The VLDB Journal (2007) 16: 317–342
DOI 10.1007/s00778-005-0169-1

REGULAR PAPER

Christoph Koch · Stefanie Scherzinger

Attribute grammars for scalable query processing on XML streams

Received: 6 January 2005 / Accepted: 8 July 2005 / Published online: 2 February 2006

Abstract We introduce the notion of XML Stream Attribute Grammars (XSAGs). XSAGs are the first scalable query language for XML streams (running strictly in linear time with bounded memory consumption independent of the size of the stream) that allows for actual data transformations rather than just document filtering. XSAGs are also relatively easy to use for humans. Moreover, the XSAG formalism provides a strong intuition for which queries can or cannot be processed scalably on streams. We introduce XSAGs together with the necessary language-theoretic machinery, study their theoretical properties such as expressiveness and complexity, and discuss their implementation.

Keywords Stream processing · Query languages · Attribute grammars · XML

1 Introduction

In recent years, XML has become a standard format for document exchange and now seems to evolve into a popular representation language for streaming data as well. This development calls for flexible query languages for processing streams which support data transformations.

In [16, 25, 29], fragments of the standard XQuery language [34] are evaluated on XML streams. These fragments tend to support powerful data transformations. In consequence, query processing neither scales in terms of runtime nor memory consumption. Indeed, in these works, memory buffers are required that can grow arbitrarily large, depending on the amount of data communicated via stream.

This problem is due to the nature of XQuery, which renders it ill-suited for stream processing: Features such as nested for-loops with transitive paths (e.g. using descendant axis), which may lead to a nonlinearly-sized output, and nonlocal computations such as joins and the reordering and sorting of data cannot be handled scalably on streams. In addition, the syntax of XQuery makes it difficult for a user to tell whether a query can – at least in principle – be evaluated scalably, in linear time using little memory.

While XQuery was not designed to be evaluated on XML streams, the STX scripting language [12] employs a processing model where stylesheets are evaluated in a single pass over the input. Via memory buffers that store XML events, parts of the input stream may be revisited.

Yet query languages that require unbounded buffers constitute a scalability issue on streams. Thus, they are not in the spirit of the database community's quest for tailored formalisms that provide the appropriate tradeoffs between expressiveness and complexity for the data management challenge at hand.

XML streams may be *very* long, or should even be assumed to be infinite. For query processing to be feasible on streams, there is a need for special-purpose query languages and evaluation algorithms which *scale to streams*, that is,

(a) which can be evaluated strictly in linear time in the size of the input,
(b) which work by one linear forward scan of the data,
(c) and for which, at any time during query evaluation, memory consumption is bounded w.r.t. the length of the stream but not the depth of the XML tree.[1]

Among the models of computation that allow for better control of complexity than languages such as XQuery (see [22], where it is shown that already nonrecursive XQuery is NEXPTIME-hard. Thus, under widely held complexity-theoretic assumptions, XQuery is even exponentially harder than relational algebra. In any case, both XQuery and relational algebra support joins, and it is well known that even single-join queries cannot be processed scalably on streams,

C. Koch · S. Scherzinger (✉)
Lehrstuhl für Informationssysteme
Universität des Saarlandes, Saarbrücken, Germany
E-mail: scherzinger@infosys.uni-sb.de

[1] Note that a stack of memory proportional to the maximum depth of the XML tree is necessary for even the most basic sequential navigation and parsing tasks (see e.g. [18, 20, 21, 30]). To be precise, in (c) we thus call for memory consumption that is bounded w.r.t. the length of the stream but not the depth of the XML tree (an indication of its structural complexity). XML trees tend to be very shallow but wide, therefore such a stack is not considered a bottleneck to scalability.

cf. [19]), there are various forms of automata/transducers and certain attribute grammars. The former are, however, tedious to specify queries with, because their specifications tend to be large, technical, and hard to read. The latter approach is pursued in the current paper.

In this work, we develop and investigate a formalism for processing XML streams called *XML Stream Attribute Grammars* (XSAGs), a new class of attribute grammars specifically designed for scalable XML stream processing. XSAGs can be evaluated strictly in linear time in a streaming fashion, consuming only a stack of memory bounded by the depth of the XML tree being streamed. Thus, XSAGs satisfy our desiderata (a) through (c).

XSAGs are based on *extended regular tree grammars*, i.e., regular tree grammars in which the right-hand sides of productions may contain regular expressions, allowing the specification of nodes in the parse tree that have an unbounded number of children. Extended regular tree grammars are thus well-suited for specifying classes of unranked trees denoting XML documents. We assume that extended regular tree grammars are often available for XML streams in the dialect of *Document Type Definitions* (DTDs). This adds to the relevance of the present formalism.

An XSAG is obtained by annotating a given extended regular tree grammar with attribution functions, which define the output to be produced from the input stream when the annotated production is matched. In the tradition of L-attributed grammars [1], which can be evaluated in a single pass over the input, XSAGs are assumed to perform a single scan of the XML stream. This amounts to a single depth-first left-to-right traversal of the document tree.

Also in the tradition of L-attributed grammars, right-hand sides of productions can be annotated with *two* attribution functions. The first is placed at the beginning of the right-hand side and is executed when reaching the start tag of a node in the XML document (or equivalently, when descending into a subtree). The second is placed at the end of the right-hand side and is executed when reaching the corresponding end tag (or, equivalently, when returning from the depth-first left-to-right traversal of the subtree).

We illustrate the use of XSAGs by an example.

Example 1 (a) Consider the extended regular tree grammar $G = (Nt, T, P, bib)$ with nonterminals

$$Nt = \{bib, book, article, year, title, author\},$$

grammar start symbol *bib*, and terminals

$$T = \{\text{bib, book, article, year, title, author, } PCDATA\}.$$

The productions consist of a nonterminal on the left-hand side and a terminal on the right-hand side, with ϵ or a regular expression over nonterminals enclosed in parentheses. For instance, the production for grammar start symbol *bib* below declares that the root node is labeled "bib" and may have an arbitrary number of "book" or "article" children in no given order. Grammar G with productions P,

$$\begin{aligned} bib &::= \text{bib}((book \cup article)^*) \\ book &::= \text{book}(year.title.author.author^*) \\ article &::= \text{article}(year.title.author.author^*) \\ year &::= \text{year}(PCDATA) \\ title &::= \text{title}(PCDATA) \\ author &::= \text{author}(PCDATA) \end{aligned}$$

defines an XML bibliography database.

(b) By changing the first production to

$$bib ::= \{\texttt{ECHO}\}\ \text{bib}((book \cup article)^*)$$

we obtain an XSAG which simply outputs the input stream. Indeed, the start production matches the root node of the document and `ECHO` writes the entire subtree under the matched node to the output as XML.

(c) If we are only interested in books arriving on the stream we can use the XSAG obtained by changing the *bib* and *book* productions to

$$\begin{aligned} bib ::= &\{\texttt{print } \langle\texttt{books}\rangle\} \\ &\text{bib}\big((book \cup article)^*\big)\ \{\texttt{print } \langle\texttt{/books}\rangle\} \\ book ::= &\{\texttt{ECHO}\}\ \text{book}\big(year.title.author.author^*\big) \end{aligned}$$

Here, we apply `ECHO` to *book* subtrees, but not to articles. We explicitly output the start and end tags of the root node, and label the root node of the output produced by this XSAG "books", rather than "bib".

We extend the notion of basic XSAGs (bXSAGs) exemplified so far to introduce the *easy XSAGs* (yXSAGs). In yXSAGs, we may additionally annotate the regular expressions inside productions with attribution functions which adds to the flexibility of the formalism. To this end, we introduce the class of regular expressions which allow us to unambiguously assign the attribution functions to both symbols and operators while parsing the input stream with a lookahead of one token.

Example 2 The yXSAG with production

$$\begin{aligned} article ::= &\{\texttt{print } \langle\texttt{article}\rangle\} \\ &\text{article}\big((\{\texttt{ECHO}\}(year.title)). \\ &\qquad (\{\texttt{print } \langle\texttt{authors}\rangle\texttt{; ECHO}\} \\ &\qquad (author.author^*) \\ &\qquad \{\texttt{print } \langle\texttt{/authors}\rangle\})\big) \\ &\{\texttt{print } \langle\texttt{/article}\rangle\} \end{aligned}$$

outputs articles as they arrive on the stream, but groups the authors of each article under a common *authors* node. The second appearance of `ECHO` in the production applies to the tree region matched by the regular expression $author.author^*$, i.e., to the subtrees below *article* nodes that are rooted by *author* nodes.

In the previous examples, attributes were implicitly employed by the ECHO macro (see Sect. 4.4). The following example makes explicit use of attributes. In order to assure scalability in the strictest sense, we require that attributes range over a finite domain fixed with the XSAG.

Example 3 The following yXSAG

bib ::= {print ⟨books⟩}
bib((*book* ∪ *article*)*) {print ⟨/books⟩}
book ::= book(({MATCH_CHILDREN(2003, c)} *year*).
({if c = true then
begin print ⟨book⟩; ECHO end}
(*title*.*author*.*author**)
{if c = true then print ⟨/book⟩}))

(with the remaining productions as in Example 1(a)) outputs books published in "2003" with their *title* and *author* children, but without the *year*.

For a given *year* node, macro MATCH_CHILDREN sets the Boolean-valued condition attribute[2] *c* to true if the string of characters encountered while scanning the children of the *year* node from left to right matches "2003". Otherwise, *c* is set to false. This condition attribute is passed on during the traversal of the document tree.

The regular expression *title*.*author*.*author** describes a tree region among the children of a *book* node. Just before we first enter this tree region, we examine the value of *c*. If *c* is true as the current book has been published in 2003, we print start tag "⟨book⟩" and echo the tree region to the output. In this case, we further output end tag "⟨/book⟩" on leaving the tree region.

Note that this yXSAG is equivalent to the XQuery

```
⟨books⟩
{ for $x in //book
  where $x/year = "2003"
  return ⟨book⟩ {$x/title} {$x/author} ⟨/book⟩ }
⟨/books⟩
```

on documents conforming to our grammar.

Attribute grammars are well known in the field of compilers. Recently, they have been revisited in the context of XML, for instance for grammar-directed XML publishing [4, 5, 7] and querying [6, 33]. Some of their theory relevant in the context of structured documents has been studied in [14, 27, 28].

Our emphasis is on designing a *practical* formalism for query processing that is *relatively easy to use*. Attribute grammars are widely agreed to carry a strong intuition for specifying syntax-directed translations. In our setting, they provide a metaphor for strictly linear-time one-pass XML transformations that can be grasped very intuitively. This renders it relatively easy for a user to recognize or design queries which can be executed (scalably) on a stream, even if this intuition is paid for by our formalism being more operational than languages such as XQuery.

While ease of use cannot be conclusively asserted based only on our own observations and the examples we provide, alternative formalisms such as deterministic pushdown transducers (DPDTs) are unsuitable as query languages to be used by humans: Unless DPDTs are generated from queries (e.g. from XPath expressions as in [18]), the tediousness of specifying DPDTs by hand makes them unattractive as query languages; query processors for languages such as XML Query, on the other hand, do not scale to streams. We can therefore argue that XSAGs achieve our goal of relative ease of use. Already, bXSAGs are much more practical than DPDTs. yXSAGs permit very convenient and elegant nesting of attributions, which, as can be seen in Example 2 and others throughout the paper, allow us to specify many interesting data transformations conveniently.

Contributions

The technical contributions of this paper are as follows:

- We examine the framework of extended regular tree grammars appropriate for attribution and in the context of XML stream processing.
- In order to characterize yXSAGs properly, we introduce the notion of *strongly one-unambiguous regular expressions*. The strongly one-unambiguous regular expressions are precisely those for which the *parse tree* of a word (analogously to the derivation tree of a grammar) can be unambiguously constructed *online*, with just a one-symbol lookahead, while processing the stream. yXSAGs allow for attributions to be nested inside regular expressions by only permitting strongly one-unambiguous regular expressions on the right-hand sides of productions.
- We introduce and formally define our two notions of XML stream attribute grammars, bXSAGs and yXSAGs, which we compare with respect to usability.
- We define *XML-DPDT*s as deterministic pushdown transducers with a natural stack discipline that assures that the size of the stack remains strictly proportional to the depth of the XML tree and which can only accept well-formed XML documents. In a sense, *XML-DPDT*s capture the intuition of scalable XML stream processing and serve as an expressiveness yardstick for XSAGs.
- We show that both bXSAGs and yXSAGs are precisely as expressive as *XML-DPDT*s. XSAGs provide the same quasi-optimal trade-off between expressiveness and evaluation cost as do *XML-DPDT*s.
- Finally, we study the complexity of XSAG query evaluation and its implementation.

Structure

The structure of this paper basically follows the order of contributions described above. In Sect. 2 we study the

[2] In the technical sections of this paper, we will use a more explicit syntax when employing attributes (e.g. see Example 30).

language-theoretic foundations of one-unambiguous and strongly one-unambiguous regular expressions and classes of regular tree grammars suitable for the deterministic parsing of XML streams. In Sect. 3, we present an efficient method to check whether a regular expression is strongly one-unambiguous.

The subsequent sections explore the XSAG formalism: The syntax and semantics of basic and easy XSAGs are defined in Sect. 4. In Sect. 5, we introduce the *XML-DPDT*s as deterministic pushdown transducers for XML stream processing and show their equivalence to bXSAGs and yXSAGs. Thus we may evaluate XSAGs by translating them to *XML-DPDT*s. We discuss the complexity of XSAG evaluation in Sect. 6. In particular, we contrast the aforementioned technique with a hybrid evaluation where only the grammar component is translated to an *XML-DPDT* while the attribution functions are interpreted at runtime.

Section 7 concludes with a discussion of the XSAG formalism and future work.

2 Preliminaries

2.1 Regular expressions and one-unambiguity

We assume that *regular expressions* are constructed from a set of atomic symbols, using the concatenation operator, union operator, and the Kleene star, denoted ., ∪, and * respectively. We denote by $L(\rho)$ the language defined by the regular expression ρ and by $symb(\rho)$ the atomic symbols that occur in ρ.

By a *marking* of a regular expression ρ over alphabet Σ, we denote a regular expression ρ' such that each occurrence of an atomic symbol in ρ is replaced by the symbol with its position among the atomic symbols of ρ added as subscript. For instance, the marking of $(a \cup b)^*.a.a^*$ is $(a_1 \cup b_2)^*.a_3.a_4^*$. The reverse of a marking, indicated by $^\#$, is obtained by dropping the subscripts.

A regular expression ρ is called *ambiguous* [8] if there are two words $w_1, w_2 \in L(\rho')$ such that $w_1 \neq w_2$ but $w_1^\# = w_2^\#$. A regular expression is called *unambiguous* if it is not ambiguous.

Example 4 The language defined by the regular expression $\rho = (a \cup b)^*.a.a^*$ is ambiguous because the language defined by its marking $\rho' = (a_1 \cup b_2)^*.a_3.a_4^*$ contains the words $a_1.a_3$ and $a_3.a_4$ which correspond to the same word aa of $L(\rho)$.

Let ρ be a regular expression, ρ' its marking, and $\Sigma' = symb(\rho')$ the marked alphabet used by ρ'. Then ρ is called *one-ambiguous* [11] iff there are words u, v, w over Σ' and symbols $x \neq y \in \Sigma'$ with $x^\# = y^\#$ such that $uxv, uyw \in L(\rho')$. A regular expression is called *one-unambiguous* if it is not one-ambiguous.

Intuitively, a one-unambiguous regular expression ρ allows us to determine which atomic symbol in ρ matches the next symbol from an input word $w \in L(\rho)$ while we parse w from left to right with a lookahead of one token.

Example 5 Consider the regular expression $\rho = a^*.a$ and its marking $\rho' = a_1^*.a_2$. Let $u = a_1$, $x = a_2$, $v = \epsilon$, $y = a_1$, and $w = a_2$. Clearly, $uxv = a_1a_2$ and $uyw = a_1a_1a_2$ are both words of $L(\rho')$, so ρ is one-ambiguous. The equivalent regular expression $a.a^*$ is one-unambiguous.

For a given regular language L over alphabet Σ, the set $first(L)$ consists of precisely those symbols x such that there is a word w with $xw \in L$. For each $x \in \Sigma$, the set $follow(L, x)$ consists of those symbols y such that there are words v, w with $vxyw \in L$. Finally, $last(L)$ consists of those symbols x such that there is a word w with $wx \in L$.

Definition 1 ([11]) Let ρ be a regular expression and let ρ' be its marking. The *Glushkov automaton* of ρ is the nondeterministic finite-state automaton (NFA) $\mathcal{G}(\rho) = (Q, symb(\rho), \delta, q_0, F)$ with

- $Q = symb(\rho') \cup \{q_0\}$, i.e. the states of $\mathcal{G}(\rho)$ contain the marked symbols in ρ' and a new initial state q_0.
- For each $a \in symb(\rho)$,

$$\delta(q_0, a) = \{x \mid x \in first(\rho') \wedge x^\# = a\}.$$

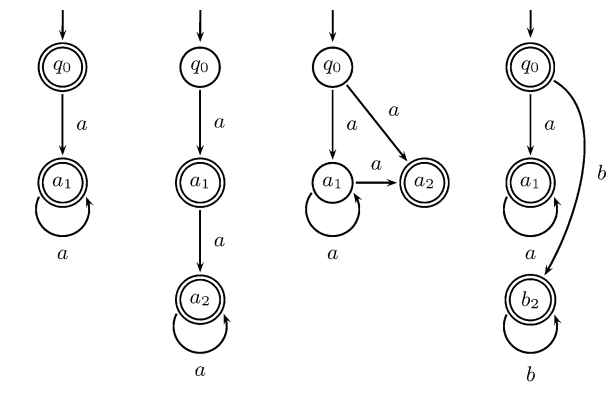
- For $x \in symb(\rho')$, $a \in symb(\rho)$,

$$\delta(x, a) = \{y \mid y \in follow(\rho', x) \wedge y^\# = a\}.$$

- $F = last(\rho') \cup \{q_0 \mid \epsilon \in L(\rho)\}$.

The Glushkov automaton $\mathcal{G}(\rho)$ for a regular expression ρ has no transitions that lead to the initial state and any two transitions that lead to the same state have identical labels [11]. This property is apparent for the Glushkov automata in Fig. 1.

Proposition 1 ([11]) *A regular expression ρ is one-unambiguous iff its Glushkov automaton $\mathcal{G}(\rho)$ is deterministic.*



(a) $\mathcal{G}((a^*)^*)$ (b) $\mathcal{G}(a.a^*)$ (c) $\mathcal{G}(a^*.a)$ (d) $\mathcal{G}(a^* \cup b^*)$

Fig. 1 Examples of Glushkov automata

(a) $a^* \cup b^*$. (b) $a.a^*$.

Fig. 2 Parse trees of regular expressions

Deterministic finite-state automata (DFAs) can be constructed from one-unambiguous regular expressions in just quadratic time [11]. Note that the Glushkov automaton for a one-unambiguous regular expression can be computed in even linear time [10] if the alphabet is considered fixed. However, we believe that this is not a realistic assumption in the context of grammars for XML data: for instance, in DTDs, there cannot be more productions than alphabet symbols.

