T U M

I N S T I T U T F Ü R I N F O R M A T I K

Event-driven Evaluation of Grammar Queries

Alexandru Berlea

TUM-I0513
Juli 05

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

Event-driven Evaluation of Grammar Queries

Alexandru Berlea

Technische Universität München, Germany
berlea@in.tum.de
Fax: +49 89 289 18161

Abstract. We present a solution for answering queries on XML streams. Our approach extends the class of queries for which streamed solutions have been proposed to the class of queries expressible by monadic second order logic. We provide an algorithm which efficiently answers the queries despite their large expressiveness. We show that the algorithm reports matches at the earliest possible time during the scan of the input which implicitly leads to high adaptiveness in terms of memory consumption. The efficiency is documented with an experimental evaluation of our approach.

1 Introduction

Most of the XML applications build in memory the tree representation of the manipulated XML data before processing it. This approach is not suitable for handling very large XML documents or settings in which the XML data is received linearly via some communication channel, rather than being completely available in advance. For these applications, special algorithms have to be developed, which view the XML data as a stream of events, rather than as a tree. An event contains a small piece of information, e.g. a *start-tag* or an *end-tag*. An application receiving the stream performs its task by reacting to the events. The advantage of this event-driven approach is that it allows one to buffer only the relevant parts of the input, saving thus time and memory. In particular, it allows the construction of the XML tree in memory and its subsequent processing, being thus at least as expressive as the tree-based approach.

A fundamental task in XML applications is locating nodes of an XML input tree which have a desired property. Here, we call these nodes *matches* and the process of locating them *querying*. The most widely known query-language is XPath [1] , used both standalone or as part of other important languages like XML Schema Language [2] , XSLT [3] or XQuery [4].

The research interest in querying XML streams has been very vivid recently and there is a very rich literature on this topic. The related work is reviewed in Section 8. The proposed query languages are generally able to implement different subsets of XPath. Most of them are subsumed by an XPath fragment called *Core XPath* [5], mainly featuring location paths and predicates using location paths but without arithmetics and data value comparisons, which is very relevant for the efficiency of the evaluation of full XPath queries.

Our main contribution is a novel solution for efficient event-based evaluation of queries which go beyond the capabilities of the languages for which this problem was addressed yet. Most of them can be expressed using first-order logic possibly extended with regular expressions on vertical paths, but are less expressive than monadic second order logic (MSO). In contrast, the queries that we consider here, *grammar queries*, are defined by using forest grammars [6, 7] (also called elsewhere regular hedge grammars [8]) which are equally expressive with the powerful MSO queries. Grammar queries basically cover all Core XPath queries, as opposed to most other event-driven approaches, which consider only fragments of it. Moreover, grammar queries allow the convenient specification of horizontal contextual conditions which are not possible or are difficult to express in XPath.

Grammar queries can be evaluated using pushdown forest automata as presented in [9, 6]. The original construction generally requires to build the whole input tree in memory. The event-based query evaluation is addressed only for a very restricted class of queries. These are the so called *right-ignoring* queries for which all the information needed to decide whether a node is a match has been seen by the time the end-tag of the node is encountered. The restriction of right-ignoring queries is very severe, as for instance they do not capture simple XPath patterns with qualifiers on the nodes in the path. In this paper we lift this restriction. Rather than a-priori (i.e. statically) handling only a restricted subset of queries, we show here how *arbitrary* grammar queries can be evaluated in an event-driven manner.

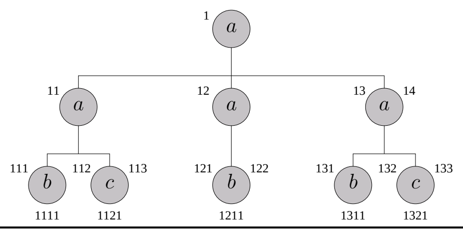


Fig. 1: Input tree t

Example 1. Consider an XML document whose tree representation is depicted in Fig. 1. Each location in the tree corresponds to an event in the corresponding stream of events, as depicted below. We identify locations by strings of numbers, s.t. the time ordering of the events corresponds to the lexicographic order of the locations:

```
 1   11  111 1111 112 1121  113  12  121 1211  122 ......
<a> <a> <b> </b> <c> </c> </a> <a> <b> </b> </a> ......
```

It should be clear that the amount of memory necessary to answer an arbitrary query inherently depends on the query and on the input document. Consider for example the XPath pattern `//a/b` locating `b` nodes which have as father an `a` node. The node 111 is a match in our input. This can be detected as early as at the location 111, as the events following 111 can not change the fact of 111 being a match.

The query `//a[c]/b` locates `b` nodes which have a node `a` as father and a `c` sibling. The node 111 is again a match but this becomes clear only after seeing that the `a` parent has also a child `c` at location 112. One has thus to remember 111 as a potential match between the events 111 and 112. As the events to the right of 112 can not change the fact of 111 being a match, 111 can be reported and discarded at 112.

Finally, as an extreme case consider the (MSO expressible) XPath pattern `/*[not(d)]//*` locating all descendant nodes of the root element if this has no child node `d`. Any node in the input is a potential match until seeing the last child of the root element. In our example all nodes have to be remembered as potential matches up to the last event 14. Note thus that any algorithm evaluating a query needs in the worst case linear space in the input size. However, most of the practical queries require a quite small amount of memory as compared to the size of the input.

The contributions of this paper are as follows. We introduce a method which allows one to talk about the earliest detection location of a match for some given query and input tree. The main contribution is proving that matches of grammar queries can be detected at their earliest detection location by an automata-based construction and hereby proving the following theorem:

Theorem 1. *Matches of MSO definable queries are recognizable at their earliest detection location.*

Based on the construction used for proving Theorem 1 we give an efficient algorithm for grammar query evaluation, which reports matches at their earliest detection point. As a consequence potential matches are remembered only as long as necessary, meaning that our construction implicitly adapts its memory consumption to the strict requirements of the query on the input at hand. The algorithm has been completely implemented in the freely available XML querying tool Fxgrep [7]. We provide experimental evidence of the practical performance of the algorithm.

The paper is organized as follows. In Section 2 we introduce a set of useful notations. In Section 3 we present the forest grammars and in Section 4 the grammar queries. Section 5 introduces the pushdown forest automata which are able to efficiently implement forest grammars. In Section 6 we briefly present how a pushdown automaton can be used to answer right-ignoring queries on streams. The main contribution on query evaluation for XML streams is given in Section 7 where the algorithm is presented and its correctness, optimality, complexity and performance are addressed. Related work is discussed in Section 8. We conclude in Section 9.

2 Preliminaries

Conceptually, an XML element is a *tree*. The content of an element is an ordered sequence of trees, which we call a *forest*. An XML document is, again, a forest as the root element might be preceded and followed by processing instruction trees. Therefore, we start by introducing a couple of useful notations for trees and forests. Let Σ be an alphabet. We formally define the sets $\mathcal{T}_\Sigma$ of trees t and $\mathcal{F}_\Sigma$ of forests f over Σ as follows:

$$\begin{array}{lcl} t & ::= & a\langle f\rangle,\ \ a \in \Sigma \\ f & ::= & \varepsilon \mid t_1 \ldots t_n,\ \ n > 0 \end{array}$$

where ε denotes the *empty forest*. The symbol a denotes the *label* and f the children of an element t. To denote the label of t we also write $lab(t) = a$.

Let f be a forest. The set $\Pi(f) \subseteq \mathbb{N}^*$ contains all paths π in f and is defined as follows:

$$\begin{aligned} \Pi(\varepsilon) &= \{\lambda\} \\ \Pi(t_1 \ldots t_n) &= \{\lambda\} \cup \{i\pi \mid 1 \leq i \leq n, \pi \in \Pi(f_i) \text{ for } t_i = a_i\langle f_i \rangle\} \end{aligned}$$

where λ denotes the empty string.

$N(f) = \Pi(f) \setminus \{\lambda\}$ is the set of nodes in f. A node identifies one of f's subtrees. For $\pi \in N(f)$, $f[\pi]$ is called the *subtree of f located at* π and is defined as follows:

$$(t_1 \ldots t_n)[i\pi] = \begin{cases} t_i \quad , & \text{if } \pi = \lambda \\ f_i[\pi], & \text{if } \pi \neq \lambda \text{ and } t_i = a\langle f_i \rangle \end{cases}$$

For a path π, we define $last_f(\pi)$ as the number of children of the node π:

$$last_f(\pi) = max(\{n \mid \pi n \in N(f)\} \cup \{0\})$$

Note that $last_f(\pi) = 0$ iff π identifies a leaf.

Let f be a forest. The set $L(f) \subseteq \mathbb{N}^*$ of *locations* in f, is defined by:

$$\begin{aligned} L(\varepsilon) &= \{1\} \\ L(t_1 \ldots t_n) &= \{i \mid 1 \leq i \leq n+1\} \cup \{il \mid 1 \leq i \leq n, l \in L(f_i) \\ &\qquad \text{for } t_i = a_i\langle f_i \rangle\} \end{aligned}$$

A location corresponds to the points in time at which events are triggered, with the time ordering being the lexicographical order of the locations. According to the definition above, the start-tag of a node l is seen at the moment l and the end-tag is seen at the moment $l(n+1)$, where $n = last_f(l)$ (see Example 1).

The *preceding nodes* of a location $l \in L(f)$ in a forest f are the set $prec_f(l) = \{\pi \mid \pi \in N(f), \pi < l\}$, where "$<$" denotes lexicographical comparison.

A forest f_2 is a *right-completion* of a forest f_1 at location $l \in L(f_1)$ iff f_1 and f_2 consists of the same events until l. Formally: $prec_{f_1}(l) = prec_{f_2}(l)$ and $lab(f_2[\pi']) = lab(f_1[\pi'])$ for all $\pi' \in prec_{f_1}(l)$.

