule for a specific fixed EDTD. This defines automata-recognisable abstract states.

It is now straightforward to combine this generalised check rule with updates to the stored XML document. In general, the updates can be handled by a separate rule – call it update_rule – which has to be composed sequentually with the check rule. Thus, we obtain a rule of the form

update_rule ; check

If the stored XML document is considered to consitute a tree-based database, such a general update rule with follow-on validation will define a transaction, and a set of such transaction specifications would constitute a formal model of a database system in the sense of [Ma et al., 2008]. This could be completed by query rules of the form "check ; query_rule" which do not modify the stored XML document.

For the transactions it is of no importance, if we use stored EDTDs by means of the relations element and sibling or fix the EDTD by instantiation of the ASM. However, if we use stored EDTDs, we may update these as well, i.e. we combine the updates to the XML documents with schema updates as proposed in [Kirchberg et al., 2005]. This leads to rules of the form

(update_rule $\|$ schema_update_rule) ; check

Several rules of this form – maybe complemented by some query rules as above – again contitute a system of transactions.

Let us now illustrate updates and schema updates by some examples starting with the XML document in Example 6 – illustrated by the XML tree in Figure 2 – and the EDTD from Example 2, which we represented in Example 4. For simplicity, we combine update_rule and schema_update_rule into a single ASM rule.

Example 7. Assume we want to add addional ads for new and used cars that are read in one by one. We need a new variable car introduced into the signature as

car(0) monitored

and a not further specified read rule that is supposed to read in the next input (if it exists) and assign it to car. We further assume that all valid inputs are pairs (model, year), but the second component is $\perp$ for the case of new cars. This leads to the following instantiation of update_rule:

update_rule =
  read(car) ;
  if car $\neq \perp$
  then if $\exists m, y.\ \text{car} = (m, y) \wedge m \neq \perp \wedge y \neq \perp$
    then choose i, j, k with $\forall n, t.\ \text{doc\_element}(n, t, i) \neq \perp \wedge$
      $\text{doc\_element}(n, t, j) \neq \perp \wedge \text{doc\_element}(n, t, k) \neq \perp \wedge$
        $i \neq j \wedge i \neq k \wedge j \neq k$

do doc_element(ad,u,i) := 1 ‖ parent(j, i) := 1 ‖
doc_element(model,⊥,j) := 1 ‖ parent(k, i) := 1 ‖
doc_element(model,⊥,k) := 1 ‖ next(j, k) := 1 ‖
choose i' with doc_element(used_cars,⊥,i') $\neq \perp$
do parent(i, i') := 1 enddo ‖
if $\exists \ell$. doc_element(ad, u, ℓ) $\neq \perp \wedge \forall \ell'$. next($\ell, \ell'$) $= \perp$
then next(ℓ, i) := 1
endif
enddo
elsif $\exists m$. car $= (m, \perp) \wedge m \neq \perp$
then if $\exists m, y$. car $= (m, y) \wedge m \neq \perp \wedge y \neq \perp$
then choose i, j with $\forall n, t$. doc_element(n, t, i) $\neq \perp \wedge$
doc_element(n, t, j) $\neq \perp \wedge i \neq j$
do doc_element(ad,n,i) := 1 ‖ parent(j, i) := 1 ‖
doc_element(model,⊥,j) := 1 ‖
choose i' with doc_element(new_cars,⊥,i') $\neq \perp$
do parent(i, i') := 1 enddo ‖
if $\exists \ell$. doc_element(ad, n, ℓ) $\neq \perp \wedge \forall \ell'$. next($\ell, \ell'$) $= \perp$
then next(ℓ, i) := 1
endif
enddo
endif ; update_rule
endif

So far, we always ignored the concrete text values, thus also in this example the actual value m of the model and y of the year is not stored. It is a straightforward exercise to modify the rule in Example 7 in a way that takes concrete values into consideration.

Example 8. Let us now address a modification of the schema, i.e. the EDTD in Example 2. Assume that for new cars we always want to have a price, while for used cars a price can be optional. This leads to an EDTD with $\Sigma = \{$root, dealer, used_cars, new_cars, ad, model, year, year$\}$:

root: dealer	dealer: used_cars new_cars
used_cars: $(\text{ad}^u)^*$	new_cars: $(\text{ad}^n)^*$
ad^u: model year price$^?$	ad^n: model price