Example 6 The regular expressions $(a^*)^*$, $a.a^*$, and $a^* \cup b^*$ are one-unambiguous as the Glushkov automata in Figs. 1a, b, and d are deterministic. However, the Glushkov automaton of the regular expression $a^*.a$ in Fig. 1c has several distinct transitions leading from states q_0 and a_1 under input symbol a, so it is not deterministic. Thus, $a^*.a$ is not one-unambiguous.

2.2 Strong one-unambiguity

Intuitively, by a *bracketing of a regular expression* ρ we refer to a *labeling of the nodes in the parse tree* of ρ using distinct indices. We realize this by assigning the indices in a depth-first left-to-right traversal of the parse tree. See Fig. 2 for two examples. The bracketing $\rho^{[]}$ is then obtained by inductively mapping each subexpression π of ρ with index i to $[_i.\pi.]_i$. Thus, a bracketing of a regular expression is a regular expression over the alphabet $\Sigma \cup \Gamma$, where $\Gamma = \{[_i,]_i \mid i \in \{1, 2, 3, \ldots\}\}$. We assume that Σ and Γ are disjoint.

Example 7 The bracketing of regular expression $a^* \cup b^*$, derived from the parse tree in Fig. 2a, is

$$[_1.\big(([_2.([_3.a.]_3)^*.]_2) \cup ([_4.([_5.b.]_5)^*.]_4)\big).]_1.$$

The bracketing of regular expression $a.a^*$

$$[_1.[_2.a.]_2.[_3.([_4.a.]_4)^*.]_3.]_1.$$

can be derived from the parse tree in Fig. 2b.

Let $^{\#[]}$ denote the reverse of the bracketing, obtained by dropping the brackets. Let $w \in L(\rho^{[]})$, then the subsequence of brackets in w is called a *bracketing of* $w^{\#[]}$.

Example 8 Consider $a^* \cup b^*$ from the previous example. For $\epsilon \in L(a^* \cup b^*)$, we may find the two bracketings $[_1[_2]_2]_1$ and $[_1[_4]_4]_1$, while the bracketing for $a \in L(a.a^*)$ is unique, namely $[_1[_2]_2[_3]_3]_1$.

There is a strong correspondence between the bracketing of a word and its parse tree (see Sect. 2.6). We now define the class of regular expressions for which we may unambiguously derive bracketings or parse trees for input words with just one token lookahead.

Definition 2 Let ρ be a regular expression and let $\rho^{[]}$ be its bracketing. A regular expression ρ is called *strongly one-unambiguous* iff there do not exist words u, v, w over $\Sigma \cup \Gamma$, words $\alpha \neq \beta$ over Γ, and a symbol $x \in \Sigma$ such that $u\alpha x v, u\beta x w \in L(\rho^{[]})$ or $u\alpha, u\beta \in L(\rho^{[]})$.

Example 9 The regular expression $\rho = a^* \cup b^*$ from Example 7 is one-unambiguous but it is not strongly one-unambiguous: The empty word in $L(\rho)$ can be matched as $[_1[_2]_2]_1 \in L(\rho^{[]})$ and as $[_1[_4]_4]_1 \in L(\rho^{[]})$, so $u = [_1$, $\alpha = [_2]_2]_1$, and $\beta = [_4]_4]_1$. The equivalent regular expression $a.a^* \cup b^*$ is strongly one-unambiguous.

Example 10 The bracketing of regular expression $\rho = (a^*)^*$ is $\rho^{[]} = [_1.([_2.([_3.a.]_3)^*.]_2)^*.]_1$. ρ is not strongly one-unambiguous, as for the word $aa \in L(\rho)$ there are words $[_1[_2[_3a]_3[_3a]_3]_2]_1$ and also $[_1[_2[_3a]_3]_2[_2[_3a]_3]_2]_1$ in $L(\rho^{[]})$, so $u = [_1[_2[_3a, \alpha =]_3[_3, \beta =]_3]_2[_2[_3, x = a$, and $v = w =]_3]_2]_1$.

Example 11 The bracketing of a regular expression is strongly one-unambiguous.

Note that in order to define strong one-unambiguity of a regular expression ρ, we could just as well consider a simpler version of $\rho^{[]}$ in which only opening, but not closing brackets are inserted, i.e., which is obtained by inductively mapping each subexpression π of ρ with index i to $[_i.\pi$. However, the symmetric brackets better suit our needs.

It is easy to see that the condition required for a regular expression to be strongly one-unambiguous is a strengthening of the condition for one-unambiguity:

Proposition 2 *Each strongly one-unambiguous regular expression is also one-unambiguous.*

Proof In the course of computing the bracketing $\rho^{[]}$ for a regular expression ρ, we assign a unique identifier to each node in the parse tree of ρ. In particular, there is such an identifier for each leaf node (corresponding to an atomic symbol that is marked in ρ'). W.l.o.g., we assume that the indices of brackets immediately surrounding atomic symbols in $\rho^{[]}$ are the same as the indices assigned to atomic symbols in ρ'. (For instance, for $\rho^{[]} = [_1.[_2.a.]_2.[_3.b.]_3.]_1$, ρ' must be $a_2.a_3$.)

Assume that ρ is not one-unambiguous. Then there are words u, v, w over Σ' and symbols $x_i, x_j \in \Sigma'$ with $i \neq j$ and $x_i^{\#} = x_j^{\#}$ such that $ux_iv, ux_jw \in L(\rho')$. Let us now consider the corresponding word in $L(\rho^{[]})$. It follows that there are words $\hat{u}, \hat{v}, \hat{w}$ over $\Sigma \cup \Gamma$ and α, β over Γ such that $\hat{u}\alpha[_i x]_i \hat{v}, \hat{u}\beta[_j x]_j \hat{w} \in L(\rho^{[]})$. Since $i \neq j$ and thus $\alpha[_i \neq \beta[_j$, by Definition 2, ρ is not strongly one-unambiguous. □

2.3 Extended regular tree grammars

We regard XML documents [9] without attributes as our data model. This causes no restriction on the applicability of our work, as attributes can be modeled as special children of a node which precede all other children. Throughout this paper, we will frequently use the terms XML document and XML stream synonymously.

We now introduce the *extended regular tree grammars*, a natural grammar mechanism for XML documents.

Definition 3 (ERTG, [26]) Let *Tag* be a set of node labels ("tags") and let *Char* be a set of characters distinct from the tags. An *extended regular tree grammar*[3] is a grammar $G = (Nt, T, P, s)$ where

1. Nt is a set of nonterminals,
2. $T = Tag \cup Char$ is a set of terminals,
3. P is a set of productions $nt ::= t(\rho)$ where $nt \in Nt$, $t \in T$, and
   - if $t \in Tag$ then ρ is either ϵ or a regular expression over alphabet Nt,
   - if $t \in Char$ then $\rho = \epsilon$ and $nt \neq s$, and
4. $s \in Nt$ is the grammar start symbol.

Definition 4 An XML document D is *well-formed* w.r.t. an ERTG G if $D \in L(G)$.

The restriction that the grammar start symbol of an ERTG may only derive tag nodes ensures that in each well-formed XML document there is at least one element (the root node) and that all XML start and end tags are properly nested. Furthermore, the first symbol in the document is the start tag for the root node. An XML document is *malformed* if it is not well-formed. The process of checking whether a document is well-formed with respect to a specific ERTG is called *validation*.

Remark 1 (PCDATA) We introduce a syntactic macro to describe character content of leaf nodes. Let the set of characters be $Char = \{c_1, \ldots, c_n\}$. As a shortcut, we define the regular expression macro $PCDATA := (\hat{c}_1 \cup \cdots \cup \hat{c}_n)^*$ using new nonterminals $\hat{c}_1, \ldots, \hat{c}_n$ and productions $\hat{c}_i ::= c_i(\epsilon)$ for each $1 \leq i \leq n$. *PCDATA* can be used just like a terminal on the right-hand sides of grammar productions. For sake of syntactic simplicity and brevity, we will consider *PCDATA* as a terminal as we already did in Example 1.

Example 12 The ERTG $G = (Nt, T, P, bib)$ with $Nt = \{bib, publication, year, title, author\}$, $T = \{$bib, book, article, year, title, author, *PCDATA*$\}$, grammar start symbol *bib*, and productions in P,

$$
\begin{aligned}
bib &::= \text{bib}(publication^*)\\
publication &::= \text{book}(year.title.author.author^*)\\
publication &::= \text{article}(year.title.author.author^*)\\
year &::= \text{year}(PCDATA)\\
title &::= \text{title}(PCDATA)\\
author &::= \text{author}(PCDATA)
\end{aligned}
$$

defines the same language as the grammar of Example 1. Note that the two *publication* productions carry different terminals (book and article) on their right-hand sides.

Example 13 Consider the ERTG $G = (Nt, T, P, bib)$ with $Nt = \{bib, book, citation, cited\_book, title, author\}$, further $T = \{$bib, book, citation, title, author, *PCDATA*$\}$, grammar start symbol *bib*, and productions in P,

$$
\begin{aligned}
bib &::= \text{bib}(book^*)\\
book &::= \text{book}(title.author.author^*.citation)\\
citation &::= \text{citation}(cited\_book^*)\\
cited\_book &::= \text{book}(title.author.author^*)\\
title &::= \text{title}(PCDATA)\\
author &::= \text{author}(PCDATA).
\end{aligned}
$$

G defines a bibliography database with two kinds of books: Books derived from a *book* production may again cite books, yet *cited_books* only consist of a title and one or more authors.

Here, the nonterminals *book* and *cited_books* define subtrees which share the same node label "book" yet differ in their children.

It is always possible to remove productions and nonterminals from an ERTG that cannot be reached from its grammar start symbol, thus constructing an equivalent *reduced* extended regular tree grammar [13]. In the following, we assume that all ERTGs are reduced.

2.4 Extended regular tree grammars for XML streams

Consider an ERTG with a nonterminal nt. Let $\theta(nt)$ denote the set of terminals t such that the grammar contains a production $nt ::= t(\rho_{nt,t})$ (where $\rho_{nt,t}$ identifies the regular expression for this particular production). Given a regular expression ρ, let $\tau(\rho)$ denote the regular expression in which each nonterminal nt in ρ is replaced by the union of terminals $\cup\theta(nt)$.

Example 14 Consider the ERTG from Example 12. For regular expression $\rho = publication^*$ from the right-hand side of the *bib* production, $\tau(\rho) = (\text{book} \cup \text{article})^*$.

Clearly, if $\tau(\rho)$ is (strongly) one-unambiguous, so is ρ, while the reverse does not necessarily hold.

[3] Strictly speaking we use a two-sorted alphabet of characters and tags, but we still consider these grammars ERTGs.

Each extended regular tree grammar can be alternatively considered as an extended context-free word grammar (CFG) which is obtained by simply rewriting each $tag(\rho)$ in the right-hand side of a production into $\langle tag\rangle\rho\langle/tag\rangle$. *Deterministic* context-free languages are precisely those recognizable by the deterministic pushdown automata (DPDA, see e.g. [2]). DPDAs run comfortably on streams requiring only a stack of memory bounded by the depth of the input tree. Using automata, we can thus scalably recognize the deterministic context-free languages.

The problem of processing an (extended) attribute grammar on a document requires an additional, different restriction on the grammar besides determinism to allow for deterministic computation: We need to unambiguously refer to the *atomic symbols* in the regular expressions to be able to access or assign attributes. In attribute grammars, a straightforward solution [27] is to require for right-hand side regular expressions ρ that $\tau(\rho)$ be *unambiguous*.

Example 15 Consider a grammar with productions

$$\begin{aligned} bib &::= \text{bib}((book_1 \cup book_2)^*) \\ book_1 &::= \text{book}(\epsilon) \\ book_2 &::= \text{book}(\epsilon) \end{aligned}$$

The regular expression $(book_1 \cup book_2)^*$ is unambiguous, but $\tau((book_1 \cup book_2)^*) = (\text{book} \cup \text{book})^*$ is not. Therefore, when processing the tags of the children of the *bib* node, we cannot determine which of the two *book* productions (with their possibly different attributions when we consider attribute grammars) are to be applied.

On streams, we cannot look ahead beyond a nonterminal (as the nonterminal may stand for a large subtree that we do not want to buffer) when parsing the input. Thus, we will assume the stronger notion of *one-unambiguity* for regular expressions $\tau(\rho)$ in the definition of TDLL(1) grammars below. That is, we will require that $\tau(\rho)$ can be unambiguously parsed with just one symbol of lookahead.

Definition 5 ([24]) A *TDLL(1) grammar*[4] is an extended regular tree grammar $G = (Nt, T, P, s)$ where $\tau(s)$ is unambiguous[5] and in which for each regular expression ρ on the right-hand side of a production, $\tau(\rho)$ is one-unambiguous.

Example 16 The grammars of Examples 1, 12, and 13 are TDLL(1). Meanwhile, since the grammar of Example 15 contains a regular expression ρ such that $\tau(\rho)$ is not even unambiguous, that grammar is not TDLL(1).

TDLL(1) grammars allow us to use attributed extended regular tree grammars on XML streams. However, as we show in the following section, the ability to use attribution functions *inside* the regular expressions in the right-hand sides of productions will allow us to write many practical queries in a much more user-friendly fashion. Our machinery for achieving this is the novel notion of STDLL(1) grammars, where we require for a right-hand side regular expression ρ that $\tau(\rho)$ is *strongly one-unambiguous*.

Definition 6 An *STDLL(1) grammar* is an extended regular tree grammar $G = (Nt, T, P, s)$ where $\tau(s)$ is unambiguous[5] and in which for each regular expression ρ on the right-hand side of a production, $\tau(\rho)$ is strongly one-unambiguous.

As all strongly one-unambiguous regular expressions are also one-unambiguous, all STDLL(1) grammars are obviously also TDLL(1) grammars.

Example 17 The grammars of Examples 1, 13, and 12 are also STDLL(1) grammars.

2.5 Document type definitions

Document type definitions (DTDs) are a special dialect of extended regular tree grammars. For XML elements that exclusively have elements as children (but no character data), the W3C recommendation[6] explicitly requires one-unambiguous *content models* (that is, right-hand side regular expressions) in order to assure compatibility with SGML.

Productions defining elements which contain character data, so-called *mixed-content models*, must be constructed according to either the pattern

$$nt_0 ::= (\text{PCDATA} \cup nt_1 \cup \cdots \cup nt_m)^*$$

or $nt_0 ::= \text{PCDATA}$ (where $nt_0, \ldots, nt_m$ are distinct DTD element names, i.e., nonterminals). Clearly, regular expressions such constructed are one-unambiguous.

Since DTDs also contain at most one production $nt ::= t(\rho)$ for each "element" t, so if regular expression ρ over nonterminals is one-unambiguous, the regular expression $\tau(\rho)$ over terminals is also one-unambiguous. Thus, DTDs are TDLL(1) grammars (see also [24]).

In fact, we suspect that most practical DTDs actually use only strongly one-unambiguous regular expressions in productions which makes them STDLL(1) grammars. Strong one-unambiguity is only a short way from one-unambiguity, and many of the most widely used forms of regular expressions are actually strongly one-unambiguous, e.g. consider regular expressions of the form $(e_1 \cup \cdots \cup e_m)^*$ where $e_1, \ldots, e_m$ are distinct element names.

However, the definition of macro *PCDATA* in Remark 1 causes mixed-content models to be strongly one-ambiguous. For now, we define mixed content in the productions of STDLL(1) grammars using the syntactic macro "$PCDAT := \hat{c}_1 \cup \cdots \cup \hat{c}_n$" instead of *PCDATA*, and refer to Sect. 4.2 for further discussion.

Figure 3 gives an overview of the classes of ERTGs considered in this paper.

[4] TDLL(1) is an acronym for **t**op-down **l**eft-to-right parsing, yielding the **l**eftmost derivation tree, with a **1**-token lookahead.

[5] Note that $\tau(s)$ is guaranteed to be a very simple form of regular expression, a disjunction of atomic symbols. In this special case, unambiguity implies one-unambiguity and even strong one-unambiguity.

[6] Sections. 3.2.1, 3.2.2, and Appendix E in [9].

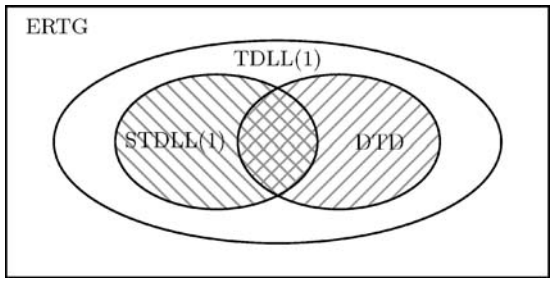
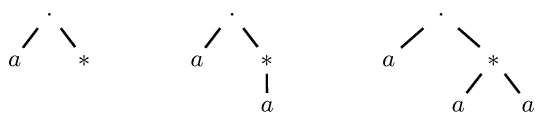


Fig. 3 Subclasses of extended regular tree grammars



(a) a (b) aa (c) aaa

Fig. 4 Parse trees for words in $L(a.a^*)$

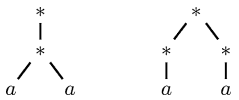


Fig. 5 Parse trees for word aa in $L((a^*)^*)$

2.6 TDLL(1) and STDLL(1) parse trees

Given a regular expression π defined over atomic symbols T, we obtain an equivalent (extended) *regular grammar* $G = (V, T, P, \underline{\pi})$ by recursively decomposing π into productions P,

$$\underline{\rho_1.\rho_2} ::= \underline{\rho_1}\ \underline{\rho_2} \qquad \underline{\rho_1 \cup \rho_2} ::= \underline{\rho_1} \mid \underline{\rho_2} \qquad \underline{\rho^*} ::= \underline{\rho}^*$$

for regular expressions ρ, ρ_1, and ρ_2. Here, by $\underline{\rho}$, we refer to a symbol of $V \cup T$ rather than a regular expression. The nonterminals V consist precisely of the symbols $\underline{\rho}$ such that ρ is non-atomic. (For the special case that π is atomic, P is empty and the start symbol $\underline{\pi}$ is allowed to be a terminal. This is somewhat nonstandard but fits ours needs.)

Regular grammars provide us with a natural way of assigning parse trees to words. For the sake of simplicity, we will use interior nodes of the forms "$\cup$", "$.$", and "$*$", rather than nonterminals $\underline{\rho_1 \cup \rho_2}$, $\underline{\rho_1.\rho_2}$, and $\underline{\rho^*}$.

Example 18 Figure 4 shows parse trees for words a, aa, and aaa in $L(a.a^*)$.

Example 19 Consider the regular expression $\rho = (a^*)^*$ which is not strongly one-unambiguous. Figure 5 shows two alternative parse trees for word $aa \in L(\rho)$. In contrast, the parse trees for words defined by strongly one-unambiguous regular expressions, as in the previous example, are unique.

For TDLL(1) grammars, the parse trees are simply the usual *document trees* associated with XML documents. However, the parse trees for STDLL(1) grammars also incorporate the parse trees for matching the input against the regular expressions on the right-hand sides of productions. If a regular expression contains a nonterminal nt, we assign it the terminal t as its unique child in the parse tree if the production used to rewrite nt is of the form $nt ::= t(\rho)$.

We illustrate the two forms of parse trees by an example.