3 Forest Grammars

Forest grammars are a very expressive and theoretically robust formalism for specifying properties of forests. Schema languages like DTD, XML Schema Language, DSD [10] or RelaxNG [11] basically are more or less restricted forms of forest grammars.

A *forest grammar* over an alphabet Σ is a tuple $G = (R, r_0)$, where R is a set of productions using non-terminals from a set X and terminal symbols from Σ, and $r_0 \in \mathcal{R}_X$ (the set of regular expressions over X) is the *start expression*.

The productions in R have the form $x \rightarrow a\langle r \rangle$ with $x \in X$, $a \in \Sigma$ and $r \in \mathcal{R}_X$. Intuitively, and using the terminology from schema languages, this specifies that the children of an a element derived using the production must conform to the *content model* r.

Example 2. The grammar $G = (R, x_1|x_a)$ over $\{a, b, c\}$ with R being the following set of productions will be used in our running example:

$$\begin{array}{llll} (1)\ x_\top \to a\langle x_\top^*\rangle & (3)\ x_\top \to c\langle x_\top^*\rangle & (5)\ x_b \to b\langle x_\top^*\rangle & (7)\ x_1 \to a\langle x_\top^*(x_1|x_a)x_\top^*\rangle \\ (2)\ x_\top \to b\langle x_\top^*\rangle & (4)\ x_a \to a\langle x_b x_c\rangle & (6)\ x_c \to c\langle x_\top^*\rangle & \end{array}$$

The XML language specified by G is the set of documents in which there is a path from the root to a node labeled a, whose children are a node labeled b and a node labeled c, and whose ancestors are all labeled a. The first three productions make $x_\top$ account for trees with arbitrary content. As specified by production (4), x_a stands for the a element with the b and the c children. Productions (5) and (6) say that these children can have arbitrary content. Finally, production (7) says that the a specified by (4) can be at arbitrary depth in the input, and all its ancestors must be labeled a. The start expression $x_1|x_a$ specifies that the root element is to be derived either from x_1 or from x_a.

The meaning of a grammar is formally defined as follows. A set of productions R together with a distinguished non-terminal $x \in X$ or a regular expression $r \in \mathcal{R}_X$ defines a *tree derivation* relation $\mathcal{D}eriv_{R,x} \in \mathcal{T}_\Sigma \times \mathcal{T}_X$ or a *forest derivation* relation $\mathcal{D}eriv_{R,r} \in \mathcal{F}_\Sigma \times \mathcal{F}_X$, respectively, as follows:

$(a\langle f\rangle, x\langle f'\rangle) \in \mathcal{D}eriv_{R,x}$ iff $x \to a\langle r\rangle \in R$ and $(f, f') \in \mathcal{D}eriv_{R,r}$

$(t_1 \ldots t_n, t'_1 \ldots t'_n) \in \mathcal{D}eriv_{R,r}$ iff $x_1 \ldots x_n \in [\![r]\!]_\mathcal{R}$
and $(t_i, t'_i) \in \mathcal{D}eriv_{R,x_i}$ for $i = 1, \ldots, n$

$(\varepsilon, \varepsilon) \in \mathcal{D}eriv_{R,r}$ iff $\lambda \in [\![r]\!]_\mathcal{R}$

where $[\![r]\!]_\mathcal{R}$ is the string language specified by the regular expression r.

If $(f, f') \in \mathcal{D}eriv_{R,r}$ we say that f' is a *derivation* of f w.r.t. R and r. For a grammar $G = (R, r)$ we write $(f, f') \in \mathcal{D}eriv_G$ and say that f' is a derivation of f w.r.t. the grammar G iff $(f, f') \in \mathcal{D}eriv_{R,r}$. Note that a derivation f' is a relabeling of f and can be seen as a proof of the validity of f according to the schema G.

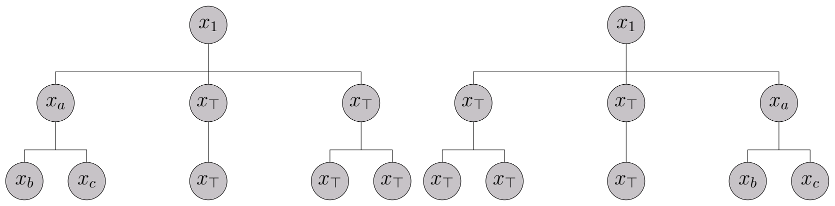


Fig. 2: Possible derivations of t

Example 3. Let t be the tree depicted in Fig. 1. Two possible derivations of t w.r.t. G are depicted in Fig. 2.

The *meaning* $[\![R]\!]$ of a set of productions R assigns sets of trees to the non-terminals $x \in X$ and sets of forests to regular expressions $r \in \mathcal{R}_X$ and is defined by:

$$t \in [\![R]\!]\, x \text{ iff } \exists t' \in \mathcal{T}_X \text{ and } (t,t') \in \mathcal{D}eriv_{R,x}$$
$$f \in [\![R]\!]\, r \text{ iff } \exists f' \in \mathcal{F}_X \text{ and } (f,f') \in \mathcal{D}eriv_{R,r}$$

If $t \in [\![R]\!]\, x$ or $f \in [\![R]\!]\, r$ we say that t can be derived from x or f can be derived from r, respectively.

A forest grammar $G = (R, r_0)$ specifies a *regular forest language* as the set of forests $\mathcal{L}_G = [\![R]\!]\, r_0$. This might be considered the XML language specified by G.

4 Grammar Queries

It is easy to see that a grammar G together with a distinguished non-terminal x specifies a query, namely, all the nodes π in the input f for which there is a derivation f' w.r.t. G in which π is labeled with x.

More generally, a *query* Q is a pair (G, T) consisting of a forest grammar $G = (R, r_0)$ and a set of *target non-terminals* $T \subseteq X$ where X is the set of non-terminals in R. The *matches* of Q in an input forest f are given by the set $\mathcal{M}_{Q,f} \subseteq N(f)$:

$$\pi \in \mathcal{M}_{Q,f} \text{ iff } \exists (f,f') \in \mathcal{D}eriv_G, \exists x \in T \text{ and } lab(f'[\pi]) = x$$

Example 4. The query $Q_1 = (G, \{x_b\})$ locates nodes b having only a ancestors and only one sibling c to the right. The leftmost b is a match, as one can see by definition by looking at the first derivation in Fig. 2. Similarly, the rightmost b is a match as defined by the second derivation w.r.t. G.

The query $Q_2 = (G, \{x_a\})$ locates the a nodes which have a child b followed by a child c. These are the leftmost and the rightmost a nodes.

Note that we decided for an *all-matches* semantics of our queries, i.e. all nodes π as in the definition are to be reported as matches. This is reasonable, because a user query typically is aimed at finding *all* the locations with the specified properties, as for instance in XPath. Furthermore we do not want to place on user the burden of specifying the query via an unambiguous grammar, therefore the definition above refers to *any* derivation.

Forest grammars can basically be thought of as non-deterministic (unranked) tree automata. Non-terminals correspond to states of tree automata and derivations to accepting runs of them. A grammar query can be thus seen as an automaton with a set of distinguished states. These queries were proven to be equally expressive with queries definable in MSO on forests [12]. MSO queries are not practicable due to their high evaluation complexity, yet they have been used a convenient benchmarks for comparing XML query languages [13] due to their large expressive power. Indeed MSO queries, subsume most of fundamental features of the query languages which have been proposed for XML. Grammar queries have thus the same expressive power as MSO queries, while being efficiently implementable, even in an event-based setting as we show in the next section.

While very expressive, grammar queries are not easily usable for users not-familiar with grammar formalisms. Alternatively, queries can be specified using the more intuitive Fxgrep [7] pattern language which is syntactically similar with XPath, but allows the specification of more precise horizontal contextual constraints on nodes in patterns. In particular, any node in a pattern can be provided with two regular expressions over patterns to be fulfilled by the sequence of the node's left and right siblings, respectively. This allows for example the identification of nodes by their XML schema like types, even in the absence of schema information. For an overview of Fxgrep and how its patterns are automatically translated to grammar queries we refer to [6].

Earliest Detection Locations

A location l is an *early detection location* of a match node π for a query Q in input f_1 iff $\pi \in \mathcal{M}_{Q,f_2}$ for all right-completions f_2 of f_1 at l.

A location l is the *earliest detection location* of a match node π iff l is the smallest early detection location of π in lexicographical order.

Example 5. Reconsider Example 1. Given the query `//a/b`, the earliest detection location of node 111 is 111. As for the query `//a[c]/b` the earliest detection location of node 111 is 112. Finally, for the query `/*[not(d)]//*`, there is no early detection location for any of the match nodes. This matches can not be detected until the last location in the input has been reached.

5 Automata for Forest Grammars

It is well known that regular ranked tree languages are recognizable by the class of bottom-up tree automata [14]. Also, every unranked tree can be encoded to a unique ranked tree representation and the notion of regular tree language is invariant under these encodings (see e.g. [6]). Therefore, bottom-up tree automata can be used to recognize regular forest languages.

Equally expressive with bottom-up automata but much more concise and efficient to implement in practice are the *pushdown forest automata* [9, 6]. Any implementation of a bottom-up automaton has to choose a traversal strategy for the input tree. The idea of a pushdown forest automaton (PA) is based on the observation that, when reaching a node during the traversal, the information gained from the already visited part of the tree can be used at the transitions of the automaton at that node. This supplementary information allows a significant reduction in the size of the states and the number of possible transitions to be considered by a deterministic PA as compared with a corresponding deterministic bottom-up automaton. Intuitively, in the case of a depth-first, left-to-right traversal, which corresponds to XML event-based processing, the advantage is that information gained by visiting the left siblings as well as the ancestors and their left siblings can be taken into account before processing the current node. The name of the automata (pushdown forest automata) is due to the fact that information from the visited part of the tree is stored on the stack (pushdown) which is implicitly used for the tree traversal.