With respect to our EDTD representation in Example 4 we obtain the following schema update rule:

schema_update_rule =
choose i, j with $\forall n, t$. element(n, t, i) $= \perp \wedge$ element(n, t, j) $= \perp \wedge i \neq j$
do element(price,u,i) := 0 ‖ element(price,n,j) := 0 ‖

choose k, ℓ with element(ad,u,k) $= 0 \wedge$ element(year,$\perp$,ℓ) $= 0$
do sibling$(\ell, i, k) := 1 \parallel$ sibling$(i, \perp, k) := 1$ enddo $\parallel$
choose k, ℓ with element(ad,n,k) $= 0 \wedge$ element(model,$\perp$,ℓ) $= 0$
do sibling$(\ell, \perp, k) := \perp \parallel$ sibling$(\ell, j, k) := 1 \parallel$
sibling$(j, \perp, k) := 1$ enddo
enddo

Example 9. Let us now integrate the schema update from Example 8 with the document update from Example 7 using the rule

schema_update_rule ; update_rule ; check

For updating the stored EDTD we can use the rule from Example 8 without change, but update_rule for adding new car ads to the XML document in Example 7 has to be refined to capture the added price tags. This is done by the following refined rule, in which the only change is that the variable car is now bound to triples for model, year and price:

```
update_rule =
  read(car) ;
  if   car ≠ ⊥
  then if   ∃m, y, p. car = (m, y, p) ∧ m ≠ ⊥ ∧ y ≠ ⊥
       then choose i, j, k, ℓ with ∀n, t. doc_element(n, t, i) ≠ ⊥∧
                 doc_element(n, t, j) ≠ ⊥ ∧ doc_element(n, t, k) ≠ ⊥∧
                 doc_element(n, t, ℓ) ≠ ⊥ ∧ i ≠ j ∧ i ≠ k ∧ i ≠ ℓ
                      j ≠ k ∧ j ≠ ℓ ∧ k ≠ ℓ
            do doc_element(ad,u,i) := 1 ∥ parent(j, i) := 1 ∥
                 doc_element(model,⊥,j) := 1 ∥ parent(k, i) := 1 ∥
                 doc_element(model,⊥,k) := 1 ∥ next(j, k) := 1 ∥
                 if   p ≠ ⊥
                 then doc_element(price,⊥,ℓ) := 1 ∥
                      parent(ℓ, i) := 1 ∥ next(k, ℓ) := 1
                 endif ∥
                 choose i′ with doc_element(used_cars,⊥,i′) ≠ ⊥
                           do parent(i, i′) := 1 enddo ∥
                 if ∃k′. doc_element(ad, u, k′) ≠ ⊥ ∧ ∀ℓ′. next(k′, ℓ′) = ⊥
                 then next(k′, i) := 1
                 endif
            enddo
       elsif ∃m, p. car = (m, ⊥, p) ∧ m ≠ ⊥ ∧ p ≠ ⊥
  then if   ∃m, y. car = (m, y) ∧ m ≠ ⊥ ∧ y ≠ ⊥
       then choose i, j, k with ∀n, t. doc_element(n, t, i) ≠ ⊥∧
                 doc_element(n, t, j) ≠ ⊥ ∧ doc_element(n, t, k) ≠ ⊥∧
```

$i \neq j \wedge i \neq k \wedge j \neq k$
do doc_element(ad,n,i) := 1 $\|$ parent(j, i) := 1 $\|$
doc_element(model,$\perp$,j) := 1 $\|$ next(j, k) := 1 $\|$
doc_element(price,$\perp$,k) := 1 $\|$
choose i' with doc_element(new_cars,$\perp$,i') $\neq \perp$
do parent(i, i') := 1 enddo $\|$
if $\exists \ell$. doc_element(ad, n, ℓ) $\neq \perp \wedge \forall \ell'$. next($\ell, \ell'$) $= \perp$
then next(ℓ, i) := 1
endif
enddo
endif ; update_rule
endif

Note that the updated XML document resulting from the update in Example 9 will not be successfully validated by the check rule, unless prior to the update there were no ads for new cars. The check rule does validation against the updated EDTD and thus, price tags must be present for all new cars, but they only exist for newly inserted new cars.

There are three ways to address this problem:

1. Change the update rule requesting that in addition all existing new cars must be either deleted or receive a price tag. This means to write a complete new update rule for adding prices. We omit the simple details for this solution.

2. Relax the request that the resulting XML document after the update must adhere exactly to the updated EDTD, i.e. permit a bit of fault tolerance. We address this solution in the next two sections under two different semantics that differ in the way errors are counted. It seems most appropriate to consider the omission of price tags for new cars as a single error, no matter how often these omissions occur. This is the semantics handled in Section 9 with the number of errors being $k = 1$.