Example 20 Consider the XML document

```
⟨bib⟩
 ⟨article⟩
  ⟨year/⟩⟨title/⟩⟨author/⟩⟨author/⟩⟨author/⟩
 ⟨/article⟩
⟨/bib⟩
```

and the ERTG G from Example 1. Then we may observe the following:

1. If G is viewed as a TDLL(1) grammar, the document parses into the tree depicted in Fig. 6a.
2. The parse tree for the same document, if G is viewed as an STDLL(1) grammar, is shown in Fig. 6b, where nodes associated with nonterminals are set in italics. Here, we assume that the operation "." associates to the right and that the production

   $$\textit{article} ::= \text{article}(\textit{year}.\textit{title}.\textit{author}.\textit{author}^*)$$

   is thus equivalent to

   $$\textit{article} ::= \text{article}(\textit{year}.(\textit{title}.(\textit{author}.(\textit{author}^*)))).$$

3. If the *article*-production of G were changed to

   $$\textit{article} ::= \text{article}((\textit{year}.\textit{title}).(\textit{author}.\textit{author}^*))$$

   then the TDLL(1) parse tree for the above XML document would remain unchanged. However, the STDLL(1) parse tree would differ from Fig. 6b in the subtree rooted at the node with subscript 5, as shown in Fig. 6c.

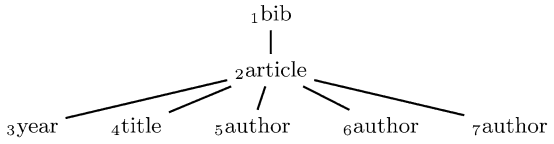
3 Checking for strong one-unambiguity

Next we describe our algorithm for checking whether a given regular expression is strongly one-unambiguous.

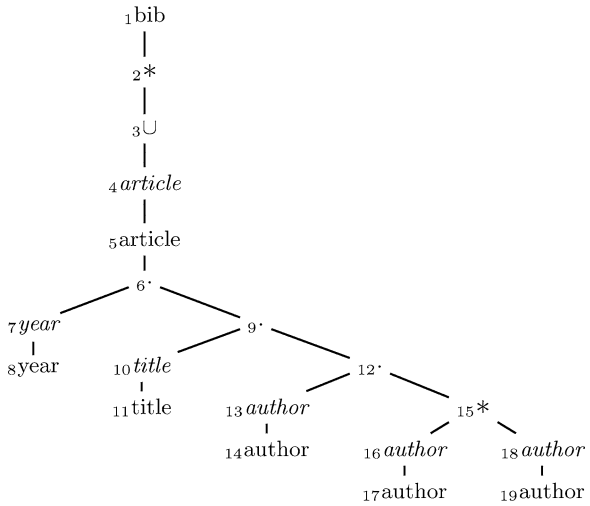
Given a regular expression ρ, let $\odot$ be a new symbol that does not occur in ρ. In the following, we require that all input words are terminated by symbol $\odot$ to precisely capture the end of words. When processing XML, $\odot$ has the role of reading the closing tag of the parent node.

In a Glushkov automaton $\mathcal{G}(\rho^{[]}.\odot)$, i.e. of the bracketing, with state set Q, initial state q_0, and the single final state $\odot'$, let $\hat{Q} = \{q \in Q \mid q \in \mathit{symb}(\rho')\} \cup \{q_0\} \cup \{\odot'\}$ denote the *non-auxiliary* states, i.e., the states not introduced for brackets. Let $\delta^{\mathcal{G}}$ be the transition function of $\mathcal{G}(\rho^{[]}.\odot)$.

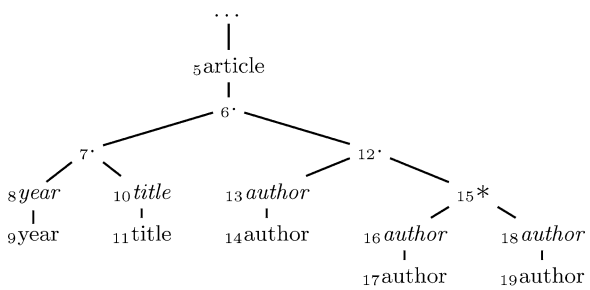
For a DFA with transition function δ and an input word $w = w_1 \ldots w_n$, we will use the notation $\delta^*(q, w)$ as a shortcut for $\delta(\ldots\delta(\delta(q, w_1), w_2), \ldots, w_n)$.



(a) Parse tree for TDLL(1) grammar.



(b) Parse tree for STDLL(1) grammar.



(c) Parse tree for modified STDLL(1) grammar.

Fig. 6 Parse trees of Example 20

Proposition 3 *Let ρ be a regular expression. Then ρ is strongly one-unambiguous iff for $\mathcal{G}(\rho^{[]}.\odot)$ there do not exist words $\alpha \neq \beta$ over Γ, and states $x \in \hat{Q}$ and $y, z \in \hat{Q} \setminus \{q_0\}$ with $y^{\#} = z^{\#}$, such that $(\delta^{\mathcal{G}})^*(x, \alpha y^{\#}) = y$ and $(\delta^{\mathcal{G}})^*(x, \beta z^{\#}) = z$.*

Observe that y and z do not have to be distinct.

Proof Let ρ be a regular expression and let $\mathcal{G}(\rho^{[]}.\odot)$ be a Glushkov automaton recognizing $L(\rho^{[]}.\odot)$.

$\Rightarrow$ Assume that ρ is not strongly one-unambiguous, then there exist words r, s, t over $\Sigma \cup \Gamma$, words $\alpha \neq \beta$ over Γ, and a symbol $x \in \Sigma$ such that (1) $r\alpha x s, r\beta x t \in L(\rho^{[]})$ or (2) $r\alpha, r\beta \in L(\rho^{[]})$. Let $r = r_1 \dots r_n$. We define a state $q \in \hat{Q}$ and words $\hat{\alpha} \neq \hat{\beta} \in \Gamma^*$ as follows:

(a) If $r \in \Gamma^*$, then $q := q_0$ and $\hat{\alpha} := r\alpha$ and $\hat{\beta} := r\beta$,

(b) otherwise, r contains at least one symbol from Σ. Let r_i be the rightmost symbol from Σ in r, i.e. $r_{i+1}, \dots, r_n \in \Gamma$. Then $q := \delta(q_0, r_1 \dots r_i)$. If $i = n$ then $\hat{\alpha} := \alpha$ and $\hat{\beta} := \beta$, otherwise $\hat{\alpha} := r_{i+1} \dots r_n \alpha$ and $\hat{\beta} := r_{i+1} \dots r_n \beta$.

Thus in case (1), there are transitions $(\delta^{\mathcal{G}})^*(q, \hat{\alpha}x) = y$ and $(\delta^{\mathcal{G}})^*(q, \hat{\beta}x) = z$ where y and z are in $\hat{Q} \setminus \{q_0\}$ and $x = y^{\#} = z^{\#}$. Similarly, in case (2), there are transitions $(\delta^{\mathcal{G}})^*(q, \hat{\alpha}\odot) = \odot'$ and $(\delta^{\mathcal{G}})^*(q, \hat{\beta}\odot) = \odot'$ where $\odot'$ is in $\hat{Q} \setminus \{q_0\}$.

$\Rightarrow$ Let $q \in \hat{Q}$ and let u be a word over $\Sigma \cup \Gamma$ such that $(\delta^{\mathcal{G}})^*(q_0, u) = q$. Assume that there are words $\alpha \neq \beta$ over Γ and states $y, z \in \hat{Q} \setminus \{q_0\}$ with $y^{\#} = z^{\#} = x$ such that $(\delta^{\mathcal{G}})^*(q, \alpha y^{\#}) = y$ and $(\delta^{\mathcal{G}})^*(q, \beta z^{\#}) = z$. It follows from the construction of Glushkov automata that we may reach a final state from any given state. That is, there are words v, w over $\Sigma \cup \Gamma$ such that $(\delta^{\mathcal{G}})^*(y, v) = (\delta^{\mathcal{G}})^*(z, w) = \odot'$ where $\odot'$ is the single final state of $\mathcal{G}(\rho^{[]}.\odot)$. But then there are words $u\alpha x v$ and $u\beta x w \in L(\rho^{[]}.\odot)$. Consequently, ρ is not strongly one-unambiguous. □

Example 21 $\rho = (a^*)^*$ is a one-unambiguous regular expression, because the Glushkov automaton $\mathcal{G}(\rho)$ in Fig. 1a is deterministic. However, the Glushkov automaton $\mathcal{G}(\rho^{[]}.\odot)$ in Fig. 7 has several distinct paths between pairs of states in $\hat{Q} = \{q_0, a_1, \odot'\}$, e.g. $]_3[_3 a$ and $]_3]_2[_2[_3 a$ leading from state a_1 back to the same state. So by Proposition 3, ρ is not strongly one-unambiguous.

A finite-state transducer (FST) is a NFA with output which issues a fixed word w in each transition $q \xrightarrow{a/w} q'$ from state q to q' on input symbol a. A deterministic finite-state transducer (DFT) is an FST that is deterministic, i.e., a DFA if the output is ignored and for which no two transitions $q \xrightarrow{a/v} q'$ and $q \xrightarrow{a/w} q'$ exist such that $v \neq w$.

Theorem 1 *For each strongly one-unambiguous regular expression ρ there exists a DFT $\mathcal{A}^{[]}(\rho)$ which*

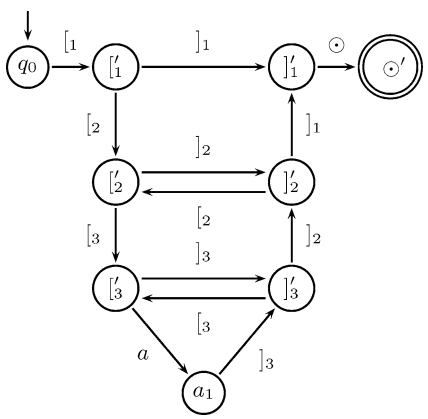


Fig. 7 Glushkov automaton of Example 21

1. *recognizes* $L(\rho.\odot)$ *and*
2. *outputs the bracketing of word* w *for input* $w.\odot \in L(\rho.\odot)$.

Proof Let ρ be a strongly one-unambiguous regular expression over alphabet Σ and let $\rho^{[]}$ be its bracketing over alphabet $\Sigma \cup \Gamma$.

Let $\mathcal{G}(\rho^{[]}.\odot)$ be a Glushkov automaton which recognizes $L(\rho^{[]}.\odot)$. $\mathcal{G}(\rho^{[]}.\odot)$ has state set Q, initial state q_0, transition function $\delta^{\mathcal{G}}$, and the single final state $\odot'$. By Proposition 1, as $\rho^{[]}.\odot$ is (strongly) one-unambiguous $\mathcal{G}(\rho^{[]}.\odot)$ is deterministic.

Let $\mathcal{A}^{[]}(\rho) = (\hat{Q}, \hat{\Sigma}, \Gamma, \hat{\delta}, q_0, \{\odot'\})$ be the DFT with

- state set $\hat{Q}$ as the non-auxiliary states of $\mathcal{G}(\rho^{[]}.\odot)$,
- input alphabet $\hat{\Sigma} = \Sigma \cup \{\odot\}$, output alphabet Γ,
- the transition function $\hat{\delta}$ mapping from $\hat{Q} \times \hat{\Sigma}$ to finite subsets of $\hat{Q} \times \Gamma^*$ such that

$$\hat{\delta}(q, a) := \{\langle q, w\rangle \mid (\delta^{\mathcal{G}})^*(q, wa) = q'\},$$

  i.e., iff state q' is reachable from q via input word wa, where $q, q' \in \hat{Q}$, atomic symbol $a \in \hat{\Sigma}$, and word of brackets $w \in \Gamma^*$, and
- the same initial and final state as $\mathcal{G}(\rho^{[]}.\odot)$.

We now verify the correctness of our claims. Below, given an FST $\mathcal{A}$, let $\alpha(\mathcal{A})$ denote the NFA obtained from $\mathcal{A}$ by ignoring its output.

1. The NFA $\alpha(\mathcal{A}^{[]}(\rho))$ is precisely the Glushkov automaton $\mathcal{G}(\rho.\odot)$ which we know recognizes $L(\rho.\odot)$. Therefore, $\alpha(\mathcal{A}^{[]}(\rho))$ recognizes the same language.
2. We argue that $\mathcal{A}^{[]}(\rho)$ is deterministic:
   i. As $\rho.\odot$ is one-unambiguous, $\alpha(\mathcal{A}^{[]}(\rho))$ is deterministic by Proposition 1.
   ii. As ρ is also strongly one-unambiguous, it follows from Proposition 3 that there are no words $\alpha \neq \beta$ over Γ and states $x \in \hat{Q}$, and $y, z \in \hat{Q} \setminus \{q_0\}$ with $a = y^{\#} = z^{\#}$, s.t. $(\delta^{\mathcal{G}})^*(x, \alpha a) = y$ and $(\delta^{\mathcal{G}})^*(x, \beta a) = z$. Thus, there are no transitions $\langle y, \alpha\rangle, \langle z, \beta\rangle \in \hat{\delta}(x, a)$ where $\alpha \neq \beta$.
3. The construction of $\mathcal{A}^{[]}(\rho)$ renders it easy to see that for a strongly one-unambiguous regular expression ρ, the output on a word $w.\odot \in L(\rho.\odot)$ is the bracketing of w. □

Theorem 2 *Let* ρ *be a regular expression. There is an* $O(|\rho|^3)$ *algorithm that checks whether* ρ *is strongly one-unambiguous, and if so, outputs* $\mathcal{A}^{[]}(\rho)$.

Proof Let ρ be a regular expression. We examine the steps involved in the construction of $\mathcal{A}^{[]}(\rho)$.

1. We compute $\rho^{[]}$ in time $O(|\rho|)$.
2. We may compute the Glushkov automaton $\mathcal{G}(\rho^{[]}.\odot)$ in quadratic time. If $\mathcal{G}(\rho^{[]}.\odot)$ is not deterministic, then ρ is not one-unambiguous by Proposition 1 and consequently cannot be strongly one-unambiguous (see Prop. 2).
3. Let $\delta^{\mathcal{G}}$ be the transition function of $\mathcal{G}(\rho^{[]}.\odot)$. Let Q be the state set of $\mathcal{G}(\rho^{[]}.\odot)$ and let $\hat{Q}$ be the subset of non-auxiliary states.
   If ρ is not strongly one-unambiguous, then there may be infinitely many output words for a transition between two states for a given word of input symbols. So as soon as (any) two distinct output words v and w over Γ have been discovered with $p \in (\delta^{\mathcal{G}})^*(q, vx)$ and $p \in (\delta^{\mathcal{G}})^*(q, wx)$, with $q, p \in \hat{Q}$ and $x \in \Sigma$, we know that ρ is not strongly one-unambiguous.
   More formally, let G be the directed graph with nodes Q and edges $\{\langle q, p\rangle \mid \exists q, p \in Q : \delta^{\mathcal{G}}(q, a) = p\}$. For each state $q \in \hat{Q}$, we compute a spanning (at best) subtree T_q of G, where states in $\hat{Q}$ must be leaves. Obviously, we only visit each edge at most once (but not passing through nodes of $\hat{Q}$), and if we return to a node already visited, ρ is not strongly one-unambiguous and the computation terminates.
   For each q, the computation of T_q can be done in time $O(|\mathcal{G}(\rho^{[]}.\odot)|)$ and thus in $O(|\rho|^2)$.
4. Finally, for each pair of states $q \in \hat{Q}$ and $p \in \hat{Q} \setminus \{q_0\}$, the tree T_q has to be traversed backward from the leaf p, if it is part of T_q, to the root q to obtain the output word w for transition $\langle p, w\rangle \in \hat{\delta}(q, p^{\#})$ of $\mathcal{A}^{[]}(\rho)$.
   This takes time $O(|\rho|^2)$ for each of the q (of which there are $|\hat{Q}| = O(|\rho|)$ many).

We thus construct $\mathcal{A}^{[]}(\rho)$ in time $O(|\rho|^3)$ in total. □

Example 22 Consider the regular expression $\rho = (a^*.b)^*$ and its bracketing $\rho^{[]} = [_1.([_2.[_3.([_4.a.]_4)^*.]_3.[_5.b.]_5.]_2)^*.]_1$. The Glushkov automaton $\mathcal{G}(\rho.\odot)$, shown in Fig. 8a, is deterministic, so $\rho.\odot$ and also ρ are one-unambiguous. Figure 8b depicts the Glushkov automaton $\mathcal{G}(\rho^{[]}.\odot)$, while the FST $\mathcal{A}^{[]}(\rho)$ from our construction is shown in Fig. 8c. As this FST is deterministic, ρ is also strongly one-unambiguous. Note that the DFA obtained from $\mathcal{A}^{[]}(\rho)$ by ignoring its output is the Glushkov automaton $\mathcal{G}(\rho.\odot)$.

As the following example demonstrates, the size of a FST $\mathcal{A}^{[]}(\rho)$ from our construction can be cubic in the size of ρ, which shows that our algorithm for computing $\mathcal{A}^{[]}(\rho)$ is in a sense optimal.

Example 23 Consider the family $(\rho_n)_{n\geq 1}$ of strongly one-unambiguous regular expressions with n-th member

$$\rho_n = ((((((a_1^*.a_2)^*.a_3)^*.a_4)^* \ldots)^*.a_{n-1})^*.a_n)^*.$$

and n-th member alphabet $\Sigma_n = \{a_1, \ldots, a_n\}$.

Figure 9 shows the labeled parse tree of ρ_4 for $n = 4$ and Fig. 10 shows the structure of FST $\mathcal{A}^{[]}(\rho_4)$ and some labelings of transitions. We observe the following:

- Each state a_i' has $i+1$ outgoing transitions, of which $i-1$ are backward transitions, i.e., the transitions $\delta(a_j', a_i) = (a_i', b)$ where $i < j$. Thus, the overall number of transitions of FST $\mathcal{A}^{[]}(\rho_n)$ is $\Theta(n^2)$.