Also, rather than working on ranked encodings of unranked trees, the PAs directly recognize unranked trees and forests. Besides saving the time needed for encoding, this also has the advantage of making the construction of the automata more straightforward and intelligible.

5.1 Pushdown Forest Automata

In addition to the tree states of classical tree automata, a PA also has *forest states*. Intuitively, a forest state contains the information gained from the already visited part of the tree (*context information*) at any point during the tree traversal. Let us consider a depth-first, left-to-right traversal. The following notations are essentially those introduced in [6].

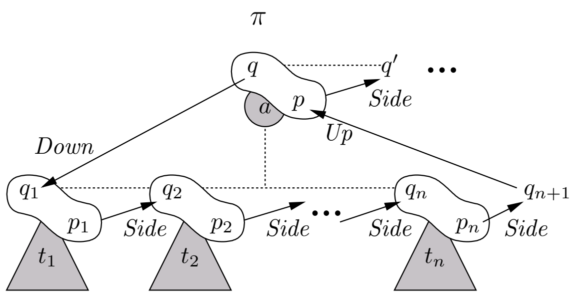


Fig. 3: The processing model of an LPA

The behaviour of a *left-to-right PA* (LPA) is depicted in Fig. 3. When arriving at some node π labeled a, the context information is available in the forest state q by which the automaton reaches the node. The automaton has to traverse the content of π and compute a tree state p, which describes π within the context q. In order to do so, the children of π are recursively processed. The context information for the first child, q_1, is obtained (via a $Down$ transition) by refining q by taking into account that the father is labeled a. Subsequently the q_2 context information for the second child is obtained (via a $Side$ transition) from q_1 and the information p_1 gained from the traversal of t_1. Proceeding in this manner, after visiting all π's children, enough context-information is collected in q_{n+1} in order to compute p (via an Up transition). After processing π the context information for the subsequent node is updated into q'.

Formally, an LPA $A = (P, Q, I, F, Down,\ Up, Side)$ consists of a finite set of *tree states* P, a finite set of *forest states* Q, a set of *initial states* $I \subseteq Q$, a set of *final states* $F \subseteq Q$, a *down-relation* $Down \subseteq Q \times \Sigma \times Q$, an *up-relation* $Up \subseteq Q \times \Sigma \times P$ and a *side-relation* $Side \subseteq Q \times P \times Q$. Based on $Down$, Up and $Side$, the behavior of A is described by the relations $\delta_{\mathcal{F}}^{A} \subseteq Q \times \mathcal{F}_{\Sigma} \times Q$ and $\delta_{\mathcal{T}}^{A} \subseteq Q \times \mathcal{T}_{\Sigma} \times P$ as follows, where the notations correspond to those in Fig. 3:

- ✧ $(q, a\langle t_1 \dots t_n\rangle, p) \in \delta_{\mathcal{T}}^{A}$ iff $(q, a, q_1) \in Down,\ (q_1, t_1 \dots t_n, q_{n+1}) \in \delta_{\mathcal{F}}^{A}$ and $(q_{n+1}, a, p) \in Up$ for some $q_1, q_{n+1} \in Q$.
- ✧ $(q_1, t_1 f, q_{n+1}) \in \delta_{\mathcal{F}}^{A}$ iff $(q_1, t_1, p_1) \in \delta_{\mathcal{T}}^{A},\ (q_1, p_1, q_2) \in Side$ and $(q_2, f, q_{n+1}) \in \delta_{\mathcal{F}}^{A}$ for some $p_1 \in P, q_2 \in Q$
- ✧ $(q_1, \varepsilon, q_1) \in \delta_{\mathcal{F}}^{A}$ for all $q_1 \in Q$

The language accepted by the automaton A is given by:

$$\mathcal{L}_A = \{f \in \mathcal{F}_\Sigma \mid \exists\, q_1 \in I, q_2 \in F \text{ and } (q_1, f, q_2) \in \delta^A_{\mathcal{F}}\}$$

Note that A visits the nodes in the input in document order, which is precisely the order in which the XML input is reported by an event-based parser. Therefore, there is no need to build the document tree in memory. Running A only needs a stack to remember the forest state q (see Fig. 3) at each opened and not yet closed tag and can be implemented in the event-based manner. A $Down$ transition is triggered by the start tag event `<a>`. The corresponding `</a>` event triggers the Up transition. A $Side$ transition is executed immediately after the preceding Up transition.

5.2 From Forest Grammars to PA

A compilation schema from a forest grammar $G = (R, r_0)$ into a deterministic LPA (DLPA) (where $Down$, Up and $Side$ are functions rather than relations) accepting the same regular forest language has been given in [6]. The idea is that at any time the DLPA keeps track of all possible content models of the elements whose content has not been seen all yet. In the case of an XML stream these are the *opened* elements, whose start-tags have been seen and whose end-tags have not been read yet, including an imaginary top element whose content is the whole document. The document is accepted at the end of the document if the sequence of top-level nodes conforms to r_0.

Let X be the set of non-terminals in G and let $r_1, \ldots, r_l$ be the regular expressions occurring on the right-hand sides in the productions R, where l is the number of productions. For $0 \leq j \leq l$, let $A_j = (Y_j, y_{0,j}, F_j, \delta_j)$ be the non-deterministic finite automaton (NFA) accepting the regular string language defined by r_j as obtained by the Berry-Sethi construction [15]. Here, Y_j is the set of NFA states, $y_{0,j}$ the start state, F_j the set of final states and $\delta_j \in Y_j \times X \times Y_j$ is the transition relation.

By possibly renaming the NFA states we can always ensure that $Y_i \cap Y_j = \emptyset$ for $i \neq j$. Let $Y = Y_0 \cup \ldots \cup Y_l$ and $\delta = \delta_0 \cup \ldots \cup \delta_l$. A DLPA $A_{\vec{G}}$ accepting $\mathcal{L}_G$ can be defined as $A_{\vec{G}} = (2^X, 2^Y, \{q_0\}, F, Down, Up, Side)$. A tree state synthesized for a node is the set of non-terminals from which the node can be derived. A forest state consists of the NFA states reached within the possible content models of the current level and can be computed as follows.

We start with the content model r_0, i.e. $q_0 = \{y_{0,0}\}$. We accept the top level sequence of nodes if it conforms to r_0, i.e. $F = \{q \mid q \cap F_0 \neq \emptyset\}$. The possible content models of a node are computed from the content models in which the node may occur: $Down(q, a) = \{y_{0,j} \mid y \in q,\ (y, x, y_1) \in \delta,\ x \to a\langle r_j \rangle\}$. When finishing a sequence of siblings we consider only the fulfilled content models in order to obtain the non-terminals from which the father node may be derived: $Up(q, a) = \{x \mid x \to a\langle r_j \rangle \text{ and } q \cap F_j \neq \emptyset\}$. The possible content models are updated after finishing visiting the next node in a sequence of siblings: $Side(q, p) = \{y_1 \mid y \in q, x \in p \text{ and } (y, x, y_1) \in \delta\}$. The resulting $A_{\vec{G}}$ is obviously deterministic, since it has one initial state and its transitions are functions rather than relations.

As an example, the run of $A_{\vec{G}}$ on our input document is presented in Appendix A.

6 Right-ignoring Queries

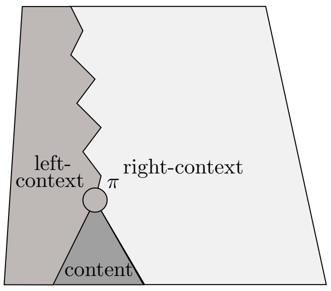


Fig. 4: The context and the content of a match

In this section we briefly present the ideas [9, 6] which allow the evaluation of a right-ignoring query $Q = (G, T)$ using the $A_{\vec{G}}$ LPA. Let us investigate what are the requirements for Q to be right-ignoring. Consider a match node π of Q as depicted in Fig. 4. Since the query is right-ignoring, all the nodes from the right-context are irrelevant for the decision as to whether π is a match. That is, π is a match however the right-siblings of π and of every ancestor of π might look like.

Let us consider that π is the k-th out of m siblings, with $m \geq k$. Since π is a match, according to the definition, there is a derivation labeling the sequence of siblings containing π with $x_1 \ldots x_k \ldots x_m$ and $x_k \in T$. There is thus a content model r_j s.t. $x_1 \ldots x_k \ldots x_m \in [\![r_j]\!]_{\mathcal{R}}$. The fact that the right siblings of π might be any trees implies that x_i must be able to derive any tree, i.e. $[\![R]\!]\, x_i = \mathcal{T}_X$ for all $i = k+1, \ldots, m$. Also, as the number of right-siblings might be arbitrary, all of the above must hold for all $m \in \mathbb{N}$ with $m \geq k$.

To ensure the above there must exist an NFA state $y_k \in Y_j$ reached after seeing the left siblings of π with $y_k \in F_j$ and s.t. for all $p \in \mathbb{N}$, there are $y_{k+1}, \ldots, y_p \in F_j$ with $(y_i, x_\top, y_{i+1}) \in \delta_j$ for $i = k+1, \ldots, p$, where $x_\top \in X$ and $[\![R]\!]\, x_\top = \mathcal{T}_\Sigma$. We call such a y_k a *right-ignoring NFA state*. The non-terminal $x_\top$ is to be seen as a wild-card non-terminal which can derive any tree and which is made available in any forest grammar. The necessity of the above follows from the fact that no other non-terminal x in the grammar can be s.t. $[\![R]\!]\, x = \mathcal{T}_\Sigma$, as in general the alphabet Σ is neither finite nor known in advance[1].

Given an NFA state y, we use the predicate $rightIgn(y)$ to test whether y is a right ignoring state. Testing whether $rightIgn(y)$ holds, can be done statically by checking in the NFA whether there are cycles visiting y and consisting only of $x_\top$ edges, needing thus time linear in the size of the NFA.