3. Relax the request that price is mandatory for new cars, which requires a minor change to the update_rule in the last example and the check rule. We omit the details.

Note that for the second solution with approximate validation the update rule in Example 9 will nonetheless not permit the insertion of new ads for new cars without price, so after several updates that delete or update certain ads the XML document will adhere to the updated EDTD. In other words, it is likely that the XML document will converge to a correct one. This constitutes some form of weak constraint enforcement. This convergence is not possible for the third solution.

8 Fault Tolerance and Error Correction

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. [Thomo et al., 2008] 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 [Thomo et al., 2008]), but instead illustrate the resulting VPA for the EDTD in Example 2.

Example 10. 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):

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

with start state $q_{0,0}$, final states $F = \{q_{14,2j} \mid 0 \leq j \leq k\}$. The following transitions (with $0 \leq j \leq 2k$) 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

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

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 $\perp$ 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,⊥,i_2) = 1
                then previous(depth+1) := ⊥
                endif)) ; check
    endif
```

for update:

```
check'(u) =
    read_next(tag) ;
    if count < k
    then  if ∃h.tag = new_h ∧
                ∃n, t_1, i_1, n_2, t_2, i_2 with element(n, t_1, i_1) = 1 ∧
                    element(n_2, t_2, i_2) ≠ ⊥∧
                    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 ∃h.tag = \overline{new}_h ∧
                ∃n_1, t_1, i_1, n_2, t_2, i_2 with element(new_h, t_1, i_1) = 1
                    ∧ element(n_2, t_2, i_2) = 2 ∧
                ∃x ≠ ⊥.change(n_1, t_1, new_h) = x ∧ ∀n', t', i', y.
                    (change(n', t', i') = y ⇒ y ≤ 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) := ⊥) ;
                previous(depth) := i_1 ;
                if sibling(i_1,⊥,i_2) = 1
                then previous(depth+1) := ⊥
                endif)) ; check
            endif
    endif
```

The cases for insertion and update use the rule read_next(tag) just as in the case of exact validation. Thus, refining the ASM to capture stored XML documents is done in exactly the same way as for the exact validation case, which we handled in Section 6. Then updates and schema updates as in Section 7 can be added as well.

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 11. 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

9 Fault Tolerance with Error Tables

The approach to fault tolerance in the previous section counts each violation to the EDTD as an individual error. The outlined validation procedure then accepts up to k such errors. However, it may also be the case that EDTD violations are applied more consistently in the sense that throughout the whole document a particular tag is always substituted by another one or omitted or added. This might create a far larger number of individual errors as acceptable by the validation procedure described so far, in particular for large XML documents.

Therefore, Thomo et al. proposed a different semantics for dealing with errors, i.e. EDTD violations in XML documents [Thomo et al., 2008]. According to this semantics we accept the deletion of a pair of tags, the insertion of a new pair of tags, and the update, i.e. name change, of a pair of tags as only one single error, no matter how often this error occurs.

Approximate validation of streaming XML documents with this semantics is tricky. As for the approxomate validation under the first semantics the VPA-based solution in [Thomo et al., 2008] is again the product of the VPA for exact validation as described in Section 4 with a VPT. In this case, however, a simple count for the errors that can be built into the states of the VPT, is insufficient, as we have to memorize insertions, updates and deletes that already occurred.

This is achieved by building VPTs for each pre-defined set of up to k such modifications and to superimpose them. In other words, the resulting product VPA accepts a set of XML documents that is the union of several sets, each of which reflects one particular error set.

We omit the details of this automata-theoretic solution and its representation by an ASM. Instead of this let us refine the general ASM for exact validation of streaming XML documents to one that permits up to k errors under the second semantics. In order to do this we can exploit the idea used for the automata-theoretic solution, which amounts to using error tables. As in the previous section we need an extension of the signature for this:

change(3) controlled   count(0) derived

Intially, change will be completely undefined, while the derived variable count is defined as the number of entries in the change relation, i.e. initially count will be 0:

$$\text{count} = \#\{(n, t, n') \mid \text{change}(n, t, n') \neq \bot\}.$$