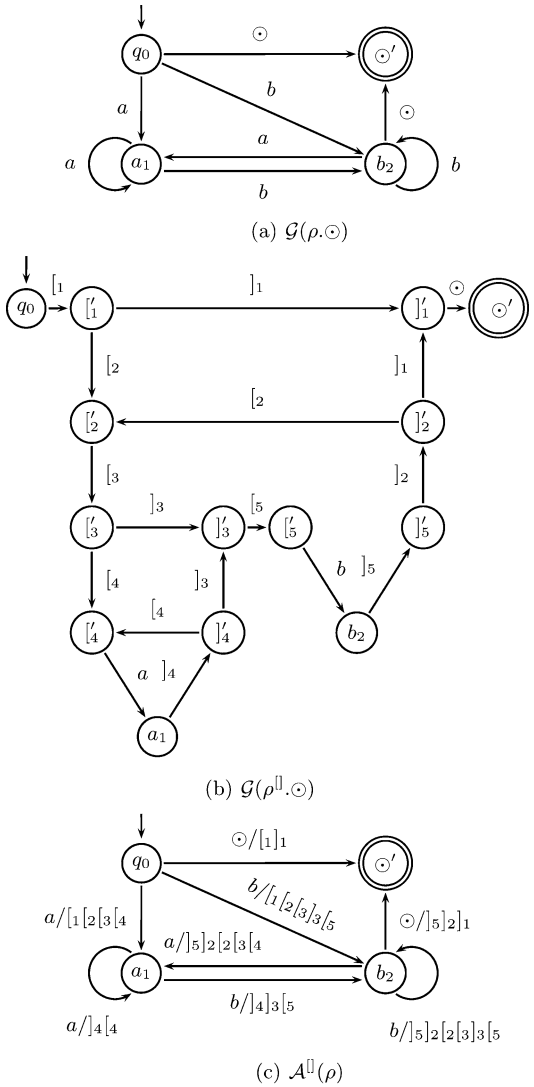


Fig. 8 Stepwise DFT construction of Example 22

– We now study the lengths of the output words of backward transitions. The bracketing of ρ_n assigns each node in the parse tree of ρ_n an identifier. Let l_j be the identifier assigned to node a_j and let p_j be the identifier assigned to the parent node of a_j. Analogously, l_i and p_i are the identifiers of node a_i and its parent node respectively. As the nodes are labeled by a depth-first left-to-right traversal of the parse tree, $p_j < p_i \leq 2n$ and

$$b = \begin{cases}]_{l_j}]_{p_j}[_{p_j}[_{p_j+1}[_{p_j+2} \cdots [_{p_i}[_{p_i+1}]_{p_i+1}[_{l_i} & i \neq 1 \\ ]_{l_j}]_{p_j}[_{p_j}[_{p_j+1}[_{p_j+2} \cdots [_{2n} & i = 1 \end{cases}$$

Thus, $|b|$ is $\Theta(n)$.

Consequently, the size of $\mathcal{A}^{[]}(\rho_n)$ is $\Theta(n^3)$.

4 XML stream attribute grammars

We are now in the position to define our main grammar formalism, the *XML Stream Attribute Grammars* (XSAGs).

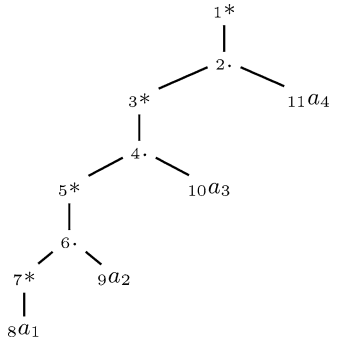


Fig. 9 Parse tree for $\rho_4 = (((a_1^*.a_2)^*.a_3)^*.a_4)^*$

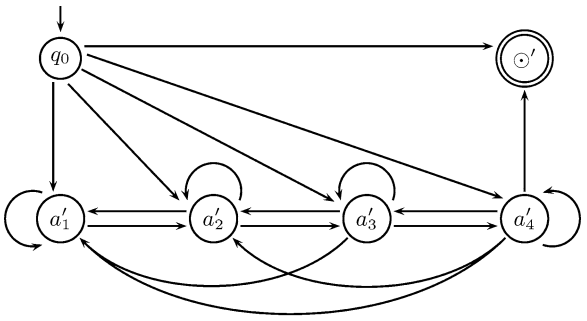


Fig. 10 FST from Example 23

4.1 XSAGs in the abstract

Definition 7 (Syntax) Let $Att = \{a_1, \ldots, a_k\}$ be a set of attributes and Dom be a finite set of *domain values*.[7] Let *string* be a set of fixed character strings.

Let $F_{\$[}$ denote the class of partial functions

$$f_{\$[} : Dom^k \to Dom^k \times string$$

called *first-visit attribution functions*, and let $F_{\$]}$ denote the class of partial functions

$$f_{\$]} : Dom^{2k} \to Dom^k \times string$$

called *second-visit attribution functions*. (We will introduce a language for implementing these partial functions in Sect. 4.3).

A *basic XSAG (bXSAG)* is an attributed extended regular tree grammar $G = (Nt, T, P, s)$ with nonterminals Nt, grammar start symbol s, terminals $T = Tag \cup Char$, and productions in P where each production is of one of the four forms

$$nt ::= t(\rho) \quad nt ::= \{f_{\$[}\}\ t(\rho)$$

$$nt ::= t(\rho)\ \{f_{\$]}\} \quad nt ::= \{f_{\$[}\}\ t(\rho)\ \{f_{\$]}\}$$

where $nt \in Nt$, $t \in T$, $f_{\$[} \in F_{\$[}$, $f_{\$]} \in F_{\$]}$, and

[7] Alternatively, we may regard attributes as *states*, encoding the current evaluation state.

– if $t \in Tag$ then ρ is either ϵ or a regular expression over alphabet Nt,
– if $t \in Char$ then $\rho = \epsilon$ and $nt \neq s$.

The abstract syntax of an *attributed regular expression* over symbols Σ can be specified by the EBNF

$$aregex ::= (\text{“}\{\text{”}\ F_{\$[}\ \text{“}\}\text{”})?\ \ aregex_0\ \ (\text{“}\{\text{”}\ F_{\$]}\ \text{“}\}\text{”})?$$
$$aregex_0 ::= \Sigma \mid aregex\ \text{“.”}\ aregex \mid aregex\ \text{“}\cup\text{”}\ aregex \mid aregex\ \text{“*”}$$

An *easy XSAG* (yXSAG) is an attributed extended regular tree grammar $G = (Nt, T, P, s)$ with nonterminals Nt, grammar start symbol s, terminals $T = Tag \cup Char$, and productions in P where each production is of one of the four forms

$$nt ::= t(\alpha) \quad nt ::= \{f_{\$[}\}\ t(\alpha)$$
$$nt ::= t(\alpha)\ \{f_{\$]}\} \quad nt ::= \{f_{\$[}\}\ t(\alpha)\ \{f_{\$]}\}$$

where $nt \in Nt$, $t \in T$, $f_{\$[} \in F_{\$[}$, $f_{\$]} \in F_{\$]}$, and

– if $t \in Tag$ then α is either ϵ or an attributed regular expression over symbols Nt such that the following holds: For the regular expression ρ obtained from α by removing the attributions (enclosed in curly braces), $\tau(\rho)$ is *strongly* one-unambiguous, and
– if $t \in Char$ then $\alpha = \epsilon$ and $nt \neq s$.

The only differences between bXSAGs and yXSAGs are that the former use TDLL(1) grammars while the latter use STDLL(1) grammars, and that in yXSAGs, right-hand side regular expressions may be attributed. (In fact, it is precisely the restriction to STDLL(1) grammars which makes it safe to attribute regular expressions in yXSAGs.) Note that there are XSAGs which are both bXSAGs and yXSAGs. At the same time there are bXSAGs which are not yXSAGs and vice versa.

bXSAGs and yXSAGs are (attributed) *extended* regular tree grammars. For such grammars, nodes of the parse tree may have an arbitrary number of children. When dealing with streams, we generally cannot store the attribute values of all these children in memory. We thus have to introduce special restrictions to be able to deal with streams on the one hand and at the same time assure ease of use and expressiveness to cover practical queries on the other.

We define XSAGs as L-attributed grammars, i.e., attribute grammars whose attributes are evaluated in a single depth-first left-to-right traversal of the document tree. Each node v of the parse tree is visited twice (the visits are referred to by \$[and \$]), first from the previous sibling or the parent of v (if v has no previous sibling) and a second time on returning from the rightmost child of v. In the first visit to a node, a first-visit attribution function is evaluated. In the second visit to a node, a second-visit attribution function is evaluated.

We thus assume that each node v in the parse tree is assigned two attribution functions $f^v_{\$[} \in F_{\$[}$ and $f^v_{\$]} \in F_{\$]}$, and we refer to such parse trees as *attributed parse trees*.[8]

The main purpose of the XSAG grammar component is to unambiguously map XML documents to parse trees. Note that for the evaluation of XSAGs, it will not be necessary at any time to maintain entire parse trees in memory. It remains to specify how our attribute grammars are evaluated on parse trees.

(1) Let us first consider the case of bXSAGs. For a given node v, let p be the production that was used to parse it. Then, let

$$f^v_{\$[} := \begin{cases} f^p_{\$[} \ldots & \text{if } p: nt ::= \{f^p_{\$[}\}\ t(\rho)\ \{f^p_{\$]}\} \\ & \text{or } p: nt ::= \{f^p_{\$[}\}\ t(\rho) \\ f^d_{\$[} \ldots & \text{otherwise} \end{cases}$$

and

$$f^v_{\$]} := \begin{cases} f^p_{\$]} \ldots & \text{if } p: nt ::= \{f^p_{\$[}\}\ t(\rho)\ \{f^p_{\$]}\} \\ & \text{or } p: nt ::= \quad t(\rho)\ \{f^p_{\$]}\} \\ f^d_{\$]} \ldots & \text{otherwise} \end{cases}$$

where $f^p_{\$[} \in F_{\$[}$ and $f^p_{\$]} \in F_{\$]}$, $t \in T$, and ρ is either ϵ or a one-unambiguous regular expression over nonterminals, and where further

$$f^d_{\$[} : \langle x_1, \ldots, x_k \rangle \mapsto \langle x_1, \ldots, x_k, \epsilon \rangle \quad \text{and}$$
$$f^d_{\$]} : \langle x_1, \ldots, x_k, x_{k+1}, \ldots, x_{2k} \rangle \mapsto \langle x_{k+1}, \ldots, x_{2k}, \epsilon \rangle.$$

The default semantics introduced by $f^d_{\$[}$ and $f^d_{\$]}$ corresponds to the usual notion of *copy semantics*.

(2) For yXSAGs, all nodes labeled with terminals are assigned just as in the case of bXSAGs. For a given node v which is not labeled with a terminal, let α be the attributed regular (sub)expression that was used to parse v. Then, let

$$f^v_{\$[} := \begin{cases} f^\alpha_{\$[} \ldots & \text{if } \alpha = \{f^\alpha_{\$[}\}\ \alpha'\ \{f^\alpha_{\$]}\} \\ & \text{or } \alpha = \{f^\alpha_{\$[}\}\ \alpha' \\ f^d_{\$[} \ldots & \text{otherwise} \end{cases}$$

and

$$f^v_{\$]} := \begin{cases} f^\alpha_{\$]} \ldots & \text{if } \alpha = \{f^\alpha_{\$[}\}\ \alpha'\ \{f^\alpha_{\$]}\} \\ & \text{or } \alpha = \quad \alpha'\ \{f^\alpha_{\$]}\} \\ f^d_{\$]} \ldots & \text{otherwise} \end{cases}$$

where $f^\alpha_{\$[} \in F_{\$[}$, $f^\alpha_{\$]} \in F_{\$]}$, and α' is an attributed regular expression, and where the defaults $f^d_{\$[}$ and $f^d_{\$]}$ are defined as for bXSAGs.

During the evaluation of yXSAGs, this assignment can be computed incrementally: For each symbol read from the

[8] Note that attributed parse trees differ from the *annotated parse trees* frequently used in literature [1]: An annotated parse tree shows the actual values of attributes at nodes rather than the attribution functions assigned to the nodes.

input stream, the DFTs from Theorem 1 output a sequence of brackets which describes the depth-first left-to-right traversal of the STDLL(1) parse tree. The brackets are then interpreted as identifiers of attribution functions. Thus, upon reading one input symbol, we execute the composition of all attribution functions represented by the brackets as output by such DFTs.

Example 24 For the yXSAG from Example 2 and the parse tree from Fig. 6c, we assign attribution functions $f^{v_5}_{\$[} := \{\texttt{print } \langle\texttt{article}\rangle\}$ and further $f^{v_5}_{\$]} := \{\texttt{print } \langle\texttt{/article}\rangle\}$ to the node with subscript 5. We further set $f^{v_7}_{\$[} := \{\texttt{ECHO}\}$ and $f^{v_7}_{\$]} := f^{d}_{\$]}$. Similarly, $f^{v_{12}}_{\$[} := \{\texttt{print } \langle\texttt{authors}\rangle; \texttt{ ECHO}\}$ and also $f^{v_{12}}_{\$]} := \{\texttt{print } \langle\texttt{/authors}\rangle\}$. For all other nodes w, $f^{w}_{\$[} := f^{d}_{\$[}$ and $f^{w}_{\$]} := f^{d}_{\$]}$.

To provide a clear picture of the evaluation of XSAG attributes, we distinguish between the states of attribute values *before* (using the subscript "in") and *after* (using the subscript "out") the application of an attribution function.

In the following, we will allow for assignments of an expression returning an m-tuple to a term $\langle v_1, \ldots, v_m\rangle$, meaning the component-wise assignment of elements of the m-tuple to $v_1, \ldots, v_m$.

Definition 8 (Semantics) Let $q_\perp \in Dom$ be a special "uninitialized" value. We evaluate an XSAG on a parse tree $\mathcal{P}$ in a depth-first left-to-right traversal of $\mathcal{P}$ in which we compute, for each attribute $a_i \in Att$ and each node v of $\mathcal{P}$, the four assignments $(a_i)^v_{\$[.in}$, $(a_i)^v_{\$[.out}$, $(a_i)^v_{\$].in}$, and $(a_i)^v_{\$].out}$ (inductively) as follows.

$$(a_i)^v_{\$[.in} := \begin{cases} q_\perp & \ldots\ v \text{ is the root node} \\ (a_i)^{v_0}_{\$[.out} & \ldots\ v \text{ is the first child of } v_0 \\ (a_i)^{v_0}_{\$].out} & \ldots\ v \text{ is the right sibling of } v_0 \end{cases}$$

$$(a_i)^v_{\$].in} := \begin{cases} (a_i)^v_{\$[.out} & \ldots\ v \text{ has no children} \\ (a_i)^w_{\$].out} & \ldots\ w \text{ is the rightmost child of } v \end{cases}$$

In the first visit to node v, we compute

$$\langle (a_1)^v_{\$[.out}, \ldots, (a_k)^v_{\$[.out}, \sigma\rangle := f^v_{\$[}\big((a_1)^v_{\$[.in}, \ldots, (a_k)^v_{\$[.in}\big)$$

and write σ to the output. In the second visit to v, we compute

$$\langle (a_1)^v_{\$].out}, \ldots, (a_k)^v_{\$].out}, \sigma\rangle := f^v_{\$]}\big((a_1)^v_{\$[.out}, \ldots, (a_k)^v_{\$[.out}, (a_1)^v_{\$].in}, \ldots, (a_k)^v_{\$].in}\big)$$

and write σ to the output. In case $f^v_{\$[}$ or $f^v_{\$]}$ is undefined on its input, the evaluation terminates and the input is rejected.

Let $L(G)$ be the language accepted by XSAG G. For input $w \in L(G)$, we define $G(w) \in string^*$ to be the output produced by G in accepting w. The *translation* defined by an XSAG G, denoted $T(G)$, is defined as $T(G) = \{(w, o) \mid w \in L(G) \text{ and } o \in G(w)\}$.

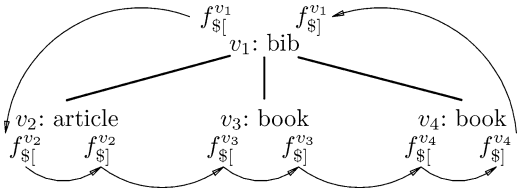


Fig. 11 Attributed bXSAG parse tree and traversal of Example 25

The *result* of the evaluation of an XSAG on an input tree is the *output* (rather than attribute values) it computes, if it accepts its input.

Even though this semantics may seem involved, we believe that its application is natural.

Note that an XSAG may reject an XML document in two ways: A document that does not conform to the grammar component is rejected. Further, the document can also be rejected if an attribution function is not defined for its input.

Example 25 Consider bXSAG G, $Dom = \{q_\perp, q_b, q_a\}$, $Att = \{prev\}$, the productions

$$bib ::= \{f^{bib}_{\$[}\}\ \text{bib}((book \cup article)^*)\ \{f^{bib}_{\$]}\}$$
$$book ::= \{f^{book}_{\$[}\}\ \text{book}(\epsilon)$$
$$article ::= \{f^{article}_{\$[}\}\ \text{article}(\epsilon)$$

and the attribution functions

$$f^{bib}_{\$[} : x \mapsto \langle x, \langle\texttt{bib}\rangle\rangle$$
$$f^{bib}_{\$]} : (x_1, x_2) \mapsto \langle x_2, \langle\texttt{/bib}\rangle\rangle$$
$$f^{article}_{\$[} : x \mapsto \langle q_a, \langle\texttt{article/}\rangle\rangle$$
$$f^{book}_{\$[} : x \mapsto \begin{cases} \langle q_b, \langle\texttt{book/}\rangle\rangle & \ldots\ x = q_a \\ \langle q_b, \epsilon\rangle & \ldots\ \text{otherwise} \end{cases}$$

The grammar requires the input to consist of a dummy bibliography database containing *book* and *article* nodes without children. As output, the XSAG writes a root node labeled "bib", to which it assigns nodes labeled "book" and "article" as children, filtering out books that are not right neighbors of articles.[9]

The TDLL(1) parse tree of the XML document

$$\langle\texttt{bib}\rangle\langle\texttt{article/}\rangle\langle\texttt{book/}\rangle\langle\texttt{book/}\rangle\langle\texttt{/bib}\rangle$$

is shown in Fig. 11. Naturally, we assign $f^{bib}_{\$[}$ to $f^{v_1}_{\$[}$, $f^{bib}_{\$]}$ to $f^{v_1}_{\$]}$, $f^{article}_{\$[}$ to $f^{v_2}_{\$[}$, $f^{book}_{\$[}$ to $f^{v_3}_{\$[}$ and $f^{v_4}_{\$[}$, and have $f^{v_2}_{\$]}$, $f^{v_3}_{\$]}$, $f^{v_4}_{\$]}$ as $f^{d}_{\$]} : (x_1, x_2) \mapsto (x_2, \epsilon)$, i.e. the default copy semantics. (*prev* is initialized with $q_\perp$.)

G is evaluated on the parse tree as shown in Table 1. The five columns have the following meanings. The first column shows the current XML start or end tag being read. The

[9] This is a somewhat contrived example but it illustrates a number of important points related to the evaluation of XSAGs.

Table 1 Run of bXSAG G of Example 25

		Attribute value *prev*		
Input	F	Before	After	Output
⟨bib⟩	$f_{\$[}^{v_1}$	$q_\perp$	$q_\perp$	⟨bib⟩
⟨article⟩	$f_{\$[}^{v_2}$	$q_\perp$	q_a	⟨article/⟩
⟨/article⟩	$f_{\$]}^{v_2}$	(q_a, q_a)	q_a	ϵ
⟨book⟩	$f_{\$[}^{v_3}$	q_a	q_b	⟨book/⟩
⟨/book⟩	$f_{\$]}^{v_3}$	(q_b, q_b)	q_b	ϵ
⟨book⟩	$f_{\$[}^{v_4}$	q_b	q_b	ϵ (!!)
⟨/book⟩	$f_{\$]}^{v_4}$	(q_b, q_b)	q_b	ϵ
⟨/bib⟩	$f_{\$]}^{v_1}$	$(q_\perp, q_b)$	q_b	⟨/bib⟩

second column shows what attribution function is applied in this step. The third and fourth columns show the value of attribute *prev* before and after the application of the attribution function, respectively. For second-visit attribution functions, we show both input values. The rightmost column shows which output is produced and written to the output stream. Clearly, G outputs

$$\langle \texttt{bib} \rangle \langle \texttt{article/} \rangle \langle \texttt{book/} \rangle \langle \texttt{/bib} \rangle$$

and accepts its input.