Similar considerations have to be made due to the right-ignorance for all the nodes lying on the path from the root to π. Therefore we need to consider all the non-terminals with which a derivation defining a match may label the nodes lying on the path from

[1] We do not consider optimizations possible when the schema of the XML data is available.

the match to the root. These are the so-called *match-relevant* non-terminals, defined by:

x is match-relevant iff
$x \in T$ or $x \to a\langle r_j\rangle, (y_1, x_1, y) \in \delta_j$ and x_1 is match-relevant

We call a query Q *right-ignoring* iff all $y \in Y$ with $(y_1, x, y) \in \delta$ and x match-relevant are right-ignoring. As presented above, testing whether a query is right ignoring can be done completely statically.

The right-ignorance of Q ensures thus that if the left-context of a match is fulfilled, then the right-context is also always satisfied. Hence, to check whether a node is a match, it suffices to look into the forest state in which $A_{\vec{G}}$ leaves a node, which synthesizes the information gained after visiting the left-context and the content of the node, depicted in dark grey in Fig. 4:

Theorem 2. *Let q_π be the forest state in which $A_{\vec{G}}$ leaves a node π. If Q is right-ignoring then $\pi \in \mathcal{M}_{Q,f}$ iff $y \in q_\pi$, $(y_1, x, y) \in \delta$ for some $y, y_1 \in Y$ and $x \in T$.*

Proof. The theorem is proven as Theorem 7.4 in [6].

To answer queries on XML streams without building the document tree in memory, it remains to show how a left-to-right pushdown automaton can be implemented in an event-based manner.

Event-driven Runs of Pushdown Forest Automata

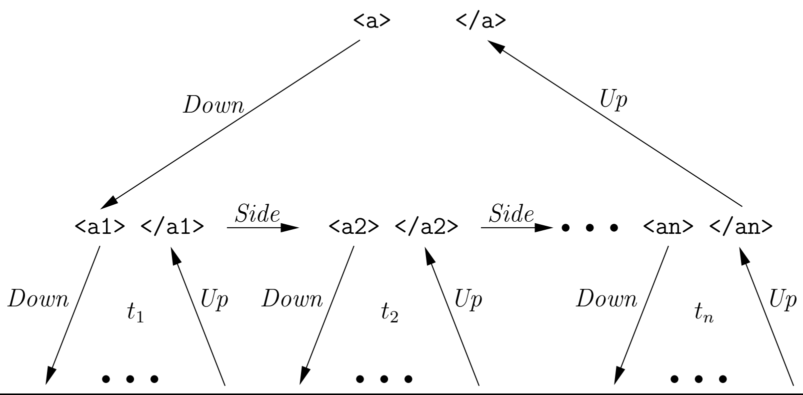


Fig. 5: Event-driven run of a pushdown forest automaton

Consider an LPA $A_{\vec{G}} = (2^X, 2^Y, \{q_0\}, F, Down, Up, Side)$ as defined in Section 5.1 and its processing model as depicted in Fig. 3 (on page 8). The order in which $A_{\vec{G}}$ visits the nodes of the input is the order of a depth-first, left-to-right search, which corresponds exactly to the document-order.

Compare Fig. 3 and Fig. 5. At every node π, $A_{\vec{G}}$ executes one $Down$ transition at the moment when it proceeds to the content of π and one Up followed by one $Side$ transitions at the moment when it finishes visiting the content of π. These moments correspond to the start and end tags, respectively, of the node π. The algorithm implementing the event-driven run of $A_{\vec{G}}$ is depicted in Listing 1.1.

We handle the following events:

1. `startDoc`, which is triggered before starting reading the stream;
2. `endDoc`, which is triggered after finishing reading the stream;
3. `enterNode`, which is triggered when a start-tag is read;
4. `leaveNode`, which is triggered when an end-tag is read.

```
1  Stack s;
2  ForestState q;
3
4  startDocHandler(){
5    $q$ := $q_0$;
6  }
7
8  enterNodeHandler(Label a){
9    s.push($q$);
10   $q$ := $Down(q, \mathrm{a})$ ;
11 }
12
13 leaveNodeHandler(Label a){
14   TreeState $p$ = $Up(q, \mathrm{a})$ ;
15   $q$ := $Side(\mathrm{s.pop()}, \mathrm{a})$ ;
16 }
17
18 endDocHandler(){
19   if $q \in F$ then output("Input accepted.")
20   else output("Input rejected.");
21 }
```

Listing 1.1. Skeleton for the event-driven run of a pushdown forest automaton

The stack declared in line 1 is needed in order to remember the forest states used for the traversal of the content of the elements opened and not yet closed. The variable q declared in line 2 stores the current forest state during the traversal of the document.

At the beginning, `startDocHandler` is called and it sets the current state to the initial state of the automaton (line 5). A start tag triggers a call of `enterNodeHandler` which remembers the current state on the stack (line 9) and updates the current state as result of executing the $Down$ transition (line 10). An end-tag triggers the corresponding Up transition (line 14), followed by the $Side$ transition which uses as forest state the last remembered state on the stack, i.e. the forest state before entering the element now ending (line 15).

The number of elements on the stack always equals the depth of the XML element currently handled. Hence the maximal height of the stack is the maximal depth of the handled XML document, which is in general rather small, even for very large documents.

Depending on the purpose of the pushdown automaton, other actions can be performed in the events handler. For the purpose of validation, it must be checked at the end of the document whether the current state is a final state (line 18).

For the purpose of answering right-ignoring queries it must be checked whether the forest state obtained after the side transition has the property stated in Proposition 2. Using the above presented implementation, $A_{\vec{G}}$ is thus able to answer right-ignoring queries on XML streams.

7 Arbitrary Queries

The previous section only shows how right-ignoring queries can be answered on XML streams. In this section we lift this limitation by showing how *arbitrary* queries can be answered on XML streams.

In the case of non right-ignoring queries, the decision as to whether a node is a match cannot be taken locally, i.e. at the time the node is left, because there is still match-relevant information in the part of the input not yet visited. The decision can only be taken after seeing all of the match-relevant information.

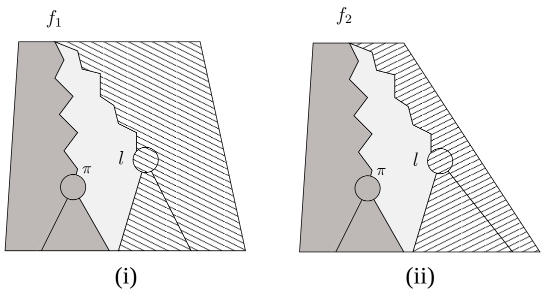


Fig. 6: Right completion of a forest

The general situation is depicted schematically in f_1 in Fig. 6 (i). The node π is a potential match considering its left context and its content (depicted in dark grey) which can be checked by the time the end-tag of the node is seen. The decision as to whether this is indeed a match node must be postponed until seeing the relevant part of the right context (depicted in light grey), which was empty in the particular case of right-ignoring queries. The location which must be reached in order to recognize π as a match is denoted as l. We call such a location l, *earliest detection location* of the match π, formally defined below.

Earliest Detection Locations

A forest f_2 is a *right-completion* of a forest f_1 at location $l \in L(f_1)$ iff f_1 and f_2 consists of the same events until l. (The tree representation of f_1 and f_2 are depicted in Fig. 6). Formally:

$$f_2 \in \mathcal{R}ightIgn_{f_1,l} \text{ iff } prec_{f_1}(l) = prec_{f_2}(l) \text{ and } lab(f_2[\pi']) = lab(f_1[\pi']) \\ \text{for all } \pi' \in prec_{f_1}(l).$$

with $prec_f(l)$ denoting the *preceding nodes* of a location $l \in L(f)$ in a forest f, defined as $prec_f(l) = \{\pi \mid \pi \in N(f), \pi < l\}$, where "$<$" denotes lexicographical comparison.

A location l is an *early detection location* of a match node π for a query Q in input f_1 iff $\pi \in \mathcal{M}_{Q,f_2}$ for all right-completions f_2 of f_1 at l.

A location l is the *earliest detection location* of a match node π iff l is the smallest early detection location of π in lexicographic order.

Example 6. Reconsider Example 1 and the accompanying input depicted in Fig. 1 (on page 2). Given the query `//a/b`, the earliest detection location of node 111 is 111. As for the query `//a[c]/b` the earliest detection location of node 111 is 112. Finally, for the query `/*[not(d)]//*`, there is no early detection location for any of the match nodes. This matches cannot be detected until the last location in the input has been reached.

7.1 Idea

We proceed now to the description of the computation performed by our algorithm for evaluating grammar queries on XML streams. This can be seen as a run of a pushdown automaton changing its state on every XML event.

For the purpose of evaluation we use the stack to remember the following information for a location l at some nesting level :

1. q, denoting the progress within the content models to be considered on the level containing l. This is exactly the forest state in which $A_{\vec{G}}$ (the DLPA accepting $\mathcal{L}_G$) reaches l;
2. ri, needed for the early detection of matches as presented below;
3. m, storing potential matches which might be confirmed on the current level, as well as the potential matches accumulated while traversing the current level up to l.

For 1, we remember the states of the finite automata corresponding to the content models which are reached considering the content seen so far on the current level. They are obtained as by performing the transitions of $A_{\vec{G}}$.

For 2, we need to know which of the content models considered on the current level occur in right-ignoring contexts. A content model for an element e occurs in a right ignoring context iff there is no content model of an enclosing element whose fulfillment depends on the right context of e. We call such content models *right-ignoring content models*.

For 3, we associate potential matches with NFA states from q. A potential match is associated with a state y at location l iff the match may be defined w.r.t. derivations in which the word of non-terminals on the current level is accepted by the NFA run reaching l in state y. The information m can be thus represented as a partial mapping from NFA states y to the corresponding potential matches $m(y)$.