During the processing of the check rule we add entries to the change relation, i.e. the error table, and as before change$(n, t, n') \neq \bot$ means that the tag with label n and type t has been changed to n' – we never modify the type t. The cases of insertions and deletions are captured by letting n or n' be $\bot$, respectively. Different to the previous section it is now irrelevant how often a certain change occurs.

With these error tables the refinement of the check rule becomes

check = choose $x \in \{0, i, d, u\}$ do check$''(x)$ enddo ;
    if count $\leq k$ then parse := 1 else parse := 0 endif

The values $0, id$ and u correspond to the cases of correct processing, insertion, delete and update, respectively. The rule check$''(0)$ is specified as for the exact validation keeping the call to the check rule, and the three error cases are specified as follows:

for insertion:
check$''(i)$ =
  read_next(tag) ;
  if $\exists h$.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 $\|$ change($\bot$,$\bot$,new$_h$) := 1 $\|$
    choose x with $\forall n', t', x$.element$(n', t', x) = \bot$ do
      element(new$_h$, $\bot$, 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$(\text{new}_h, \perp, i_1) = 1$ do
(element$(\text{new}_h, , \perp, i_1) := \perp \parallel$ element$(n, t, i) := 1 \parallel$
depth := depth - 1) enddo ; check
endif

for deletion:

check$''(d)$ =
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$(n, t, i) := 2 \parallel$ element$(n_2,t_2,i_2) := 1 \parallel$ previous(depth) := $i_2 \parallel$
depth := depth + 1 $\parallel$ change$(n_2, t_2, \perp) := 1$) ; check
elsif $\exists n, t, i, n_2, i_2, t_2$ with element$(n, t, i) = 1 \wedge$
element$(n_2,t_2,i_2) = 2 \wedge$ change$(n, t, \perp) \neq \perp$
then (element$(n_1,t_1,i_1) := 0 \parallel$ element$(n_2,t_2,i_2) := 1 \parallel$ depth := depth - 1) ;
previous(depth) := i_1 ;
if sibling$(i_1,\perp,i_2) = 1$
then previous(depth+1) := $\perp$
endif
endif ; check

for update:

check$''(u)$ =
read_next(tag) ;
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$(n, t_1, i_1) := 2 \parallel$ element$(\text{new}_h, t_2, i_2) := 1 \parallel$
previous(depth) := $i_2 \parallel$ change$(n_2, t_2, \text{new}_h) := 1$) ;
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$(\text{new}_h, t_1, i_1) = 1 \wedge$
element$(n_2, t_2, i_2) = 2 \wedge$ change$(n_1, t_1, \text{new}_h) \neq \perp$
then (element$(n_1,t_1,i_1) := 0 \parallel$ element$(n_2,t_2,i_2) := 1 \parallel$
depth := depth - 1) ;
previous(depth) := i_1 ;
if sibling$(i_1,\perp,i_2) = 1$
then previous(depth+1) := $\perp$
endif ; check
endif

As in the previous section the cases for insertion and update use the rule read_next(tag). Consequently, refining the ASM to capture stored XML documents and update them is done in exactly the same way as for exact validation as described in Section 6 and 7.

Furthermore, this ASM handles again any EDTD specified by means of the 'element' and 'sibling' functions, thus we can further refine it to obtain an ASM for approximate validation of streaming XML documents under a specific EDTD. All we have to do is to substitute for all cases in the check/check$''$ rules the possible values for tag, e.g. dealer, new_cars, etc., and eliminate identifiers.

Example 12. 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 $\|$ change(dealer,$\perp$,new_h) := 1) ; check

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

case state(model) = 1 $\wedge$ state(ad^u) = 2 $\wedge$ change(model,$\perp$,$\perp$) $\neq \perp$
then (state(model) := 0 $\|$ state(ad^u) := 1 $\|$
previous(3) := model $\|$ depth := 3 $\|$) ; check

10 Evaluation and Conclusions

In this article we showed how ASMs can be used for the specification of computations with abstract states defined by first-order structures that can be recognised by visibly pushdown automata (VPAs). Such states represent XML documents that adhere to a given extended document type definition (EDTD). Central to the approach is the validation of such documents, which we already addressed in our previous conference publication [Schewe et al., 2008]. We showed that the problems can be addressed by ASM refinements. It is straightforward to see that a VPA for XML document validation can be represented by an ASM, as ASMs provide a 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.

Practically speaking this showed that 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. More than that the elimination of automata in the approach ressembles vertical refinements in the Java/JVM study, as the refinements do not extend the problem. In [Schewe et al., 2008] we already demonstrated how this can be combined with horizontal refinements addressing the slightly more general problem of approximate XML document validation with up to k violations to the EDTD.

Theoretically, besides the successful interplay of horizontal and vertical refinements we laid the foundations for computations on automata-defined states, at least for the case of VPAs. Such computations were claimed in [Wang et al., 2008] to capture database transformations in general with classes of automata corresponding to data models. In particular, the handled case of VPAs would allow us to capture transformations on XML databases.