Example 26 Consider again the XSAG of the previous example. Alternatively, to reject the input[10] if two books arrive in sequence, we define $f_{\$[}^{book}$ as

$$f_{\$[}^{book} : x \mapsto \begin{cases} \langle q_b, \langle \texttt{book/} \rangle \rangle & \ldots\ x = q_a \\ \text{undefined} & \ldots\ x = q_b. \\ \langle q_b, \epsilon \rangle & \ldots\ \text{otherwise} \end{cases}$$

This XSAG rejects the input from Example 25.

4.2 Mixed-content with yXSAGs

Ad hoc, we cannot specify yXSAGs with mixed content models in productions. For for a nonterminal A, the regular expression $\rho = (PCDATA \cup A)^*$ is not strongly one-unambiguous by the Definition of macro *PCDATA* (see Remark 1). Consequently, a grammar with such a production is not STDLL(1) and cannot serve as a yXSAG grammar component. Yet it is possible to specify such attribute grammars and then translate them to valid yXSAGs, as we briefly sketch out below. We refrain from presenting this translation in all its technical details and rather concentrate on the basic idea.

Replacing occurrences of macro *PCDATA* in mixed-content regular expressions with macro *PCDAT* yields an attributed strongly one-unambiguous regular expression. However, we still need to ensure that the attribution functions are properly evaluated.

Assume the attributed regular expression α

$$= \left(\left(\{f_{\$[}^{PCDATA}\} PCDATA \{f_{\$]}^{PCDATA}\}\right) \cup \left(\{f_{\$[}^{A}\} A \{f_{\$]}^{A}\}\right)\right)^*$$

on the right-hand side of a production. If we simply replace *PCDATA* by *PCDAT*, then the attribution functions $f_{\$[}^{PCDATA}$ and $f_{\$]}^{PCDATA}$ are evaluated for every single character symbol matched by this production. Most likely, the user intended that $f_{\$[}^{PCDATA}$ and $f_{\$]}^{PCDATA}$ should only be evaluated on the first and last characters in a contiguous character sequence. In other words, we want to match the longest contiguous string of characters as one *PCDATA* tree region.

We introduce an additional attribute in order to find these character strings. On reading a character symbol, we evaluate $f_{\$[}^{PCDATA}$ when we are at the beginning of a string. We only know for sure that we have reached the end of a string when we read the next non-character symbol. We encode this decision and the evaluation of $f_{\$]}^{PCDATA}$ into the first-visit attribution functions which can be evaluated after we have read a character symbol. Yet by "delaying" the execution of $f_{\$]}^{PCDATA}$, we cannot access the same attribute values. This can be solved by introducing a fixed number of additional attributes (depending on the maximum depth of all parse trees of attributed regular expressions in the XSAG). These additional attributes simulate a bounded stack on which we can store the attributes computed by first-visit attribution function $f_{\$[}^{PCDATA}$, so that the delayed second-visit attribution function $f_{\$]}^{PCDATA}$ may have access to them when it is finally executed.

4.3 Concrete XSAGs

We introduce the simple imperative programming language *CT-Pascal* for the definition of attribution functions. This language is basically a fragment of Pascal, comprising the following constructs which can be executed in *constant time*: (1) if-then-else statements, (2) blocks of multiple commands starting with the keyword "begin" and ending with "end", (3) Boolean formulas – using "and", "or", and "not" – over equality conditions (used in if-statements), (4) assignments, (5) the keyword "reject" for terminating the computation and rejecting the input, and (6) "print" statements taking a constant string as arguments.

An assignment is a statement of the form $x := y$, where x is an l-value and y is an r-value.[11] An equality condition is a statement of the form $x = y$, where x and y are r-values.

Attribution functions are specified using *copy semantics*, i.e., if an attribute is not explicitly assigned then the XSAG evaluation assumes the default copy semantics. More precisely, the semantics of such a program is defined using the

[10] This could be alternatively achieved by modifying the grammar rather than the attributions as done in this example, but the goal here is to illustrate the use of partially undefined attribution functions.

[11] We call constructs of our language which may appear on the right-hand side of an assignment *r-values* while we call those which may appear on the left-hand side of an assignment *l-values*.

usual notion of an environment as a function $\mathcal{E} : Att \to Dom$ that maps each attribute name to a domain value. Let $Att = \{a_1, \ldots, a_k\}$ be a set of attributes with domain Dom.

Consider a program defining a first-visit attribution function $f_{\$[} : Dom^k \to Dom^k \times string$ and its execution $f_{\$[}(\$[.\vec{x})$ on attribute values

$$\$[.\vec{x} = \langle \$[.x_1, \ldots, \$[.x_k \rangle \in Dom^k.$$

At the start of the execution of attribution function $f_{\$[}(\$[.\vec{x})$, $\mathcal{E}(a_i) = \$[.x_i$ for $1 \le i \le k$. During the execution of $f_{\$[}(\$[.\vec{x})$, the attributes a_i may be read as well as written (by assignment ":=").

$f_{\$[}(\$[.\vec{x})$ evaluates to $\langle \mathcal{E}^\omega(a_1), \ldots, \mathcal{E}^\omega(a_k), o \rangle$, where $\mathcal{E}^\omega$ is the environment at the end of the execution and o is the concatenation of the symbols printed. Thus, for (partial) functions in $F_{\$[}$, the l-values consist of the set $\{\$[.a \mid a \in Att\}$ and the r-values consist of the set $\{\$[.a \mid a \in Att\} \cup Dom$.

Consider a program defining a second-visit attribution function $f_{\$]} : Dom^{2k} \to Dom^k \times string$ and its execution $f_{\$]}(\$[.\vec{x}, \$].\vec{x})$ on attribute values $\$[.\vec{x}, \$].\vec{x} \in Dom^k$. At the start of the execution of attribution function $f_{\$]}(\$[.\vec{x}, \$].\vec{x})$, $\langle \$[.\vec{x}, \$].\vec{x} \rangle$ is copied into the environment. However, the attributes of $\$[.\vec{x}$ are read-only and must therefore not appear on the left-hand sides of assignments. For (partial) functions in $F_{\$]}$, the l-values are $\{\$].a \mid a \in Att\}$ and the r-values are $\{\$[.a \mid a \in Att\} \cup \{\$].a \mid a \in Att\} \cup Dom$.

Such a program defines functions in $F_{\$[}$ resp. $F_{\$]}$ in the obvious way, with the notable fact that the functions are assumed undefined for inputs for which the "`reject`" statement is called.

Example 27 Using our Pascal-like syntax, we define the attribution functions of Example 25 as

$$\begin{aligned}
f^{bib}_{\$[} &= \{\texttt{print } \langle \texttt{bib} \rangle\} \\
f^{bib}_{\$]} &= \{\texttt{print } \langle \texttt{/bib} \rangle\} \\
f^{article}_{\$[} &= \{\texttt{print } \langle \texttt{article/} \rangle;\ \$[.\texttt{prev} := \texttt{q}_\texttt{a}\} \\
f^{book}_{\$[} &= \{\texttt{if } \$[.\texttt{prev} = \texttt{q}_\texttt{a} \texttt{ then print } \langle \texttt{book/} \rangle; \\
&\qquad \$[.\texttt{prev} := \texttt{q}_\texttt{b}\}
\end{aligned}$$

Thus, we can write the bXSAG of Example 25 as

$$\begin{aligned}
bib &::= \{\texttt{print } \langle \texttt{bib} \rangle\} \\
&\quad \text{bib}((book \cup article)^*)\ \{\texttt{print } \langle \texttt{/bib} \rangle\} \\
book &::= \{\texttt{if } \$[.\texttt{prev} = \texttt{q}_\texttt{a} \texttt{ then print } \langle \texttt{book/} \rangle; \\
&\quad \$[.\texttt{prev} := \texttt{q}_\texttt{b}\}\ \text{book}(\epsilon) \\
article &::= \{\texttt{print } \langle \texttt{article/} \rangle;\ \$[.\texttt{prev} := \texttt{q}_\texttt{a}\} \\
&\quad \text{article}(\epsilon)
\end{aligned}$$

To modify the XSAG to reject its input if two books arrive in sequence on the stream, as in Example 26, we define $f^{book}_{\$[}$ as

$$\begin{aligned}
&\{\texttt{if } \$[.\texttt{prev} = \texttt{q}_\texttt{a} \texttt{ then} \\
&\quad \texttt{begin print } \langle \texttt{book/} \rangle;\ \$[.\texttt{prev} := \texttt{q}_\texttt{b} \texttt{ end} \\
&\texttt{else if } \$[.\texttt{prev} = \texttt{q}_\texttt{b} \\
&\quad \texttt{then reject} \\
&\quad \texttt{else } \$[.\texttt{prev} := \texttt{q}_\texttt{b}\}
\end{aligned}$$

4.4 Built-in macros

We introduce three built-in macros for the convenient definition of XSAGs, namely (1) `ECHO`, (2) `ECHO_OFF`, and (3) `MATCH_CHILDREN`. These are redundant with the formalism presented so far but they allow us to define queries in a more concise way.

Output macros echo and echo off

Let v be a node in an attributed parse tree. If macro `ECHO` is used in the first-visit attribution function assigned to v, then the subtree rooted at v will be copied to the output. Correspondingly, macro `ECHO_OFF` can be used to override `ECHO` and thus to suppress the output of certain subtrees.

While Example 1 already illustrates the use of `ECHO`, the example below combines macros `ECHO` and `ECHO_OFF`.

Example 28 The yXSAG wit grammar start symbol *bib* and productions

$$\begin{aligned}
bib &::= \{\texttt{ECHO}\}\ \text{bib}\big(book^*\big) \\
book &::= \text{book}\big(title.author. \\
&\qquad (\{\texttt{ECHO_OFF}\}(author^*)).year\big) \\
title &::= \text{title}(PCDATA) \\
author &::= \text{author}(PCDATA) \\
year &::= \text{year}(PCDATA)
\end{aligned}$$

outputs each book with its title, first author, and year, while any further authors are dropped.

We now describe macro expansion for output macros `ECHO` and `ECHO_OFF`. Let $G = (Nt, T, P, s)$ be an XSAG. We assume that all nodes in the attributed parse tree are assigned two attribution functions, with default attribution functions where this has not been explicitly defined. We expand echo macros in the first-visit attribution functions of G as follows: We define two attributes *echo* and *echo_old* with domain $\{q_\perp, true, false\}$, and initialize *echo* with *false*. Setting attribute *echo* determines whether nodes in the parse tree are to be copied to the output. Attribute *echo_old* is used to properly reset *echo* in second-visit attribution functions.

1. When processing the start tag of the XML root node, *echo* is initialized with *false*. To this end, we add the command "`if` $\$[.\texttt{echo} = \texttt{q}_\perp$ `then` $\$[.\texttt{echo} := \texttt{false}$" as a prefix to each first-visit function assigned to a start production.
2. In every first-visit attribution function, we replace occurrences of `ECHO` by "$\$[.\texttt{echo} := \texttt{true}$," and occurrences of `ECHO_OFF` by "$\$[.\texttt{echo} := \texttt{false}$."

3. In the attribution functions $f_{\$[}$ and $f_{\$]}$ of every production $nt ::= \{f_{\$[}\}\ t(\rho)\ \{f_{\$]}\}$, we generate output depending on the value of *echo*:
   – If $t \in Tag$, we append

     `if $[.echo = true then print ⟨t⟩`

     to $f_{\$[}$ and we append the following to $f_{\$]}$:

     `if $[.echo = true then print ⟨/t⟩`

   – likewise, if $t \in Char$, we append the command "`if $[.echo = true then print t`" to $f_{\$[}$.
4. We add a prefix "`$[.echo_old := $[.echo`" to every first-visit attribution function $f^v_{\$[}$ which uses an echo macro. This statement stores the value of attribute *echo* just before the remaining commands are executed. Correspondingly, we add a postfix to the second-visit attribution function $f^v_{\$]}$ assigned to the same node v: "`$].echo := $[.echo_old`" resets *echo* to the value it had before the subtree rooted at node v was processed.

Example 29 Figure 12 shows the STDLL(1) parse tree of the input document

```
⟨bib⟩
 ⟨book⟩
  ⟨title/⟩⟨author/⟩⟨author/⟩⟨author/⟩⟨year/⟩
 ⟨/book⟩
⟨/bib⟩
```

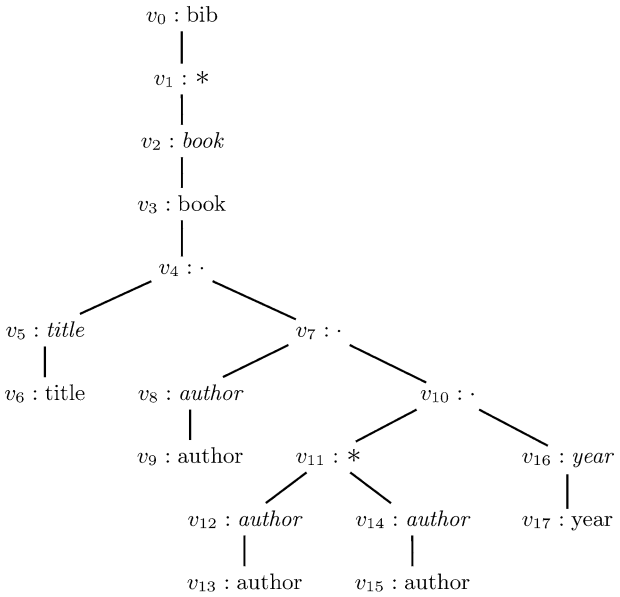
for the yXSAG of Example 28. We refer to the nodes by their identifiers v_i with $0 \le i \le 17$. The modified attribution functions $f^{v_i}_{\$[}$ and $f^{v_i}_{\$]}$, assigned to every node v_i in the attributed parse tree, are also shown in Fig. 12.

The attribution functions are evaluated along the depth-first left-to-right traversal of the attributed parse tree: Initially, *echo* is set to *false*. The declaration of `ECHO` in the first-visit attribution function at node v_0 changes *echo* to *true*. In consequence, the attribution functions for nodes v_0, v_3, v_6, and v_9, (nodes with labels in $Tag \cup Char$), generate output. Attribute *echo* is set to *false* in the first-visit attribution function assigned to node v_{11}, thus suppressing the output of the subtree rooted at this node. As attribute *echo* is reset to *true* by $f^{v_{11}}_{\$]}$, the tags of node v_{17} are output correctly.

Macro expansion may introduce redundant code, yet we can eliminate many redundant statements by a simple program analysis which takes into account the XSAG semantics. For instance, we could simplify the attribution function $f^{v_0}_{\$[}$ to "{`$[.echo := true; print ⟨bib⟩`}" and change $f^{v_0}_{\$]}$ to "{`print ⟨/bib⟩`}".

Conditional macro match

Conditional output is a typical task in query processing. We introduce conditional macro `MATCH_CHILDREN` which



$f^{v_0}_{\$[} = \{$ `if $[.echo = q⊥ then $[.echo := false;`
`$[.echo_old := $[.echo; $[.echo := true;`
`if $[.echo = true then print ⟨bib⟩` $\}$

$f^{v_0}_{\$]} = \{$ `if $[.echo = true then print ⟨/bib⟩;`
`$].echo := $[.echo_old` $\}$

$f^{v_3}_{\$[} = \{$ `if $[.echo = true then print ⟨book⟩` $\}$

$f^{v_3}_{\$]} = \{$ `if $[.echo = true then print ⟨/book⟩` $\}$

$f^{v_6}_{\$[} = \{$ `if $[.echo = true then print ⟨title⟩` $\}$

$f^{v_6}_{\$]} = \{$ `if $[.echo = true then print ⟨/title⟩` $\}$

$f^{v_9}_{\$[} = \{$ `if $[.echo = true then print ⟨author⟩` $\}$

$f^{v_9}_{\$]} = \{$ `if $[.echo = true then print ⟨/author⟩` $\}$

$f^{v_{11}}_{\$[} = \{$ `$[.echo_old := $[.echo; $[.echo := false` $\}$

$f^{v_{11}}_{\$]} = \{$ `$].echo := $[.echo_old` $\}$

$f^{v_{13}}_{\$[} = f^{v_{15}}_{\$[} = f^{v_9}_{\$[}$

$f^{v_{13}}_{\$]} = f^{v_{15}}_{\$]} = f^{v_9}_{\$]}$

$f^{v_{17}}_{\$[} = \{$ `if $[.echo = true then print ⟨year⟩` $\}$

$f^{v_{17}}_{\$]} = \{$ `if $[.echo = true then print ⟨/year⟩` $\}$

Fig. 12 Parse tree and attribution functions of Example 29

matches strings of characters from the input stream by regular expressions. Let v be a node in the attributed parse tree with attribution functions $f^v_{\$[}$ and $f^v_{\$]}$ assigned to it, and let s be the string obtained from concatenating the character data encountered in the left-to-right traversal of the children of v. Let ρ be a regular expression over terminals and let c be a Boolean-valued attribute. Assume that macro `MATCH_CHILDREN`(ρ, c) is declared in first-visit attribution function $f^v_{\$[}$. If $s \in L(\rho)$, then c is set to *true*, otherwise c is set to *false*. Naturally, the result becomes first available in attribution function $f^v_{\$]}$, as this is the earliest moment

where we have seen the complete string s. Note that within $f^v_{\$]}$ we can access the final value of c by $\$].c$, whereas $\$[.c$ will still be set to *false*.

`MATCH_CHILDREN` can be easily implemented by compiling ρ into a DFA which is then simulated by the attribution functions: We represent the current state of the DFA by an XSAG attribute and realize the state transitions with corresponding CT-Pascal statements.

Example 30 We modify Example 3 such that the new yXSAG production

book ::= book(({`MATCH_CHILDREN(2003, $[.c)`} *year*).
({`if $[.c = true then begin print ⟨book⟩; ECHO end`}
(*title.author.author**)
{`if $[.c = true then print ⟨year⟩2003⟨/year⟩⟨/book⟩`}))

selects those books whose child *year* has string value "2003"; moreover, the year is output as the rightmost child of book, rather than as the leftmost as required for the input.

4.5 bXSAGs versus yXSAGs

As we show in the next section, bXSAGs and yXSAGs have the same expressive power. However, yXSAGs are generally more convenient to use. In particular, it is often necessary to introduce more attributes and more complicated attribution functions to encode a given query as a bXSAG as when encoding it as a yXSAG.

Example 31 Consider the following STDLL(1) grammar

bib ::= bib(*article**)
article ::= article((*title.author.author**) ∪
(*year.title.author.author**.*pub*))
title ::= title(*PCDATA*)
author ::= author(*PCDATA*)
year ::= year(*PCDATA*)
pub ::= publisher(*PCDATA*)

where article entries appear either in a short version with a title and at least one author, or in a long version which also contains a year and a publisher. By changing the *bib* and *article* productions to

bib ::= {`print ⟨bib⟩`} bib(*article**) {`print ⟨/bib⟩`}
article ::= article(({`print ⟨article_short⟩; ECHO`}
(*title.author.author**)
{`print ⟨/article_short⟩`}) ∪
({`print ⟨article_long⟩; ECHO`}
(*year.title.author.author**.*pub*)
{`print ⟨/article_long⟩`})))

we obtain a yXSAG where short articles are relabeled as "article_short", and long articles are relabeled as "article_long". Note that if both short and long articles had the first child node *year* then we could not encode this transformation using XSAGs.