Consider our query $Q = (G, T)$ with a forest grammar $G = (R, r_0)$. Let $r_1, \ldots, r_p$ be the regular expressions occurring on the right-hand sides in the productions R, where p is the number of productions. For $0 \leq j \leq p$, let $A_j = (Y_j, y_{0,j}, F_j, \delta_j)$ be the non-deterministic finite automaton (NFA) accepting the regular string language defined by r_j as obtained by the Berry-Sethi construction. By possibly renaming the NFA states we can always ensure that $Y_i \cap Y_j = \emptyset$for $i \neq j$. Let $Y = Y_0 \cup \ldots \cup Y_p$ and $\delta = \delta_0 \cup \ldots \cup \delta_p$.

Initial State Initially, we start with the NFA start state $y_{0,0}$ of the start content model r_0. The content model r_0 is right-ignoring as there are no enclosing elements. Also, there are no potential matches detected yet, thus the information initially remembered on the stack consists of:

$$q_0 = \{y_{0,0}\},\ ri_0 = \{r_0\},\ m_0 = \emptyset$$

Start-Tag Transitions On a start-tag event <a> at location l, new information (q_1, ri_1, m_1) is pushed on the stack, depending on the information in the current top of the stack (q, ri, m) as follows.

The possible content models of the current element are computed from the content models in which the element may occur (as in the case of a $Down$ transition in $A_{\vec{G}}$). Before seeing any of the children of the current element we are in the initial NFA state of these content models:

$$q_1 = \{y_{0,j} \mid y \in q,\ (y, x, y_1) \in \delta,\ x \rightarrow a\langle r_j \rangle\}$$

A content model r_j considered for the current element l is right-ignoring if (1) the surrounding content model r_k is right ignoring and (2) r_k is fulfilled independently of how the right siblings of l might look like. Condition (1) can be looked up in ri. To ensure (2) there must be an right-ignoring NFA state y_1 reachable after seeing the left siblings of l. Thus:

$$ri_1 = \{r_j \mid y \in q,\ (y, x, y_1) \in \delta_k,\ x \rightarrow a\langle r_j \rangle, r_k \in ri, rightIgn(y_1)\}$$

As for the potential matches which might be confirmed while visiting the content of l, these are the matches propagated so far for which the content of the current level is fulfilled whatever follows after l. We add l to these potential matches if it can be derived from a target non-terminal considering its left-context, and its right-context is irrelevant (that is l's confirmation as a match depends thus only on its content).

$$\begin{aligned} m_1(y_{0,j}) = & \bigcup\{m(y) \mid y \in q,\ (y, x, y_1) \in \delta_k,\ x \rightarrow a\langle r_j \rangle, r_k \in ri, rightIgn(y_1)\} \\ & \cup \\ & \{l \mid y \in q,\ (y, x, y_1) \in \delta_k,\ x \rightarrow a\langle r_j \rangle, r_k \in ri, rightIgn(y_1), x \in T\} \end{aligned} \tag{1}$$

End-Tag Transitions An end-tag event `</a>` at location $\pi i(n+1)$ signals that the processing of the sequence of children $\pi i1, \ldots, \pi in$ is completed and the computation has to return to the nesting level and advance over the father node πi. The top two elements on the stack at this moment: (q, ri, m) and (q_1, ri_1, m_1) store the state of the computation after seeing the children and the left siblings of πi, respectively. (q, ri, m) and (q_1, ri_1, m_1) are consumed from the stack and used to compute the new top of the stack (q_2, ri_2, m_2), reflecting the state after finishing seeing πi, as follows.

A content model r_j is fulfilled by the children of πi iff there is some $y_2 \in q \cap F_j$, i.e. a NFA final state for r_j is reached after traversing them. It follows that πi can be derived from symbols x for which there is a production $x \to a\langle r_j\rangle$. The advance in the content models on the level of πi, after seeing πi is obtained by considering NFA transitions with symbols x from which πi may be derived. This is completely similar to an Up transition followed by a $Side$ transition in $A_{\vec{G}}$ and is summarized by:

$$q_2 = \{y_1 \mid y_2 \in q \cap F_j, x \to a\langle r_j\rangle, y \in q_1, (y, x, y_1) \in \delta\}$$

As the set of right ignoring content models only depends on the surrounding content models, it remains unchanged for a whole nesting level, that is :

$$ri_2 = ri_1$$

As for the potential matches, we have to aggregate the potential matches from the left-context of πi with those from its content. More precisely, potential matches defined by an NFA run on the children level are joined with potential matches from the left context associated with NFA states which are reached in nesting NFA runs after seeing the father node. The father node, πi is added as a potential match if it can be derived from a target non-terminal:

$$\begin{aligned} m_2(y_1) = \{&m(y_2) \cup m_1(y) \mid y_2 \in q \cap F_j, x \to a\langle r_j\rangle, y \in q_1, (y, x, y_1) \in \delta\} \cup \\ \{&\pi i \mid y_2 \in q \cap F_j, x \to a\langle r_j\rangle, y \in q_1, (y, x, y_1) \in \delta, x \in T\} \end{aligned} \qquad (2)$$

7.2 Recognizing Matches

The construction above allows the location of matches as stated by the following theorem:

Theorem 3. *A location l is an early detection location for a match node π iff $\pi \in m(y)$, $r_j \in ri$ and $rightIgn(y)$ for some $y \in q \cap Y_j$ with (q, m, ri) being the top of the stack at event l.*

Proof. The complete proof is given in Appendix B. The idea of the proof is described next.

Let l be an early detection location for π. Let f_2 be a right-completion obtained from f_1 by adding on every level from the root to l an arbitrary number of right siblings $\star\langle\rangle$ (as depicted in Fig. 7), where $\star$ is a symbol not occurring in any of the rules in the grammar. By the definition of early detection locations there is a derivation f_2' of f_2 in which π is labeled x for some $x \in T$. Also, since $\star$ does not occur in any rule,

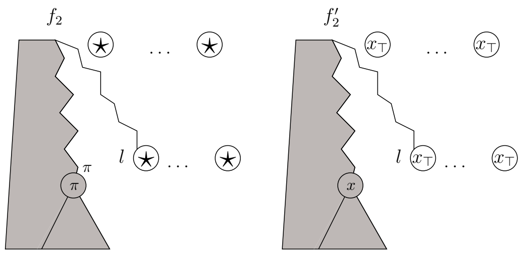


Fig. 7: Right completion at l and corresponding derivation

f_2' must label all the $\star$ nodes with $x_\top$. The y with the properties as required by this theorem is the NFA state in which the location l is reached within the NFA accepting run corresponding to f_2'.

Conversely, let (q, m, ri) be the top of the stack at event l and let $\pi \in m(y)$, $r_j \in ri$ and $rightIgn(y)$ for some $y \in q \cap Y_j$. From $\pi \in m(y)$ it follows that there is a relabeling of the nodes visited so far in which π is labeled with some $x \in T$ and which might be completed to a whole derivation according to the grammar G using $x_\top$ symbols for the not yet visited nodes. The existence of the completion on the current level follows from $rightIgn(y)$, while the existence of the completions on the enclosing levels is ensured by the condition $r_j \in ri$.

As locations are visited in lexicographic order, testing the condition in Theorem 3 ensures that every match π is detected when reaching its earliest detection location. This proves Theorem 1.

7.3 Implementation

The algorithm implementing the event-driven evaluation of the queries as above is given in Listing 1.2. We assume that `enterNodeHandler` and `leaveNodeHandler` receive as an argument, besides the label of the currently read node, also the currently reached location. For the case in which the current location is not provided by the event-based parser, note that it can be easily propagated along the event handlers in the internal parse-state. The algorithm basically follows the computation rules given above while sharing the commonalities in the rules for q, ri and m. As an abbreviation we use the operator $\oplus$ to add a new entry or update an existing set entry in a mapping m via set union.

Listing 1.2. Algorithm for event-driven query answering

```
Stack s;

enterNodeHandler(Location l, Label a){
  (q,ri,m) := s.top();
  $q_1$ := $ri_1$ := $m_1$ := $\emptyset$;

  forall $y \in q$ with $(y,x,y_1) \in \delta_k$ and $x \rightarrow a\langle r_j \rangle$
    $q_1$ := $q_1 \cup \{y_{0,j}\}$
    if $rightIgn(y_1)$ and $r_k \in ri$ then
10          $ri_1$ := $ri_1 \cup \{r_j\}$;
11          if $rightIgn(y_{0,j})$ then
12            reportMatches($m(y)$);
13            if $x \in T$ then reportMatches($\{l\}$) endif
14          else
15            $m_1$ := $m_1 \oplus \{y_{0,j} \mapsto m(y)\}$;
16            if $x \in T$ then $m_1$ := $m_1 \oplus \{y_{0,j} \mapsto \{l\}\}$ endif
17          endif
18        endif
19    endfor
20
21    s.push(($q_1$,$ri_1$,$m_1$));
22 }
23
24 leaveNodeHandler(Location $ln$, Label a){
25    ($q$,$ri$,$m$) := s.pop();
26    ($q_1$,$ri_1$,$m_1$) := s.pop();
27    $q_2$ := $m_2$ := $\emptyset$;
28    $ri_2$ := $ri_1$;
29
30    forall $y \in q_1$,$y_2 \in q$,$y_2 \in F_j$,$x \to a\langle r_j\rangle$ and $(y, x, y_1) \in \delta_k$
31      $q_2$ := $q_2 \oplus \{y_1\}$;
32      if $r_j \in ri$ then reportMatches($m(y_2)$)
33      else
34        $m_2$ := $m_2 \oplus \{y_1 \mapsto m(y_2)\}$;
35        $m_2$ := $m_2 \oplus \{y_1 \mapsto m(y)\}$;
36        if $x \in T$ then $m_2$ := $m_2 \oplus \{y_1 \mapsto \{l\}\}$ endif
37      endif
38    endfor
39
40    s.push(($q_2$,$ri_2$,$m_2$));
41 }
42
43 startDocHandler(){ s.push(($q_0$,$\{r_0\}$,$\emptyset$)); }
44
45 endDocHandler(){
46    ($q$,$ri$,$m$) := s.pop();
47
48    forall $y \in q \cap F_0$
49      reportMatches($m(y)$);
50    endfor
51 }
```

Matches are detected when their earliest detection location is reached, i.e. at the event-handler executed at the immediately preceding location. This might be the case either at start or at end tags. At start tags (line 12) we report potential matches for which we know that (a) the right siblings at the current level are irrelevant (condition $rightIgn(y_1)$ tested in line 9); (b) the right siblings of the ancestors are irrelevant (condition $r_k \in ri$ tested in line 9) and (c) the content is irrelevant (condition $rightIgn(y_{0,j})$ in line 11).