This article continues this line of thought and extends the ASM-based validation approach to effective computations by permitting updates both to the documents and their underlying schemata. We first had to permit stored documents and thus make the read-rule in the validating ASM explicit, then permit update rules. These additions constitute again horizontal refinements, thereby giving further evidence for the promising idea of combining vertical and horizontal refinements as a general development principle. It further exemplifies the idea of computations with automata-defined abstract states. We conclude that it is worthwhile to continue this line of research towards theoretical foundations of database transformations combining queries and updates. We further conclude that the combination of horizontal and vertical refinements on the basis of ASMs is a vital approach to practical systems development. The latter conclusion is further underlined by an extension to fault-tolerant computations extending the error-handling approach from [Schewe et al., 2008].

References

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

[Batory and Börger, 2008] Batory, D. and Börger, E. (2008). Modularizing theorems for software product lines: The JBook case study. *Journal of Universal Computer Science*, 14(12):2059–2082.

[Blass and Gurevich, 2003] Blass, A. and Gurevich, J. (2003). Abstract state machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 4(4):578–651.

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

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

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

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

[Fegaras, 2004] Fegaras, L. (2004). The joy of SAX. In *XIME*, Paris. ACM.

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

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

[Gurevich, 2000] Gurevich, J. (2000). Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111.

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

[Kirchberg et al., 2005] Kirchberg, M., Schewe, K.-D., and Tretiakov, A. (2005). Using XML to support media types. In Kaschek, R., Mayr, H. C., and Liddle, S. W., editors, *Information Systems Technology and its Applications – 4th International Conference, ISTA'2005*, volume 63 of *Lecture Notes in Informatics*, pages 101–113. GI.

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

[Ma et al., 2008] Ma, H., Schewe, K.-D., Thalheim, B., and Wang, Q. (2008). Abstract state services. In Song, I.-Y. et al., editors, *Advances in Conceptual Modeling – Challenges and Opportunities, ER 2008 Workshops*, volume 5232 of *LNCS*, pages 406–415. Springer-Verlag.

[McCormack, 2008] McCormack, C. (2008). Language bindings for DOM specifications. http://www.w3.org/TR/2008/WD-DOM-Bindings-20080410. Working Draft WD-DOM-Bindings-20080410.

[Papakonstantinou and Vianu, 2000] Papakonstantinou, Y. and Vianu, V. (2000). 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.

[Schellhorn, 2008] Schellhorn, G. (2008). ASM refinement preserving invariants. *Journal of Universal Computer Science*, 14(12):1929–1948.

[Schewe et al., 2008] Schewe, K.-D., Thalheim, B., and Wang, Q. (2008). Validation of streaming XML documents with abstract state machines. In Kotsis, G., Taniar, D., Pardede, E., and Khalil, I., editors, *Proceedings iiWAS 2008 – The 10th International Conference on Information Integration and Web-based Applications and Services*, pages 147–153, University of Linz, Austria. ACM.

[Schewe and Wang, 2008] Schewe, K.-D. and Wang, Q. (2008). A customised ASM thesis for database transformations. (submitted for publication).

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

[Simeoni et al., 2003] Simeoni, F., Lievens, D., Connor, R., and Manghi, P. (2003). Language bindings to XML. *IEEE Internet Computing*, 7(1):19–27.

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

[van Kesteren, 2008] van Kesteren, A. (2008). The XMLHttpRequest object. http://www.w3.org/TR/2008/WD-XMLHttpRequest-20080415. Working Draft WD-XMLHttpRequest-20080415.

[Wang and Schewe, 2007] Wang, Q. and Schewe, K.-D. (2007). Axiomatization of database transformations. In Börger, E. and Prinz, A., editors, *Proceedings ASM 2007*, University of Grimstad, Norway.

[Wang and Schewe, 2008] Wang, Q. and Schewe, K.-D. (2008). Towards a logic for abstract metafinite state machines. In Hartmann, S. and Kern-Isberner, G., editors, *Foundations of Information and Knowledge Systems – 5th International Symposium, FoIKS 2008*, volume 4932 of *Lecture Notes in Computer Science*, pages 365–380. Springer-Verlag.

[Wang et al., 2008] Wang, Q., Schewe, K.-D., and Thalheim, B. (2008). XML database transformations with tree updates. In Börger, E. et al., editors, *Abstract State Machines, B and Z – First International Conference, ABZ 2008*, volume 5238 of *Lecture Notes in Computer Science*, page 342. Springer-Verlag.

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