The following bXSAG is equivalent to the yXSAG above. It uses an attribute *state* $\in \{q_\perp, q_{init}, q_{short}, q_{long}\}$ (where $q_\perp$ is the "uninitialized" value) to distinguish the titles of short articles from those of long articles.

bib ::= {`print ⟨bib⟩`} bib(*article**) {`print ⟨/bib⟩`}
article ::= {`$[.state := q_init`}
article((*title.author.author**) ∪
(*year.title.author.author**.*pub*))
{`if $].state = q_short then print ⟨/article_short⟩ else if $].state = q_long then print ⟨/article_long⟩`}
title ::= {`if $[.state = q_init then begin $[.state := q_short; print ⟨article_short⟩ end; ECHO`} title(*PCDATA*)
author ::= {`ECHO`} author(*PCDATA*)
year ::= {`$[.state := q_long; print ⟨article_long⟩; ECHO`} year(*PCDATA*)
pub ::= {`ECHO`} publisher(*PCDATA*)

While there are other ways of encoding our query using a bXSAG, it does not seem possible to represent the query as a bXSAG without using attributes (other than those required to implement `ECHO`).

It is easy to verify that bXSAGs equivalent to the yXSAGs of Examples 2, 3, and 30 are also much more complicated.

5 Expressive power of XSAGs

In this section, we explore the expressiveness of XSAGs. First, we define a class of deterministic pushdown transducers tailored towards XML stream processing, called *XML-DPDT*s. We then present our main result, namely that XSAGs are precisely as expressive as *XML-DPDT*s. In the proofs to our expressiveness theorems, we also specify the corresponding translation algorithms.

5.1 Deterministic pushdown transducers for XML streams

We first introduce deterministic pushdown transducers (*DPDT*s) as deterministic pushdown automata with output

which accept by empty stack. As with pushdown automata [2], the *DPDT*s accepting by empty stack are equivalent to the *DPDT*s accepting by final state.

Definition 9 (*DPDT*) A *deterministic pushdown transducer* is a tuple

$$\mathcal{T} = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0)$$

where Q is a finite set of states, Σ, Γ, and Δ are the finite alphabets for input tape, stack, and output tape respectively, δ is the partial transition function

$$\delta \ : \ Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^* \times \Delta^*$$

q_0 denotes the initial state, and Z_0 the initial stack symbol. For each $q \in Q$ and $X \in \Gamma$ such that $\delta(q, \epsilon, X)$ is defined, $\delta(q, a, X)$ is undefined for all $a \in \Sigma$. A transition $\delta(q, \epsilon, X)$ is called an ϵ-transition. A DPDT without ϵ-transitions is called ϵ-free.

We define a run of $\mathcal{T}$ by means of *instantaneous descriptions* (IDs). An ID describes the configuration of a DPDT at a given instant. It is defined as a quadruple

$$(q, w, \alpha, o) \ \in \ Q \times \Sigma^* \times \Gamma^* \times \Delta^*,$$

where q is a state, w is the remaining input, α a string of stack symbols denoting the current stack, and o the output generated so far. We make a transition

$$(q, aw, X\alpha, o) \vdash (q', w, \gamma\alpha, o\sigma)$$

if $\delta(q, a, X) = (q', \gamma, \sigma)$, where $a \in \Sigma \cup \{\epsilon\}$, $X \in \Gamma$, $\alpha \in \Gamma^*$, $q' \in Q$, and $\sigma \in \Delta^*$. Here, $\gamma \in \Gamma^*$ is the string of stack symbols which replace X on top of the stack. For $\gamma = \epsilon$, the stack is popped, whereas for $\gamma = X$, the stack remains unchanged. If $\gamma = YX$, then Y is pushed on top of X.

Let $\vdash^*$ be the reflexive and transitive closure of $\vdash$. $\mathcal{T}$ accepts an input word $w \in \Sigma^*$ by empty stack if

$$(q_0, w, Z_0, \epsilon) \vdash^* (q, \epsilon, \epsilon, o)$$

for $q \in Q$ and $o \in \Delta^*$. We say o is the output for input w.

The language accepted by a DPDT $\mathcal{T}$, denoted $L(\mathcal{T})$, is the set of strings accepted by $\mathcal{T}$. The translation defined by $\mathcal{T}$, denoted by $T(\mathcal{T})$, is defined as

$$\{(w, o) \mid (q_0, w, Z_0, \epsilon) \vdash^* (q, \epsilon, \epsilon, o) \text{ for } q \in Q, o \in \Delta^*\}.$$

We call two DPDTs equivalent if they define the same translation.

Throughout this paper, for a set S, we use $S^{\leq 2}$ as a shortcut for $\{\epsilon\} \cup S \cup S \times S$.

Definition 10 (*XML-DPDT*) Let the input alphabet

$$\Sigma = \{\langle t\rangle \mid t \in Tag\} \cup \{\langle /t\rangle \mid t \in Tag\} \cup Char$$

consist of matching XML start and end tags and characters.

An *XML-DPDT* is a *DPDT*

$$\mathcal{T} = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0)$$

where $\Gamma = \{Z_0\} \cup Tag \times \Gamma'$ (i.e. the stack alphabet consists of stack start symbol Z_0 and a pair of a tag and some symbol from a set Γ'), and for which the transition function δ is restricted as follows:

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^{\leq 2} \times \Delta^*$$

1. In the very first transition the initial stack symbol is replaced; in particular, we require

   $$\delta(q_0, \langle t\rangle, Z_0) = (p, (t, Y), \sigma)$$

   for $\langle t\rangle \in \Sigma$, $p \in Q$, $(t, Y) \in \Gamma$, and $\sigma \in \Delta^*$.
2. For all other configurations of $q \in Q$ and $X \in \Gamma$, a symbol is only pushed on the stack when an XML start tag is read from the input stream. We require

   $$\delta\big(q, \langle t\rangle, X\big) = (p, (t, Y)X, \sigma)$$

   for $\langle t\rangle \in \Sigma$, $p \in Q$, $(t, Y) \in \Gamma$, and $\sigma \in \Delta^*$.
3. A symbol is only popped from the stack when a matching XML end tag is encountered in the input stream, so

   $$\delta(q, \langle /t\rangle, (t, Y)) = (p, \epsilon, \sigma)$$

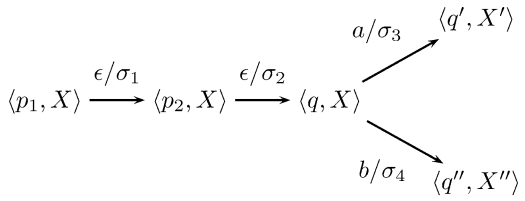
   for $p, q \in Q$, $\langle /t\rangle \in \Sigma$, $(t, Y) \in \Gamma$, and $\sigma \in \Delta^*$.

The conditions required in the *XML-DPDT* definition are only natural in the context of XML stream processing: The size of the stack is bounded by the maximum depth of the incoming document tree. Due to the restriction on the first transition, the input has to start with the root element of the XML document being read. Items are only pushed on the stack for XML start tags and are only popped from the stack for matching end tags. We store the tag of the current production on the stack in order to correctly match start and end tags. Because of the acceptance by empty stack, only well-formed XML documents are accepted.

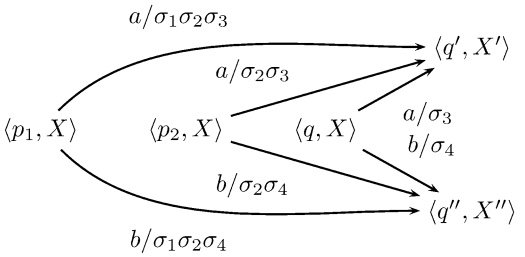
All transitions not related to reading an XML start or end tag, i.e., transitions on character symbols and ϵ-transitions, leave the stack unchanged. The latter is a prerequisite for the elimination of ϵ-transitions from *XML-DPDT*s which is not possible for general DPDTs.

Note that in work subsequent to ours, [3] introduced the so-called visibly pushdown automata and showed that the languages accepted by these automata enjoy nice closure properties. Visibly pushdown languages have recently been used in the context of XML, see [31]. Even though visibly pushdown automata are closely related to *XML-DPDT*s without output, they differ in the treatment of ϵ-transitions and the definition of the acceptance condition.

Lemma 1 *For each XML-DPDT with ϵ-transitions there is an equivalent ϵ-free XML-DPDT.*



(a) Transitions before ϵ-elimination.



(b) Transitions after ϵ-elimination.

Fig. 13 ϵ-elimination for *XML-DPDT*s.

Proof Idea. Let $\mathcal{T}$ be an *XML-DPDT* an let δ be its transition function. We construct an equivalent ϵ-free *XML-DPDT* by computing the transitive closure of ϵ-transitions. Our strategy is based on the following observations:

- By definition, all ϵ-transitions in *XML-DPDT*s are of the form $\delta(q, \epsilon, X) = (q', X, \sigma)$ for states q, q', stack symbol X, and output σ. As such, ϵ-transitions leave the stack unchanged.
- Each sequence of ϵ-transitions in accepting an XML input document is followed by a transition on an input symbol, e.g., the last transition in accepting an input document processes the XML end tag of the document root node.
- Let G be the digraph with nodes in $Q \times \Gamma^{\leq 2}$ and edges $\{\langle (q, X), (q', X')\rangle \mid \delta(q, a, X) = (q', X', \sigma)\}$. An edge from node (q, X) to (q', X') which has been introduced for a transition $\delta(q, a, X) = (q', X', \sigma)$ is labeled "a/σ." Figure 13a shows some transitions in graph representation.
  If a node $\langle q, X\rangle$ is reachable from another node via a path of ϵ-transitions, then this path is unique: As $\mathcal{T}$ is deterministic, a node with an outgoing ϵ-transition cannot have any other outgoing transitions.

Thus, for each finite sequence of ϵ-transitions and the following transition on an input symbol, we define a single transition which produces the collected output. Figure 13b shows the new transitions after the elimination of ϵ-transitions from Fig. 13a.

Proof Let $\mathcal{T}$ with $\mathcal{T} = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0)$ be an *XML-DPDT* with ϵ-transitions. Then we construct an *XML-DPDT* $\mathcal{T}'$ which differs from $\mathcal{T}$ only in its transition function δ':

- For $\langle t\rangle \in \Sigma$, $\delta'(q_0, \langle t\rangle, Z_0) := \delta(q_0, \langle t\rangle, Z_0)$.

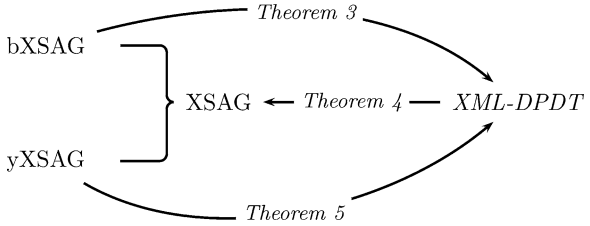


Fig. 14 Equivalence of XSAGs and *XML-DPDT*s

- For all other non-ϵ-transitions on an input symbol $t \in \Sigma$, $\delta(q, t, X) = (q', Y, \sigma)$ with $q, q' \in Q$, $X \in \Gamma$, $Y \in \Gamma^{\leq 2}$, and $\sigma \in \Delta^*$, we consider all states p such that $(p, \epsilon, X, \epsilon) \vdash^* (q, \epsilon, X, o)$ with $o \in \Delta^*$. Consequently, we define $\delta'(p, t, X) := (q', Y, o\sigma)$.

The resulting *XML-DPDT* $\mathcal{T}'$ has no ϵ-transitions. Its determinism follows from the determinism of $\mathcal{T}$: All nodes for which transitions on input symbols are defined keep their outgoing edges (and no additional edges are added). Nodes for which an ϵ-transition is defined on δ are assigned the outgoing edges of the next node reachable via ϵ-transitions which has outgoing edges for reading input symbols.

It can be shown by induction over the input words that $\mathcal{T}$ and $\mathcal{T}'$ are equivalent. □

5.2 Equivalence of XSAGs and *XML-DPDT*s

We call an XSAG G and an *XML-DPDT* $\mathcal{T}$ equivalent if they define the same translation. To show that XSAGs and *XML-DPDT*s are equivalent, we will introduce three theorems, where Fig. 14 shows how these theorems relate to each other. Theorem 3 states that every bXSAG can be translated into an equivalent *XML-DPDT*. In proving Theorem 5, we will show that the same holds for yXSAGs. As there are bXSAGs which are no yXSAGs and vice versa, these separate theorems are not redundant. Finally, given an *XML-DPDT* we construct an XSAG which is both a bXSAG and a yXSAG in the proof of Theorem 4.

Theorem 3 *For each basic XSAG there is an equivalent XML-DPDT.*

Proof Idea. We construct an *XML-DPDT* $\mathcal{T}$ to simulate a bXSAG G. $\mathcal{T}$ performs two tasks while processing the input stream: (1) $\mathcal{T}$ validates the input stream and (2) $\mathcal{T}$ also evaluates the attribution functions.

To validate the input, we construct DFAs which recognize the regular expressions on the right-hand sides of the productions of G. The DFA transitions are then encoded in the transitions of $\mathcal{T}$ and we use the *XML-DPDT* stack to switch between states of DFAs, or rather, the corresponding productions.

Further, we realize the evaluation of attribution functions within the transition function of $\mathcal{T}$. If an attribution function rejects the input stream for a certain combination of attribute values, then we do not define a corresponding transition of $\mathcal{T}$.

As a consequence of (1) and (2), input rejected by G will not be accepted by $\mathcal{T}$ either.

Proof Let Dom be the finite set of domain values for k attributes $Att = \{a_1, \ldots, a_k\}$ and let $q_\perp \in Dom$ be an initial value. We will use vector notation to denote k-tuples of attribute values, so $\vec{a}$ denotes $\langle a_1, \ldots, a_k\rangle \in Dom^k$. Let $F_{\$[}$ and $F_{\$]}$ be sets of first- and second-visit attribution functions.

Let G be a bXSAG $G = (Nt, T, P, s)$ with nonterminals Nt, terminals $T = Tag \cup Char$, productions $P = \{p_1, \ldots, p_n\}$, and grammar start symbol s. We assume that the TDLL(1) grammar component of G is reduced. Further, we assume that all productions $p_i \in P$ are of the form

$$p_i : \quad nt^{p_i} ::= \left\{f^{p_i}_{\$[}\right\} t^{p_i}(\rho^{p_i}) \left\{f^{p_i}_{\$]}\right\}$$

with nonterminal nt^{p_i}, terminal t^{p_i}, ρ^{p_i} being either ϵ or a regular expression over nonterminals such that $\tau(\rho^{p_i})$ is one-unambiguous, and attribution functions $f^{p_i}_{\$[} \in F_{\$[}$ and $f^{p_i}_{\$]} \in F_{\$]}$. If $f^{p_i}_{\$[}$ or $f^{p_i}_{\$]}$ have not been explicitly declared, we assume the default attribution functions $f^{d}_{\$[}$ or $f^{d}_{\$]}$ respectively.

Let $\mathcal{A}^{p_i} = (Q^{p_i}, Nt, \delta^{p_i}, q_0^{p_i}, F^{p_i})$ be a DFA recognizing $L(\rho^{p_i})$. $\mathcal{A}^{p_i}$ has state set Q^{p_i}, input alphabet Nt, transition function $\delta^{p_i} : Q^{p_i} \times Nt \to Q^{p_i}$, initial state $q_0^{p_i}$, and F^{p_i} as the set of final states.

W.l.o.g., we assume that the state sets of all DFAs are pairwise disjoint and define the set $Q_{DFA} = \bigcup_i Q^{p_i}$ as the union of all such DFA state sets. Further, we assume the existence of a special state $q_{root} \notin Q_{DFA}$.

We construct an *XML-DPDT*

$$\mathcal{T} = (Q^{\mathcal{T}}, \Sigma^{\mathcal{T}}, \Gamma^{\mathcal{T}}, \Delta^{\mathcal{T}}, \delta^{\mathcal{T}}, q_0^{\mathcal{T}}, Z_0^{\mathcal{T}}),$$

as follows:

1. The state set is $Q^{\mathcal{T}} = Dom^k \times (Q_{DFA} \cup \{q_{root}\})$. A state $(a_1, \ldots, a_k, q)$ is a $(k+1)$-tuple consisting of the current attribute values $a_1, \ldots, a_k$ and state marker q. The state marker either corresponds to the state of the DFA related to the current production, or it is set to q_{root} which denotes the start or end of the input.
2. The initial state is $q_0^{\mathcal{T}} = (q_\perp, \ldots, q_\perp, q_{root})$, where all attributes are initialized to $q_\perp$ and the state marker is set to q_{root}.
3. The input alphabet $\Sigma^{\mathcal{T}}$ is finite and consists of XML start and end tags and character symbols, i.e.,

   $$\Sigma^{\mathcal{T}} = \{\langle t\rangle \mid t \in Tag\} \cup \{\langle/t\rangle \mid t \in Tag\} \cup Char.$$

4. The stack alphabet $\Gamma^{\mathcal{T}}$ consists of symbols in $\{\$\} \cup Tag \times Q^{\mathcal{T}}$. That is, the stack is either empty or holds a symbol on top of the stack. This can be the initial stack symbol $Z_0^{\mathcal{T}} = \$$ or a pair consisting of a tag and a state.[12]

[12] For the sake of syntactic brevity, we will write stack symbols in $Tag \times Q^{\mathcal{T}} = Tag \times (Dom^k \times (Q_{DFA} \cup \{q_{root}\}))$ as if they were tuples in $Tag \times Dom^k \times (Q_{DFA} \cup \{q_{root}\})$.