At end tags (line 32) we report potential matches for which the content was fulfilled (condition $y_2 \in F_j$ in line 30) and the upper-right context is irrelevant (condition $r_j \in ri$).

Note that there is no need to propagate a confirmed match π beyond its earliest detection location l where it is reported. (see tests in lines 11 and 32).

Also, potential matches are discarded implicitly precisely as soon as enough information is seen in order to reject them. Potential matches $m(y)$ at a location l are no longer propagated when y is not involved in the NFA transitions. This happens at end tag events if there is no transition (y, x, y_1) in any of the possible content models. Also at end tag events, potential matches in $m(y)$ are discarded if y is not a final state in any of the considered content-models on the finished level. Thereby matches are remembered only as long as the strictly necessary portion of the input has been seen in order to confirm them.

Finally, at the end of the input potential matches not yet confirmed and conforming to the top-level content model (condition $y \in q \cap F_0$ in line 48) are reported as matches in line 49.

7.4 Complexity

Let $|D|$ be the size of the input data, i.e. the number of nodes in it. The size of a query Q can be estimated as the number of NFA states $|Y|$ plus the number of non-terminals $|X|$. Let pot_{max} be the maximum number of potential match nodes at any given time during the traversal.

For every node in the input `enterNodeHandler` and `leaveNodeHandler` is called once. In `enterNodeHandler` at πi, the loop starting at line 7 is executed for every $y \in q$, for every outgoing NFA transition (y, x, y_1) and for all content models r_j for x. The size of q is in $O(|q_{max}|)$, where q_{max} is the forest state q with the maximum number of elements. Let cm_{max} be the maximum number of content models considered on a level and let out_{max} be the maximum number of outgoing NFA transitions from an NFA state. The loop is executed thus up to $|q_{max}| \cdot out_{max} \cdot cm_{max}$ times.

The set union in line 8 can be computed in time $O(|q_{max}|)$. The set union in line 10 needs time $O(cm_{max})$. Reporting the confirmed matches additionally requires time $O(pot_{max})$. Finally the set unions in lines 15 and 16 necessitate again $O(pot_{max})$ time. A call to `enterNodeHandler` amounts thus to $O(|q_{max}| \cdot out_{max} \cdot cm_{max} \cdot (|q_{max}| + cm_{max} + pot_{max}))$ time.

In `leaveNodeHandler`, the loop starting at line 30 is executed in the worst case, for every $y \in q_1$ and every $y_2 \in q$, i.e. up to $|q_{max}|^2$ times. The set union in line 31 is computed in time $O(|q_{max}|)$. Reporting the confirmed matches possibly adds $O(pot_{max})$ time. The set unions in lines 34, 35 and 36 need $O(pot_{max})$ time. A call to `leaveNodeHandler` amounts thus to $O(|q_{max}|^2 \cdot (|q_{max}| + pot_{max}))$ time.

As `leaveNodeHandler` and `enterNodeHandler` are called each once for every node, the overall time complexity of event driven evaluation of queries is thus in $O(|D| \cdot (|q_{max}| \cdot out_{max} \cdot cm_{max} \cdot (|q_{max}| + cm_{max} + pot_{max}) + |q_{max}|^2 \cdot (|q_{max}| + pot_{max})))$. The values of $|q_{max}|$, out_{max} and cm_{max} are bounded by values which do not depend on $|D|$. Experimental evidence show them to be small, and correspondingly the algorithm scales well with the size of the query as presented in the next section.

The worst complexity in the size of the document is obtained for $pot_{max} = |D|$, in the case where all the nodes are potential matches until the end of the document. In general, however, the number of potential matches is much less than the total number of nodes ($pot_{max} \ll |D|$) and can be assimilated with a constant. In this case we obtain a time linear in the size of the document, as suggested by our experimental results.

As for the space complexity, let d be the depth of the input document. During the scan of the document we store at each location the (q, ri, m) tuples for all ancestor locations up to the root, which correspond to the opened and not yet closed elements at the current location. For every level, q has up to $|q_{max}|$ elements, m stores up to $|q_{max}| \cdot pot_{max}$ locations and ri up to cm_{max} content models. As all these elements can be stored in constant space and the height of the stack is at most d, we obtain the worst case space complexity $O(d \cdot (|q_{max}| + |q_{max}| \cdot pot_{max} + cm_{max}))$. Most of the practical queries need only a small amount of memory, as the information relevant to whether a node is a match is typically located in the relative proximity of the node (that is pot_{max} is small).

7.5 Experimental Results

The algorithm presented here has been completely implemented in Fxgrep. Even though many proposals for evaluating XML queries on streams exist, there are surprisingly few tools publicly available. Furthermore, most of the proposals for which public implementations exist impose serious limitations on Core XPath (see Section 8 on related work). A more mature implementation we were able to experiment with was SPEX [16] which covers a large subset of XPath. As a reference for the in-memory DOM approach we used Xalan-Java 2.6.0 [17] one of the most popular XSLT processors, which also provides a command line XPath processor.

We used for our experiments the Protein Sequence Database [18], an XML document of over 700 MB size, containing around 25 million nodes with a maximal depth of 7 and an average depth of approximately 5. The experiments were performed on an Athlon XP 3000+ with 1GB of memory running under Linux (kernel version 2.6.8).

Even though the querying capabilities of Fxgrep go beyond those of Core XPath , for the comparison with other query tools for XML streams we had to limit ourselves to queries expressible in Core XPath. We used the following queries:

Q_1: As a representative of simple queries, specifying only the path to the matches, we chose Q_1 looking for protein entries containing a reference with an author element. The query is expressible with the XPath pattern `//ProteinEntry//refinfo/author`.

Q_2: A slightly more complex query, containing simple structure qualifiers is Q_2. The query locates authors of entries, the description of which contains the word “iron” and s.t. the year “2000” is mentioned among its references, i.e. the XPath pattern
```
//ProteinEntry[//description[contains(.,'iron')]]
  //refinfo[//year[contains(.,'2000')]]//author.
```

Q_3: Finally we use a more complex query Q_3 locating authors of proteins containing a reference to the year 2000, which are followed by two protein entries, the descriptions of which contain the word ”iron”.

	Fxgrep			Spex			Xalan		
	T	P	R	T	P	R	T	P	R
Q_1	29.8	8.7	3.4	9.1	4.1	2.2	15.8	2.5	6.3
Q_2	31.9	8.7	3.6	21.2	4.1	5.1	22.0	2.5	8.8
Q_3	33.5	8.7	3.8	50.9	4.1	12.4	178.8	2.5	71

Table 1: Evaluation times (in seconds) for increasingly complex queries

Table 1 presents the evaluation times for Q_1, Q_2 and Q_3 on a fragment of the database of 16 MB size. Absolute times are difficult to compare in the case of tools implemented in different programming languages. Fxgrep and its underlying parser Fxp are written in SML, while SPEX and Xalan are written in Java. The SML parser is significantly slower than the Java parsers in the case of large documents as in our experiment. That is, the events are delivered at different paces by the parsers used in the three applications. It is a situation similar to comparing absolute time measurements obtained using CPUs with different tact frequencies. A sensible way to account for this is to divide the absolute times by the frequency. Therefore, besides the total time in column T, we list the parsing time in column P, as well as the relative speed in column R obtained by dividing the total time by the parsing time.

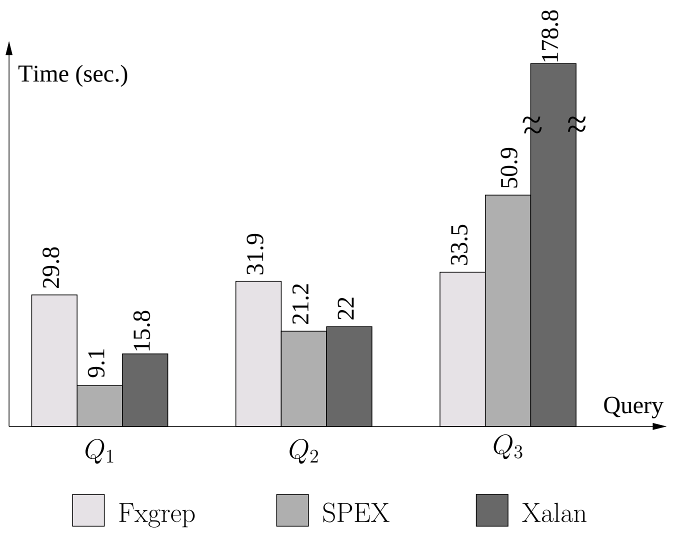


Fig. 8: Absolute evaluation times for increasingly complex queries

The absolute times are depicted in Fig. 8. One remarkable feature of Fxgrep is that the evaluation time does not significantly depend on the complexity of the query as opposed to the other tools. The reason is the expressiveness of the underlying grammar formalism in which adding supplementary contextual conditions does not significantly change the size of the underlying grammar. Interestingly enough, Fxgrep performs better even in absolute terms as the query complexity increases.