5. We obtain output alphabet $\Delta^{\mathcal{T}}$ as follows: $\sigma \in \Delta^{\mathcal{T}}$ iff $f_{\$[} \in F_{\$[}$ is an attribution function in G and there exist $\vec{a}, \vec{b} \in Dom^k$ such that $f_{\$[}(\vec{a}) = \langle \vec{b}, \sigma\rangle$, or, $f_{\$]} \in F_{\$]}$ is an attribution function in G and there exist $\vec{a} \in Dom^{2k}$ and $\vec{b} \in Dom^k$ s.t. $f_{\$]}(\vec{a}) = \langle \vec{b}, \sigma\rangle$.
6. We next define the transition function $\delta^{\mathcal{T}}$ of $\mathcal{T}$,

   $$\delta^{\mathcal{T}} : Q^{\mathcal{T}} \times \Sigma^{\mathcal{T}} \times \Gamma^{\mathcal{T}} \to Q^{\mathcal{T}} \times (\Gamma^{\mathcal{T}})^{\leq 2} \times (\Delta^{\mathcal{T}})^*.$$

   First, we define the initial transitions. For each production p_i with $nt^{p_i} = s$, i.e., a production with grammar start symbol s on the left-hand side, and its corresponding DFA $\mathcal{A}^{p_i}$, we define a transition

   $$\delta^{\mathcal{T}}\left(q_0^{\mathcal{T}}, \langle t^{p_i}\rangle, Z_0^{\mathcal{T}}\right) = \delta^{\mathcal{T}}((q_\perp, \ldots, q_\perp, q_{root}), \langle t^{p_i}\rangle, \$) := \left((\vec{b}, q_0^{p_i}), (t^{p_i}, \vec{b}, q_{root}), \sigma\right)$$

   where $q_0^{p_i}$ is the initial state of $\mathcal{A}^{p_i}$, and $\langle \vec{b}, \sigma\rangle = f^{p_i}_{\$[}\left(q_\perp, \ldots, q_\perp\right)$ with $\vec{b} \in Dom^k$ and $\sigma \in (\Delta^{\mathcal{T}})^*$.
   If $f^{p_i}_{\$[}\left(q_\perp, \ldots, q_\perp\right)$ is undefined then the above transition is not defined either and $\mathcal{T}$ will reject all corresponding XML documents.
   To define the remaining transitions, we consider each production $p_i \in P$ and its associated DFA $\mathcal{A}^{p_i}$.
   (a) For every DFA transition $\delta^{p_i}(q, l) = q'$ with states $q, q' \in Q_{DFA}$ and nonterminal l and for each terminal t such that there is a production $p_j : l ::= t(\rho)$ for some ρ, we define a transition of $\mathcal{T}$:
      (i) If $t \in Tag$, then we define a transition for reading an XML start tag from the input stream. This involves pushing a symbol on the stack: For $\vec{a} \in Dom^k$ and $X \in \Gamma^{\mathcal{T}}$,

      $$\delta^{\mathcal{T}}\left((\vec{a}, q), \langle t\rangle, X\right) := \left((\vec{b}, q_0^{p_j}), (t, \vec{b}, q')X, \sigma\right)$$

      where $q_0^{p_j}$ is the initial state of $\mathcal{A}^{p_j}$, the DFA for the next production, and $\langle \vec{b}, \sigma\rangle = f^{p_j}_{\$[}(\vec{a})$ with $\vec{b} \in Dom^k$ and $\sigma \in (\Delta^{\mathcal{T}})^*$.
      If $f^{p_j}_{\$[}(\vec{a})$ is undefined then the above transition is not defined either and $\mathcal{T}$ will reject all corresponding XML documents.
      (ii) If $t \in Char$, then we define a transition for reading a character symbol, so the stack remains unchanged: For $\vec{a} \in Dom^k$ and $X \in \Gamma^{\mathcal{T}}$,

      $$\delta^{\mathcal{T}}((\vec{a}, q), t, X) := ((\vec{c}, q'), X, \sigma\sigma')$$

      where $\langle \vec{b}, \sigma\rangle = f^{p_j}_{\$[}(\vec{a})$ and $\langle \vec{c}, \sigma'\rangle = f^{p_j}_{\$]}(\vec{b}, \vec{b})$ with $\vec{b}, \vec{c} \in Dom^k$ and $\sigma, \sigma' \in (\Delta^{\mathcal{T}})^*$.
      If $f^{p_j}_{\$[}(\vec{a})$ or $f^{p_j}_{\$]}(\vec{b}, \vec{b})$ is undefined then the above transition is not defined either and $\mathcal{T}$ will reject all corresponding XML documents.

(b) Next, we consider transitions related to reading the XML end tag $\langle /t^{p_i} \rangle$ for the current production p_i. Here, the topmost symbol is popped off the stack: For states $q_f \in F^{p_i}$, a state of a previous production $q_{prev} \in Q_{DFA} \cup \{q_{root}\}$, and attribute tuples $\vec{a}, \vec{b} \in Dom^k$, we define

$$\delta^{\mathcal{T}}((\vec{b}, q_f), \langle /t^{p_i} \rangle, (t^{p_i}, \vec{a}, q_{prev})) := ((\vec{c}, q_{prev}), \epsilon, \sigma)$$

where $\langle \vec{c}, \sigma \rangle = f_{\$]}^{p_i}(\vec{a}, \vec{b})$ with $\vec{c} \in Dom^k$ and $\sigma \in (\Delta^{\mathcal{T}})^*$.
Again, if $f_{\$]}^{p_i}(\vec{a}, \vec{b})$ is undefined then the above transition is not defined either and $\mathcal{T}$ will reject all corresponding XML documents.

$\mathcal{T}$ is an *XML-DPDT* by its very construction: $\mathcal{T}$ is deterministic as $\mathcal{T}$ has no ϵ-transitions and $\delta^{\mathcal{T}}$ is a function. Also, $\mathcal{T}$ adheres to the stack discipline required for *XML-DPDT*s and rejects malformed XML documents.

It can be shown by induction on the length of input words that G and $\mathcal{T}$ are equivalent. □

Theorem 4 *For each XML-DPDT there is an equivalent XSAG which is both a basic and an easy XSAG.*

Proof Idea. In brief, we construct an XSAG G which simulates a given *XML-DPDT* and is both a bXSAG and a yXSAG. The grammar component of G will be very general, defining the language of all well-formed XML documents over the input alphabet of the *XML-DPDT*. We enforce the equivalence of G and $\mathcal{T}$ by means of the attribution functions, exploiting the ability of attribution functions to reject the input.

Proof Let $\mathcal{T} = (Q^{\mathcal{T}}, \Sigma^{\mathcal{T}}, \Gamma^{\mathcal{T}}, \Delta^{\mathcal{T}}, \delta^{\mathcal{T}}, q_0^{\mathcal{T}}, Z_0^{\mathcal{T}})$. Then $\Sigma^{\mathcal{T}} = \{\langle t \rangle \mid t \in Tag\} \cup \{\langle /t \rangle \mid t \in Tag\} \cup Char$ and $\Gamma^{\mathcal{T}} = \Sigma^{\mathcal{T}} \times \Gamma'$ for set of symbols Γ'. By Lemma 1 we may assume $\mathcal{T}$ to be ϵ-free. Then we define an XSAG $G = (Nt, T, P, s)$ as follows:

1. Let $Att = \{a_1, a_2\}$ be the set of attributes with domain $Dom = Q^{\mathcal{T}} \cup \Gamma' \cup \{Z_0^{\mathcal{T}}\} \cup \{q_\perp\}$ where $q_\perp$ is the special "uninitialized" value. Attribute a_1 represents the current state of $\mathcal{T}$ while attribute a_2 represents what is currently stored on top of the stack of $\mathcal{T}$ besides the tag of the current production.
2. We define $Nt = \{nt_{Tag}, nt_{Char}\}$.
3. Naturally, $T = Tag \cup Char$.
4. We successively define the set of productions which is initially empty. For each terminal $t \in T$, we add a production $p^t : \quad nt^t ::= \{f_{\$[}^t\}\ t(\rho^t)\ \{f_{\$]}^t\}$.
   - If $t \in Tag$, then $nt^t := nt_{Tag}$, $\rho^t := (nt_{Tag} \cup nt_{Char})^*$, and the attribution functions are defined as follows: (1) For each transition

     $$\delta^{\mathcal{T}}(q_0, \langle t \rangle, Z_0^{\mathcal{T}}) = (q', (t, Y), \sigma)$$

     with $q' \in Q^{\mathcal{T}}$, $Y \in \Gamma'$, and $\sigma \in (\Delta^{\mathcal{T}})^*$ we define $f_{\$[}^t(q_\perp, q_\perp) := \langle q', Y, \sigma \rangle$. At the beginning of the XSAG evaluation, both attributes are set to $q_\perp$. Thus, $f_{\$[}^t$ ensures that start tag of the document root node has the label required by $\mathcal{T}$. For each transition

     $$\delta^{\mathcal{T}}(q, \langle t \rangle, (u, X)) = (q', (t, Y)(u, X), \sigma)$$

     with $q, q' \in Q^{\mathcal{T}}$, $\langle u \rangle \in \Sigma^{\mathcal{T}}$, $X, Y \in \Gamma'$, and $\sigma \in (\Delta^{\mathcal{T}})^*$ we define $f_{\$[}^t(q, X) := \langle q', Y, \sigma \rangle$. For all other input, $f_{\$[}^t$ is undefined. Further, (2) for each transition

     $$\delta^{\mathcal{T}}(q, \langle /t \rangle, (t, X)) = (q', \epsilon, \sigma)$$

     with $q, q' \in Q^{\mathcal{T}}$, $X \in \Gamma'$, $\sigma \in (\Delta^{\mathcal{T}})^*$, and for each $p \in Q^{\mathcal{T}}$ and $X' \in \Gamma'$, we define $f_{\$]}^t(p, X', q, X) := \langle q', X', \sigma \rangle$. For all other input, $f_{\$]}^t$ is undefined.
   - For each $t \in Char$, we assign $nt^t := nt_{Char}$, $\rho^t := \epsilon$, and the attribution functions as follows: For each transition

     $$\delta^{\mathcal{T}}(q, t, (u, X)) = (q', (u, X), \sigma),$$

     with $q, q' \in Q^{\mathcal{T}}$, $u \in Tag$, $X \in \Gamma'$, and $\sigma \in (\Delta^{\mathcal{T}})^*$, we define $f_{\$[}^t(q, X) := \langle q', X, \sigma \rangle$. For all other input, $f_{\$[}^t$ is undefined. The second-visit attribution function has default functionality, i.e. $f_{\$]}^t := f_{\$]}^d$.
5. The grammar start symbol is $s := nt_{Tag}$, ensuring that the grammar component is an ERTG.

The STDLL(1) grammar component of G accepts all well-formed documents over $\Sigma^{\mathcal{T}}$, as for all productions for a terminal t, $\tau(\rho^t) = (\bigcup T)^*$. The partial definitions of $\delta^{\mathcal{T}}$ translate into partial attribution functions in the natural way, and so does the use of the stack in $\mathcal{T}$ translate to the use of the implicit stack of an XSAG.[13]

Note that G is a bXSAG, as the regular expressions on the right-hand sides of productions are not attributed. Moreover, G is also a yXSAG: For each production p^t, ρ^t is either ϵ or $\tau(\rho^t)$ is strongly one-unambiguous.

It is not difficult to see that XSAG G of our construction is indeed equivalent to *XML-DPDT* $\mathcal{T}$. This can be shown by induction on the length of input words. □

Theorem 5 *For each easy XSAG there is an equivalent XML-DPDT.*

Proof Idea. Our strategy in designing an *XML-DPDT* $\mathcal{T}$ to simulate a yXSAG G is very similar to the proof of Theorem 3: We evaluate those attribution functions which do not occur inside attributed regular expressions using the

[13] XSAGs can be considered to have an implicit stack which is made accessible through the second-visit attribution functions. Function $f_{\$]}^v$ in $F_{\$]}$, assigned to node v in the attributed parse tree, has access to two sets of attribute values: (1) the attribute values as current after returning from the children of node v, and (2) the values of the attribute at the time that were current just before the children of node v were processed. For an attribute $a \in Att$, we denote (1) by $(a)_{\$].in}^v$ and (2) by $(a)_{\$[.out}^v$ in our notation from Definition 8.

stack of $\mathcal{T}$ (as done in the proof of Theorem 3). That is, we push the attribute values computed in the first visit to a node on the stack such that they are available in the second visit.

As stack operations are restricted to reading XML start or end tags, we need to apply a trick to achieve this stack functionality for the attribution functions inside attributed regular expressions as well: We simulate a stack in the states of $\mathcal{T}$, exploiting the fact that the depth of this simulated stack is bounded by the maximum height of the parse trees for attributed regular expressions occurring in productions.

This allows us to evaluate both kinds of attribution functions in a similar manner, using the *stack* of $\mathcal{T}$ in the first and the stack simulated in the *states* of $\mathcal{T}$ in the second case.

Proof Let *Dom*, *Att*, $F_{\$[}$ and $F_{\$]}$ be defined as in the related proof. Let $G = (Nt, T, P, s)$ be a yXSAG with set of nonterminals *Nt*, terminals $T = Tag \cup Char$, productions $P = \{p_1, \ldots, p_n\}$, and grammar start symbol s. We assume that the grammar component of G is reduced and that all productions $p_i \in P$ are of the form

$$p_i : \quad nt^{p_i} ::= \left\{f^{p_i}_{\$[}\right\} t^{p_i}(\alpha^{p_i}) \left\{f^{p_i}_{\$]}\right\}$$

with $nt^{p_i} \in Nt$, $t^{p_i} \in T$, attribution functions $f^{p_i}_{\$[} \in F_{\$[}$ and $f^{p_i}_{\$]} \in F_{\$]}$, and with default attribution functions in places where attribution functions have not been explicitly declared. α^{p_i} is either ϵ or an attributed regular expression.

Let $\odot$ be a special end-marker symbol $\{\odot\} \notin Nt$. With each production $p_i \in P$, we associate an FST

$$\mathcal{A}^{p_i} = (Q^{p_i}, \Sigma^{p_i}, \Delta^{p_i}, \delta^{p_i}, q_0^{p_i}, F^{p_i})$$

with state set Q^{p_i}, input alphabet $\Sigma^{p_i} \subseteq Nt \cup \{\odot\}$, output alphabet $\Delta^{p_i} \subseteq F_{\$[} \cup F_{\$]}$, initial state $q_0^{p_i}$, and set of final states F^{p_i}. The construction of this FST depends on α^{p_i}:

1. If $\alpha^{p_i} = \epsilon$, then let $\mathcal{A}^{p_i}$ be a DFT recognizing $L(\odot)$ and producing no output.
2. If α^{p_i} is an attributed regular expression, then we assume that every subexpression in α^{p_i} is of the form "$f_{\$[}\ \pi^{p_i}\ f_{\$]}$" with attribution functions $f_{\$[} \in F_{\$[}$ and $f_{\$]} \in F_{\$]}$ (or default attribution functions if these attribution functions have not been explicitly declared).
   Let ρ^{p_i} be the regular expression obtained from α^{p_i} by removing the attributions. Clearly, as $\tau(\rho^{p_i})$ is strongly one-unambiguous, so is ρ^{p_i} and we can construct the DFT $\mathcal{A}^{[]}(\rho^{p_i})$ from Theorem 1. $\mathcal{A}^{[]}(\rho^{p_i})$ accepts the language $L(\rho^{p_i}.\odot)$ and outputs the bracketing of a word w such that $w.\odot \in L(\rho^{p_i}.\odot)$.
   We define $\mathcal{A}^{p_i}$ in analogy to $\mathcal{A}^{[]}(\rho^{p_i})$, yet with the difference that it outputs the identifiers of the attribution functions instead of the bracketings of words.

W.l.o.g., for all productions $p_i \in P$ we may assume that the state sets Q^{p_i} are pairwise disjoint and we define $Q_{DFT} := \bigcup_i Q^{p_i}$ as the union of all such states. Further, we assume a special state $q_{root} \notin Q_{DFT}$.

Let h be the maximum height of all parse trees for attributed regular expressions in the productions of G. We define the simulated stack containing up to h k-tuples of attributes. That is, $S \in \mathcal{S} = (Dom^k)^{\leq h}$ denotes the current string of stack symbols and we write $\epsilon^{\mathcal{S}}$ for the empty stack.

Next, we define a recursive and partial function computing a composition of attribution functions:

$$\delta^* : (F_{\$[} \cup F_{\$]})^* \times Dom^k \times \mathcal{S} \times string^* \to Dom^k \times \mathcal{S} \times string^*$$

δ^* takes a sequence of attribution function identifiers, a tuple of attribute values, the simulated stack, and the output produced so far as input. One by one, the attribution functions are evaluated using the simulated stack. δ^* is either undefined for its input or it returns the resulting attribute values, simulated stack, and the computed output.

1. For the empty word and $\vec{a} \in Dom^k$, $S \in \mathcal{S}$ and $o \in string^*$, $\delta^*(\epsilon, \vec{a}, S, o) := (\vec{a}, S, o)$.
2. Consider a sequence of attribution functions $f = f_1 \ldots f_r$ in $(F_{\$[} \cup F_{\$]})^+$.
   If $f_1 \in F_{\$[}$, then for $\vec{a} \in Dom^k$, $S \in \mathcal{S}$, and $o \in string^*$, we evaluate
   $$\delta^*(f_1 f_2 \ldots f_r,\ \vec{a}, S, o) := \delta^*(f_2 \ldots f_r,\ \vec{b}, \vec{b}S, o\sigma)$$
   where $\langle \vec{b}, \sigma \rangle = f_1(\vec{a})$ with $\vec{b} \in Dom^k$, $\vec{b}S \in \mathcal{S}$, and $\sigma \in string^*$. If $f_1(\vec{a})$ or $\delta^*(f_2, \ldots, f_r, \vec{b}, \vec{b}S, o\sigma)$ are not defined then $\delta^*(f, \vec{a}, S, o)$ is not defined either.
3. If $f_1 \in F_{\$]}$, then for $\vec{a}, \vec{b} \in Dom^k$, $\vec{a}S \in \mathcal{S}$, and $o \in string^*$, we evaluate
   $$\delta^*(f_1 f_2 \ldots f_n,\ \vec{b}, \vec{a}S, o) := \delta^*(f_2 \ldots f_r,\ \vec{c}, S, o\sigma)$$
   where $\langle \vec{c}, \sigma \rangle = f_1(\vec{a}, \vec{b})$ with $\vec{c} \in Dom^k$, $S \in \mathcal{S}$, and $\sigma \in string^*$. If $f_1(\vec{a}, \vec{b})$ or $\delta^*(f_2 \ldots f_r, \vec{c}, S, o\sigma)$ are not defined then $\delta^*(f, \vec{b}, \vec{a}S, o)$ is not defined either.

We then construct an *XML-DPDT*

$$\mathcal{T} = \left(Q^{\mathcal{T}}, \Sigma^{\mathcal{T}}, \Gamma^{\mathcal{T}}, \Delta^{\mathcal{T}}, \delta^{\mathcal{T}}, q_0^{\mathcal{T}}, Z_0^{\mathcal{T}}\right)$$

as follows:

1. The state set is $Q^{\mathcal{T}} = Dom^k \times (Q_{DFT} \cup \{q_{root}\}) \times \mathcal{S}$. A state $(a_1, \ldots, a_k, q, S) \in Q^{\mathcal{T}}$ consists of the k current attribute values $a_1, \ldots, a_k$, state marker q denoting the state of the currently active DFT (if $q \in Q_{DFT}$) or the beginning and end of the input (if $q = q_{root}$), and the simulated stack $S \in \mathcal{S}$.
2. The initial state is $q_0^{\mathcal{T}} = (q_\perp, \ldots, q_\perp, q_{root}, \epsilon^{\mathcal{S}})$, where all attributes are initialized with $q_\perp$, the state marker is set to q_{root}, and the simulated stack is empty.
3. Input alphabet $\Sigma^{\mathcal{T}}$, stack alphabet[14] $\Gamma^{\mathcal{T}}$ with initial stack symbol $Z_0^{\mathcal{T}} = \{\$\}$, and output alphabet $\Delta^{\mathcal{T}}$ are defined in analogy to the proof of Theorem 3.

[14] For the sake of syntactic brevity, we will write stack symbols in $Tag \times Q^{\mathcal{T}} = Tag \times (Dom^k \times (Q_{DFT} \cup \{q_{root}\}) \times \mathcal{S})$ as if they were tuples in $Tag \times Dom^k \times (Q_{DFT} \cup \{q_{root}\}) \times \mathcal{S}$.

4. We next define the transition function

$$\delta^{\mathcal{T}} : Q^{\mathcal{T}} \times \Sigma^{\mathcal{T}} \times \Gamma^{\mathcal{T}} \to Q^{\mathcal{T}} \times (\Gamma^{\mathcal{T}})^{\leq 2} \times (\Delta^{\mathcal{T}})^*.$$

We begin with the transitions for reading the XML start tag of the root node: For each start production p_i, i.e., productions with nonterminal s on their left-hand sides, with associated DFT $\mathcal{A}^{p_i}$, we define a transition

$$\delta^{\mathcal{T}}\big(q_0^{\mathcal{T}}, \langle t^{p_i}\rangle, Z_0^{\mathcal{T}}\big) = \delta^{\mathcal{T}}\big((q_\perp, \ldots, q_\perp, q_{root}, \epsilon^{\mathcal{S}}), \langle t^{p_i}\rangle, \$\big) := \big((\vec{b}, q_0^{p_i}, \epsilon^{\mathcal{S}}), (t^{p_i}, \vec{b}, q_{root}, \epsilon^{\mathcal{S}}), \sigma\big),$$

where $q_0^{p_i}$ is the initial state of $\mathcal{A}^{p_i}$, further $\langle \vec{b}, \sigma\rangle = f_{\$[}^{p_i}(q_\perp, \ldots, q_\perp)$ with $\vec{b} \in Dom^k$ and $\sigma \in (\Delta^{\mathcal{T}})^*$.
If $f_{\$[}^{p_i}(q_\perp, \ldots, q_\perp)$ is undefined then the above transition is not defined either and $\mathcal{T}$ will reject all corresponding XML documents.
Next, we attend to the remaining transitions of $\mathcal{T}$. For each production $p_i \in P$ we consider the corresponding DFT $\mathcal{A}^{p_i}$: For each DFT transition

$$\delta^{p_i}(q, l) = (p, w)$$

where $q, p \in Q_{DFT}, l \in \Sigma^{p_i}$, producing a sequence of attribution functions $w \in (F_{\$[} \cup F_{\$]})^*$, we define a transition $\delta^{\mathcal{T}}$:
If $l \in Nt$, then for every terminal t such there is a production of the form $p_j : l ::= t(\rho)$ for some ρ, we distinguish between two cases:

(a) If $t \in Tag$, then we define a transition for reading an XML start tag. For $\vec{a} \in Dom^k$, $S \in \mathcal{S}$, and $X \in \Gamma^{\mathcal{T}}$,

$$\delta^{\mathcal{T}}((\vec{a}, q, S), \langle t\rangle, X) := \big((\vec{b}, q_0^{p_j}, \epsilon^{\mathcal{S}}), (t, \vec{b}, p, S')X, \sigma\sigma'\big)$$

where $q_0^{p_j}$ is the initial state of the DFT corresponding to the next production p_j. The values $\vec{b} \in Dom^k$, $\sigma, \sigma' \in (\Delta^{\mathcal{T}})^*$, and $S' \in \mathcal{S}$ are computed in two steps. First, we simulate a composition of attribution functions, as defined by w,

$$\langle \vec{a}', S', \sigma\rangle = \delta^*\big(w, \vec{a}, S, \epsilon\big)$$

with $\vec{a}' \in Dom^k$. Next, we also evaluate the first-visit attribution function from the next production p_j,

$$\langle \vec{b}, \sigma'\rangle = f_{\$[}^{p_j}\big(\vec{a}'\big).$$

If $\delta^*\big(w, \vec{a}, S, \epsilon\big)$ or $f_{\$[}^{p_j}\big(\vec{a}'\big)$ are not defined then the above transition is not defined either and $\mathcal{T}$ will reject all corresponding documents.

(b) If $t \in Char$, then we define a transition for reading a character symbol. For $\vec{a} \in Dom^k$, $S \in \mathcal{S}$, and $X \in \Gamma^{\mathcal{T}}$, we define

$$\delta^{\mathcal{T}}((\vec{a}, q, S), t, X) := ((\vec{b}, p, S'), X, \sigma\sigma'\sigma'').$$

The values $\vec{b} \in Dom^k$, $\sigma, \sigma', \sigma'' \in (\Delta^{\mathcal{T}})^*$, and $S' \in \mathcal{S}$ are computed in two steps. First, we simulate a composition of attribution functions denoted by w, as

$$\langle \vec{a}', S', \sigma\rangle = \delta^*(w, \vec{a}, S, \epsilon)$$

with $\vec{a}' \in Dom^k$. Next, we also evaluate the first- and second-visit attribution functions from production p_j by $\langle \vec{a}'', \sigma'\rangle = f_{\$[}^{p_j}(\vec{a}')$ with $\vec{a}'' \in Dom^k$, and $\langle \vec{b}, \sigma''\rangle = f_{\$]}^{p_j}(\vec{a}'', \vec{a}'')$.
If $\delta^*(w, \vec{a}, S, \epsilon)$, $f_{\$[}^{p_j}(\vec{a}')$, or $f_{\$]}^{p_j}(\vec{a}'', \vec{a}'')$ are not defined then the above transition is not defined either and $\mathcal{T}$ will reject all corresponding documents.

We now consider the case where $l = \odot$, and consequently, p is final, and we define a transition for reading an XML end tag. For $\vec{a}, \vec{b} \in Dom^k$, a state of a previous production $q_{prev} \in Q_{DFT} \cup \{q_{root}\}$, and $S, S' \in \mathcal{S}$, we define

$$\delta^{\mathcal{T}}((\vec{b}, q, S), \langle/t^{p_i}\rangle, (t^{p_i}, \vec{a}, q_{prev}, S')) := ((\vec{c}, q_{prev}, S'), \epsilon, \sigma\sigma').$$

Again, the values $\vec{c} \in Dom^k$ and $\sigma, \sigma' \in (\Delta^{\mathcal{T}})^*$ are computed by evaluating a composition of attribution functions. First, by

$$\langle \vec{b}', S'', \sigma\rangle = \delta^*(w, \vec{b}, S, \epsilon)$$

with $\vec{b}' \in Dom^k$ and $S'' \in \mathcal{S}$.[15] Finally,

$$\langle \vec{c}, \sigma'\rangle = f_{\$]}^{p_i}(\vec{a}, \vec{b}').$$

If $\delta^*(w, \vec{b}, S, \epsilon)$ or $f_{\$]}^{p_i}(\vec{a}, \vec{b}')$ are not defined then the above transition is not defined either and $\mathcal{T}$ will reject all corresponding documents.

$\mathcal{T}$ is an *XML-DPDT* by its very construction (we refer to the arguments stated in the proof of Theorem 3).

It can be shown by induction on the length of the input word that G and $\mathcal{T}$ are equivalent. □

[15] Note that reading "$\odot$" corresponds to processing the matching XML end tag, so the evaluation of the sequence of attribution functions w will leave the simulated stack empty, i.e., $S'' = \epsilon^{\mathcal{S}}$.

6 Efficient evaluation of XSAGs

Evaluating an XSAG G on an input document or tree T requires (1) the translation of an XSAG to a transducer and (2) the evaluation of the transducer on T.

As shown in the proofs of Theorems 3 and 5, we can translate an XSAG directly to an equivalent DPDT, a straightforward strategy for XSAG evaluation. This yields an exponential-time algorithm for query processing; however, the exponentiality is only with respect to the size of the XSAG:

Corollary 1 *An XSAG G can be evaluated on a tree T in time $O(f(|G|) + |T|)$ using a stack of memory of size $O(depth(T))$.*

Thus, the problem of evaluating an XSAG on an XML tree is fixed-parameter linear [15] (with the XSAG as the parameter).

Proof We first consider the translation step (1). The time to translate an XSAG G to a DPDT depends on whether G is a basic or an easy XSAG.

- For bXSAGs, the *XML-DPDT*s constructed in the proof of Theorem 3 are of size exponential in the number k of attributes in the XSAG, i.e., f is $O(2^k)$.
- For yXSAGs, the *XML-DPDT*s constructed in the proof of Theorem 5 are additionally exponential in the maximum depth h of the parse trees of the regular expressions used in productions, where h only depends on the XSAG.

(2) Once the *XML-DPDT* has been created, the query evaluation time is in principle independent of the size of the XSAG or *XML-DPDT* and only depends on the input data.

Due to the nature of *XML-DPDT*s, memory consumption is bounded at any time during query evaluation, being proportional to the depth of the input tree. □

By using a simple hybrid evaluation method, the exponential-time compilation phase can be avoided: The grammars (and in particular the regular expressions appearing in the grammar productions) are compiled into transducers which however *interpret* attribution functions (rather than materializing the *graphs* of the attribution functions as is done in our proofs). Thus, one obtains an XSAG evaluation method which runs scalably on streams and which is strictly polynomial in the size of the XSAG. These hybrid transducers and their construction are presented in more detail in [32].

Theorem 6 *A basic XSAG G can be evaluated on a tree T in time $O(|G|^2+|T|\cdot|G|)$ using a stack of size $O(depth(T))$.*

Proof In the XSAG translation (1), the grammar component of bXSAG G is translated into a transducer $\mathcal{T}$. The step dominating runtime is that of computing the Glushkov automata from the regular expressions on the right-hand sides of productions. This can be done in quadratic time [11], where the size of the automaton is quadratic in G.

During the evaluation of $\mathcal{T}$ (2), attribution functions are interpreted according to their definition (see Sect. 4.3) during transitions of $\mathcal{T}$. As each statement in an attribution function can be executed in constant time, the evaluation requires time $O(|T| \cdot |G|)$.

As bXSAG productions carry at most two attribution functions on their right-hand side, the stack discipline of $\mathcal{T}$ can be restricted as follows: Upon reading an XML start tag, a new set of attributes is computed and stored on the stack. Upon reading an end tag, the attributes are removed from the stack, while processing character data leaves the stack unchanged. Memory consumption is thus linear in the depth of the TDLL(1) parse tree for the input document. □

The main technical challenge we have to deal with when evaluating yXSAGs in the hybrid approach is the matching of attributed regular expressions on the stream and the invocation of attribution functions at the right time. Using the DFT construction of Theorem 1, preprocessing yXSAGs takes time cubic in the size of each of the productions.

Theorem 7 *An easy XSAG G can be evaluated on a tree T in time $O(|G|^3+|T|\cdot|G|)$ using a stack of size $O(depth(T)\cdot|G|)$.*

Proof For XSAG translation (1) in the hybrid evaluation of XSAGs, we translate the grammar component of a yXSAG G into a transducer $\mathcal{T}$. The exponential-time compilation phase in the translation of yXSAGs to transducers in the proof of Corollary 1 can be avoided by pushing attributes onto the stack at yXSAG regular expression nodes as well. We use the DFTs from Theorem 1 to derive the transitions of $\mathcal{T}$. By Theorem 2, this translation can be effected in time cubic in the size of G.

During XSAG evaluation (2), $\mathcal{T}$ outputs a sequence of identifiers of attribution functions in each transition, again yielding an overall evaluation time of $O(|T| \cdot |G|)$.

As attributes are also stored on the stack at yXSAG regular expression nodes, $\mathcal{T}$ does not cohere to the stack discipline required for *XML-DPDT*s. However, the stack consumption during query evaluation still remains proportional to the depth of the STDLL(1) parse tree for the input document, i.e., in $O(depth(T) \cdot |G|)$. □

7 Discussion and conclusion

The goal of this paper was to develop a framework for query formulation which

1. satisfies our criteria for scalable query processing on XML streams,
2. has a good and well-justified foundation, and
3. is user-friendly, i.e., allows us to state many common queries quickly and easily.

We can argue that XSAGs satisfy these three desiderata.

(1) Each XSAG can be translated into a DPDT with a stack discipline that assures that the size of the stack remains

proportional to the depth of the XML tree. This is known to be the minimum amount of memory required to do any meaningful (sequential) processing of XML data [21]. Of course, queries are evaluated strictly in linear time in the size of the input. A straightforward implementation of the hybrid XSAG evaluation method [32] confirms these theoretical findings.

(2) Throughout the paper, we have explained and justified our design choices. Regular tree grammars are a commonly accepted grammar formalism for XML (as are DTDs, which are restricted regular tree grammars). In the right-hand side of the productions of such grammars, we use regular expressions to be able to parse nodes with an unbounded number of children. Our restriction of these regular expressions to strongly one-unambiguous ones in the case of yXSAGs allows for precisely those expressions for which the parse trees of words can be unambiguously generated using a lookahead of only one symbol (a necessity in stream processing). Having the regular expressions inside grammar productions available for attribution allows us to conveniently define attribute grammars for unranked trees, and to approach the usability of XML Query with a formalism that allows for much better control of complexity.

We have precisely characterized the expressive power of XSAGs relative to deterministic pushdown transducers.

Note that our formalism fully fits into the classical framework of attribute grammars (and more precisely, L-attributed grammars), even if we did not introduce, say, the distinction between synthesized and inherited attributes.

(3) A number of examples in this paper and our experiences with many more demonstrate that XSAGs are of practical value, and that they fill an important void in the design space of tailored query languages.

For instance, XSAGs can be used to evaluate Boolean navigational XPath, also known as Core XPath [17], with child and descendant location steps, and further Boolean conditions with operators $\vee$, $\wedge$, and $\neg$.

Proposition 4 (Folklore) *Let Q be a Boolean navigational XPath query using child and descendant axis and let G be a DTD. Q can be evaluated on an XML document $T \in L(G)$ in time $O((|Q| + |G|)^2 + |T| \cdot |Q|)$ using main memory bounded by $O(depth(T) \cdot |Q|)$.*

To encode queries from this XPath fragment with bXSAGs, we define Boolean attributes for every location step in the query. These attributes keep track of whether a node in the query has already been matched, thus requiring only memory linear in the size of the query and the depth of the input document. We then evaluate these bXSAG in the hybrid approach (see Theorem 6).

Earlier in this paper, we defined XSAGs with attributes ranging exclusively over a finite domain to be able to assure scalability and memory bounds in the strongest sense. However, it is desirable and often justified to generalize this framework to make certain uniformity assumptions and to allow for values from an infinite domain.

In the future, we plan to carry out a more detailed study of conservative extensions of our formalism with small buffers (using uniformity assumptions for numbers, small strings, and small subtrees). As a first step, we have developed a rewrite formalism for XQuery which aims at buffer minimization by exploiting schema knowledge [23].

Acknowledgements S. Scherzinger has been partly funded by the Austrian Federal Ministry for Education, Science, and Culture, and the European Social Fund (ESF) under grant 31.963/46-VII/9/2002.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers – Principles, Techniques, and Tools. (Addison-Wesley, 1986)
2. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling. I: Parsing, vol. 1 (Prentice-Hall, 1972)
3. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proc. STOC '04: 36th Annual ACM Symposium on Theory of Computing, pp. 202–211 (2004)
4. Benedikt, M., Chan, C.Y., Fan, W., Freire, J., Rastogi, R.: Capturing both types and constraints in data integration. In: Proc. SIGMOD 2003, pp. 277–288 (2003)
5. Benedikt, M., Chan, C.Y., Fan, W., Rastogi, R., Zheng, S., Zhou, A.: DTD-directed publishing with attribute translation grammars. In: Proc. VLDB 2002, pp. 838–849 (2002)
6. Berlea, A., Seidl, H.: Binary Queries for Document Trees. Nordic J. of Computing, **11**(1), 41–71 (2004)
7. Bohannon, P., Buneman, P., Choi, B., Fan, W.: Incremental evaluation of schema-directed XML publishing. In: Proc. SIGMOD 2004, pp. 503–514 (2004)
8. R., Book, S., Even, S., Greibach, Ott, G.: Ambiguity in graphs and expressions. IEEE Transactions on, Computers, **20**(2), 149–153 (1971)
9. Bray, T., Paoli, J. Sperberg-McQueen, C.M.: Extensible Markup Language (XML) 1.0. Technical report, W3C, (1998)
10. Brüggemann-Klein, A.: Regular expressions into finite automata. Theoretical Computer Science. **120**(2), 197–213 (1993)
11. Brüggemann-Klein, A., Wood, D.: One-unambiguous regular languages. Information and Computation. **142**(2), 182–206 (1998)
12. Cimprich, P., O.B., et al.: Streaming Transformations for XML (STX), (2004) Available at http://stx.sourceforge.net
13. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications. (2002) Available at http://www.grappa.univ-lille3.fr/tata/.
14. Crescenzi, V., Mecca, G.: Grammars have exceptions. Inf. Syst., **23**(9), 539–565 (1998)
15. Downey, R.G., Fellows, M.R.: Parameterized Complexity (Springer, 1999)
16. Fegaras, L., Levine, D., Bose, S., Chaluvadi, V.: Query Processing of streamed XML data. In: Proc. CIKM 2002, pp. 126–133 (2002)
17. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. In: Proc. VLDB 2002, pp. 95–106 (2002)
18. Green, T.J., Miklau, G., Onizuka, M., Suciu, D.: Processing XML streams with deterministic automata. In: Proc. ICDT'03, pp. 173–189 (2003)
19. Grohe, M., Koch, C., Schweikardt, N.: Tight lower bounds for query processing on streaming and external memory data. In: Proc. ICALP'05, pp. 1076–1088 (2005)
20. Gupta, A., Suciu, D.; Stream processing of XPath queries with predicates. In Proc. SIGMOD 2003, pp. 419–430 (2003)
21. Koch, C.: Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In: Proc. VLDB 2003, pp. 249–260 (2003)

22. Koch, C.: On the complexity of nonrecursive XQuery and functional query languages on complex values. In: Proc. PODS'05, pp. 84–97 (2005)
23. Koch, C., Scherzinger, S., Schweikardt, N., Stegmaier, B.: Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In: Proc. VLDB 2004, pp. 228–239 (2004)
24. Lee, D., Mani, M., Murata, M.: Reasoning about XML schema languages using formal language theory. Technical Report RJ 10197 Log 95071, IBM Research (2000)
25. Ludäscher, B., Mukhopadhyay, P., Papakonstantinou, Y.: A transducer-based XML query processor. In: Proc. VLDB 2002, pp. 227–238 (2002)
26. Murata, M., Lee, D., Kawaguchi, M.M.K.: Taxonomy of XML schema languages using formal language theory. ACM Transactions of Internet Technology, 2005. forthcoming.
27. Neven, F.: Extensions of attribute grammars for structured document queries. In: Proc. DBPL 1999, pp. 99–116 (1999)
28. Neven, F., van den Bussche, J.: Expressiveness of structured document query languages based on attribute grammars. Journal of the ACM, **49**(1), 56–100 (2002)
29. Olteanu, D., Furche, T., Bry, F.: Evaluating complex queries against XML streams with polynomial combined complexity. In: Proc. BNCOD 2004, pp. 31–44 (July 2004)
30. Peng, F., Chawathe, S.S.: XPath queries on streaming data. In: Proc. SIGMOD 2003, pp. 431–442 (2003)
31. Pitcher, C.: Visibly pushdown expression effects for XML stream processing. In: Proc. PLANX (2005)
32. Scherzinger, S.: Scalable Query Processing on XML streams. Diploma thesis, University of Passau, Germany, (2004) Available online at http://www.infosys.uni-sb.de/∼scherzin/thesis.pdf.
33. van der Steen, G.: A canonical query language and its efficient implementation. In XML Europe 2000 Conference Proceedings, pp. 543–548 (2000)
34. World Wide Web Consortium. XML Query (XQuery). http://www.w3c.org/XML/query/.