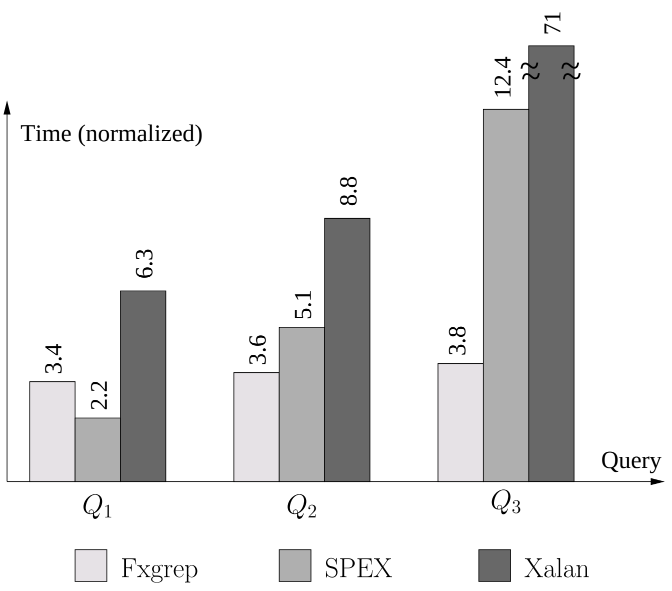


Fig. 9: Relative evaluation times for increasingly complex queries

The relative times are depicted in Fig. 9. The numbers denote how many times slower the query evaluation is as compared to the generation of the event stream. In Fxgrep around 30 percent of the evaluation time is needed for generating the stream of events. Fxgrep's throughput is comparable on average with the relative throughput of SPEX and better than what is achieved in Xalan.

	Fxgrep	Spex	Xalan
3 MB	9.8	10.2	10.1
16 MB	33.5	50.9	178.8
32 MB	61.8	96.8	614.6
159 MB	338	446	n/a

Table 2: Evaluation times (in seconds) for Q_3 for increasing document sizes

Table 2 presents the dependency of the evaluation times on the size of the document. We chose fragments of the input of increasing sizes and evaluated query Q_3. As depicted in Fig. 10 (containing the results for 3MB, 16MB and 32 MB size, respectively) the evaluation time increases linearly for Fxgrep and SPEX, as opposed to

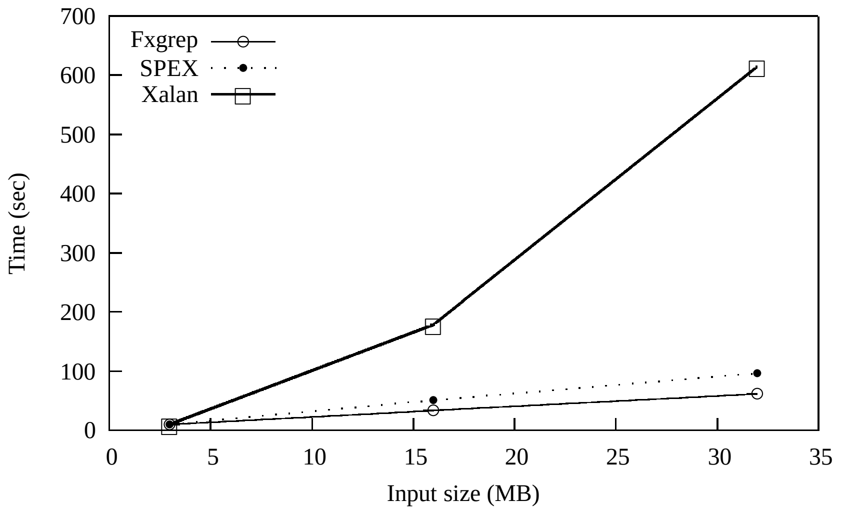


Fig. 10: Scalability with the input size

Xalan. This shows that the event-based processing mode of Fxgrep scales well with the input size, as presented in the complexity assessment presented in Section 7.4.

As for the memory usage, Fxgrep and SPEX need a constant space for all runs of up to 10 and 15 MB, respectively, including the SML runtime system and the Java Virtual Machine. As opposed to this, Xalan needs a multiple of the size of the handled input document. Even a memory space of 1 GB was not enough for Xalan in order to process the 159 MB large input.

8 Bibliographical Notes

A basic task in XML processing is XML validation. The problem of validating XML streams is addressed by Segoufin and Vianu in [19] and Chitic and Rosu in [20]. XML schema languages are basically regular forest languages[2], hence conformance to such a schema can be checked by a pushdown forest automaton. As presented in this chapter this can be performed efficiently on XML streams in the event-based manner.

Many research works deal with querying of XML streams. Most of them consider subsets of XPath. Some of them deal with XQuery, which in fact is more than a querying language as it allows the transformation of the input. In the following we are mainly interested in the querying capabilities of the considered languages.

Conventional attribute grammars (AG) and compositions thereof are proposed by Nakano and Nishimura in [22] as a means of specifying tree transformations. An algorithm is presented which allows an event-driven evaluation of attribute values. Specifying transformations, or in particular queries, using AG is however quite elaborate even

[2] The correlation between the most popular available schema languages and regular forest languages has been studied by Murata *et al.* [21].

for simple context-dependent queries and AG are restricted to use attributes of non-terminal symbols at most once in a rule. Also as no stack is used input trees have to be restricted to a maximum nesting depth.

More suited for XML are attribute grammars based on forest grammars as considered in XML Stream Attribute Grammars (XSAGs) [23] and TransformX [24][3]. A restricted form of attribute forest grammars is considered which allows the evaluation of attributes on XML streams. The attribute grammars have to be L-attributed, i.e. to allow their evaluation in a single pass in document-order. Another necessary restriction is that the regular expressions in productions are *unambiguous*, as in the case of DTDs. This ensures that every parsed element corresponds to exactly one symbol in the content model of the corresponding production, which allows the unambiguous specification and evaluation of attributes. While XSAGs are targeted at ensuring scalability and have the expressiveness of deterministic pushdown transducers, the TransformX AGs allow the specification of the attribution functions in a Turing-complete programming language (Java). In both cases, for the evaluation of the attribute grammars pushdown transducers are used. The pushdown transducers used in TransformX [24] validate the input according to the grammar in a similar manner to the pushdown forest automata. Additionally, a sequence of attribution functions is generated as specified by the attribute grammar. A second transducer uses this sequence and performs the specified computation. For the identification of the non-terminals from which nodes are derived in the (unique) parse tree, as needed for the evaluation of the AGs in [23, 24], pushdown forest automata can be used. The unambiguousness restriction of the attribute forest grammars allows one to proceed as in the case of right-ignoring queries presented in Section 6. That is, the non-terminal corresponding to the current node can be directly determined from the (single) NFA state in the current forest state, as it does not depend on the events after the current one.

A number of approaches handle the problem of querying XML streams in the context of selective dissemination of information (SDI), also known as XML message brokering [25–32]. In this scenario a large number of users subscribe to a dissemination system by specifying a query which acts like a filter for the documents of interest. Given an input document, the system simultaneously evaluates all user queries and distributes it to the users whose queries lead to at least one match. Strictly speaking, the queries are not answered. The documents which contain matches are dispatched but the location of the matches is not reported. XFilter [26] handle simple XPath patterns, i.e. without nested XPath patterns as filters. These can be expressed with regular expressions, hence they are implemented using finite string automata. YFilter [27] improves on XFilter by eliminating redundant processing by sharing common paths in expressions. More recently, in [28], the querying capabilities are extended to handle filters comparing attributes or text data of elements with constants and nested path expressions are allowed to occur basically only for the last location step. Green *et al.* [30] consider regular path expressions without filters. It is shown that a lazy construction of the DFA resulting from multiple XPath expressions can avoid the exponential blow-up in the number of states for a large number of queries. XPush [31] also handles nested path expressions and addresses the problem of sharing both path navigation and predi-

[3] In these works forest grammars are called extended regular tree grammars.

cate evaluation among multiple patterns. XTrie [32] considers a query language which allows the specification of nested path expressions and, besides, an order in which they are to be satisfied. Even though Fxgrep is not targeted at SDI, note that it basically exceeds the essential capabilities of all previously mentioned query languages.

There are a number of approaches in which queries on XML streams are answered by constructing a network of transducers [33, 34, 16, 35]. A query is there compiled into a number of interconnected transducers, each of them taking as input one or more streams and producing one or more output streams by possibly using a local buffer. The XML input is delivered to one start transducer and the matches are collected from one output transducer. The query language of XSM [33] handles only XPath patterns, without filters and deep matching (`//`), but allows instead value-based joins. XSQ [34] deals with XPath patterns in which at most one filter can be specified for a node and filters cannot occur inside another filter. The filters only allow the comparison of the text content of a child element or an attribute with a constant. SPEX [16] basically covers Core XPath. Each transducer in the network processes the input stream and transmits it augmented with computed information to its successors. The number of transducers is linear in the query size. The complexity of answering queries depends on whether filters are allowed and is polynomial in both the size of the query and of the input. XStreamQuery [35] is an XQuery engine based on a pipeline of SAX-like event handlers augmented with the possibility of returning feedback to the producer. The strengths of this construction are its simplicity and the ability to ignore irrelevant events as soon as possible. However, the approach only handles the child and descendant axes as yet.

FluXQuery [36] extends a subset of XQuery with constructs which guide an event-based processing of the queries using the DTD of the input. FluXQuery is used within the StreamGlobe project which is concerned with query evaluation on data streams in distributed, heterogeneous environments [37]. STX [38] is basically a restriction of the XSLT transformation language to what can be handled locally by considering only the visited part of the tree and selecting nodes from the remaining part of the tree. Sequential XPath [39] presents a quite restricted subset of XPath, handling only right-ignoring XPath patterns, which can be implemented without the need of any buffering. TurboXPath [40] introduces an algorithm for answering XPath queries containing both arithmetic and structural predicates and which is neither directly based on finite automata nor on transducer networks. The dynamic data structure WA (*work array*), used to match the document nodes has certain similarities with our construction. Entries are added in the WA upon each start-tag event for each sub-pattern to which the children must conform, which roughly correspond to a $Down$ transition of the LPA. Matches of the sub-patterns are detected upon end-tag events by AND-ing the fulfillment of the sub-patterns by the children, similarly to an Up transition. $Side$ transitions are not needed as the pattern language does not impose any order on the children nodes. In this perspective the context information is optimally used, as in our case, by a combination of top-down and bottom-up transitions. Recent work by Bar-Yossef *et al.* [41], indicates that the space requirement for the TurboXPath approach is near the theoretical optimum for XPath queries.

9 Conclusion

We have introduced a construction which allows the evaluation of grammar queries on unranked ordered trees in an event-based manner. The expressiveness of the queries exceeds the XML querying capabilities of languages for which streamed evaluation has been proposed yet. In particular it allows the evaluation Core XPath queries, while allowing to express also much more sophisticated contextual conditions. The construction allows to detect matches at the provably earliest time possible while scanning the input. We provide an algorithm which efficiently implements the construction and which has been used in the freely available Fxgrep language. The efficiency of our approach has been proved in practice as shown by experimental results.

References

1. : XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath (1999)
2. : XML Schema Language. http://www.w3.org/TR/xmlschema-0/ (2001)
3. : XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt (1999)
4. : XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/ (2005)
5. Gottlob, G., Koch, C., Pichler, R.: The Complexity of XPath Query Evaluation. In: Proceedings of the Eighteenth Symposium on Principles of Database Systems (PODS 2003). (2003)
6. Neumann, A.: Parsing and Querying XML Documents in SML. PhD thesis, University of Trier, Trier (2000)
7. Neumann, A., Berlea, A.: fxgrep 4.6.1. http://www2.informatik.tu-muenchen.de/˜berlea/Fxgrep/ (2005)
8. Murata, M.: Hedge-Automata: a Formal Model for XML Schemata. (Manuscript) (1999)
9. Neumann, A., Seidl, H.: Locating Matches of Tree Patterns in Forests. In Arvind, V., Ramamujan, R., eds.: Foundations of Software Technology and Theoretical Computer Science, (18th FST&TCS). Volume 1530 of Lecture Notes in Computer Science., Heidelberg, Springer (1998) 134–145
10. N.Klarlund, A.Moller, M.I.Schwartzbach.: DSD: A Schema Language for XML. In: ACM SIGSOFT Workshop on Formal Methods in Software Practice. (2000)
11. : RelaxNG Specification. http://www.relaxng.org/ (2001)
12. Frick, M., Grohe, M., Koch, C.: Query Evaluation on Compressed Trees. In: Proceedings of the 18th IEEE Symposium on Logic in Computer Science. (2003) 188–197
13. Neven, F., Schwentick, T.: Automata- and logic-based pattern languages for tree-structured data. In L. Bertossi, G. Katona, K.D.S., Thalheim, B., eds.: Semantics in Databases. Volume 2582 of Lecture Notes in Computer Science., Heidelberg, Springer (2003) 160–178
14. Gécseg, F., Steinby, M.: Tree Languages. In Rozenberg, G., Salomaa, A., eds.: Handbook of Formal Languages. Volume 3. Springer, Heidelberg (1997) 1–68
15. Berry, G., Sethi, R.: From Regular Expressions to Deterministic Automata. Theoretical Computer Science Journal **48** (1986) 117–126
16. Olteanu, D., Furche, T., Bry, F.: Evaluating Complex Queries against XML Streams with Polynomial Combined Complexity. In: Proc. of 21st Annual British National Conference on Databases (BNCOD21). (2004)
17. Project, A.X.: Xalan-java 2.6.0. Software Documentation (2005)
18. : Protein Information Ressource: The Protein Sequence Database. (http://pir.georgetown.edu/home.shtml)

19. Segoufin, L., Vianu, V.: Validating Streaming XML Documents. In: Symposium on Principles of Database Systems. (2002) 53–64
20. Chitic, C., Rosu, D.: On Validation of XML Streams using Finite State Machines. In: WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases, New York, NY, USA, ACM Press (2004) 85–90
21. Murata, M., Lee, D., Mani., M.: Taxonomy of XML Schema Languages Using Formal Language Theory. In: Extreme Markup Languages 2001, Montreal, Canada. (2001)
22. Nakano, K., Nishimura, S.: Deriving Event-Based Document Transformers from Tree-Based Specifications. In van den Brand, M., Parigot, D., eds.: Electronic Notes in Theoretical Computer Science. Volume 44., Elsevier Science Publishers (2001)
23. Koch, C., Scherzinger, S.: Attribute Grammars for Scalable Query Processing on XML Streams. In: Database Programming Languages (DBPL). (2003) 233–256
24. Scherzinger, S., Kemper, A.: Syntax-directed Transformations of XML Streams. In: Workshop on Programming Language Technologies for XML (PLAN-X) 2005. (2005)
25. Chean, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: a Scalable Continuous Query System for Internet Databases. In: Proceedings of the International Conference on Management of Data (SIGMOD 2000). (2000)
26. Altinel, M., Franklin, M.J.: Efficient Filtering of XML Documents for Selective Dissemination of Information. In: Procedings of the 28th International Conference on Very Large Data Bases (VLDB 2000). (2000)
27. Diao, Y., Fischer, P., Franklin, M.J., To, R.: YFilter: Efficient and Scalable Filtering of XML Documents. In: Proceedings of the International Conference on Data Engineering (ICDE 2002). (2002)
28. Diao, Y., Franklin, M.: YFilter: Query Processing for High-Volume XML Message Brokering. In: Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003). (2003)
29. Avila-Campillo, I., Green, T.J., Gupta, A., Suciu, D., Onizuka, M.: XMLTK: An XML Toolkit for Scalable XML Stream Processing. In: Workshop on Programming Language Technologies for XML (PLAN-X). (2002) PLAN-X 2002.
30. Green, T.J., Miklau, G., Onizuka, M., Suciu, D.: Processing XML Streams with Deterministic Automata. In: Proceedings of International Conference on Database Theory (ICDT 2003). (2003) 173–189
31. Gupta, A., Suciu, D.: Stream Processing of XPath Queries with Predicates. In: Proceedings of the International Conference on Management of Data(SIGMOD 2003). (2003)
32. Chan, C.Y., Felber, P., Garofalakis, M., Rastogi, R.: Efficient Filtering of XML Documents with XPath Expressions. In: Proceedings of the International Conference on Data Engineering (ICDE 2002). (2002)
33. Ludäscher, B., Mukhopadhyay, P., Papakonstantinou, Y.: A Transducer-Based XML Query Processor. In: Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002). (2002)
34. Peng, F., Chawathe, S.S.: XPath Queries on Streaming Data. In: Proceedings of the International Conference on Management of Data(SIGMOD 2003). (2003)
35. Fegaras, L.: The Joy of SAX. In: Proceedings of the First International Workshop on XQuery Implementation, Experience and Perspectives. (2004)
36. Koch, C., Scherzinger, S., Schweikardt, N., Stegmaier, B.: Schema-based Scheduling of Event-Processors and Buffer Minimization for Queries on Structured Data Streams. In: Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004). (2004)
37. Stegmaier, B., Kuntschke, R., Kemper, A.: StreamGlobe: Adaptive Query Processing and Optimization in Streaming P2P Environments. In: Proceedings of the International Workshop on Data Management for Sensor Networks. (2004) 88–97

38. Becker, O.: Transforming XML on the Fly. In: XML Europe 2003. (2003)
39. Desai, A.: Introduction to Sequential XPath. In: XML Conference 2001. (2001)
40. Josifovski, V., Fontoura, M., Barta, A.: Querying XML Streams. The VLDB Journal **14** (2005) 197–210
41. Bar-Yossef, Z., Fontoura, M., Josifovski, V.: On the Memory Requirements of XPath Evaluation over XML Streams. In: Proceedings of the 20th Symposium on Principles of Database Systems (PODS 2004). (2004)

A Example Run of $A_{\vec{G}}$

Example 7. The NFAs for the regular expressions occurring in our grammar G are depicted in Fig. 11. As input consider the XML document depicted in Fig. 1. The run of $A_{\vec{G}}$ on the tree representation of the input is shown in Fig. 12, where the sets containing x-s are tree states and the sets containing y-s are forest states. The order in which the tree and forest states are computed is denoted by the index at their right. Observe that the input tree, which is in the regular forest language specified by G, is accepted by $A_{\vec{G}}$ as it stops in the state $\{y_1\}$, which is a final state of the LPA.

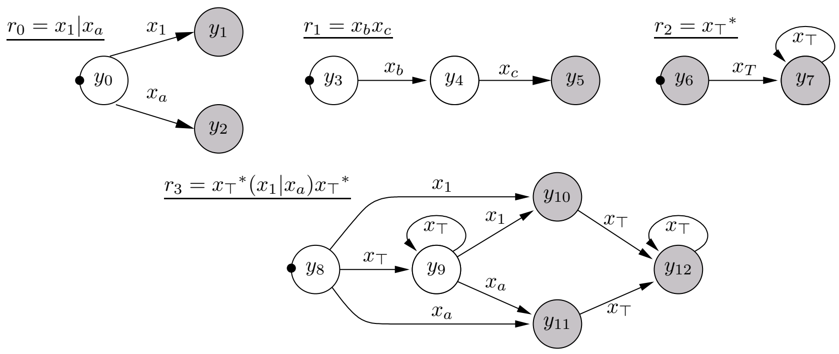


Fig. 11: NFAs obtained by Berry-Sethi construction for regular expressions in Example 1

B Proof of Theorem 3

In the following we use the notation from Section 7 where Theorem 3 was stated.

Alternative Definition of Matches

In order to proof Theorem 3 a more refined definition of matches is needed in which the NFA states reached while checking the content models of elements are given explicitely.

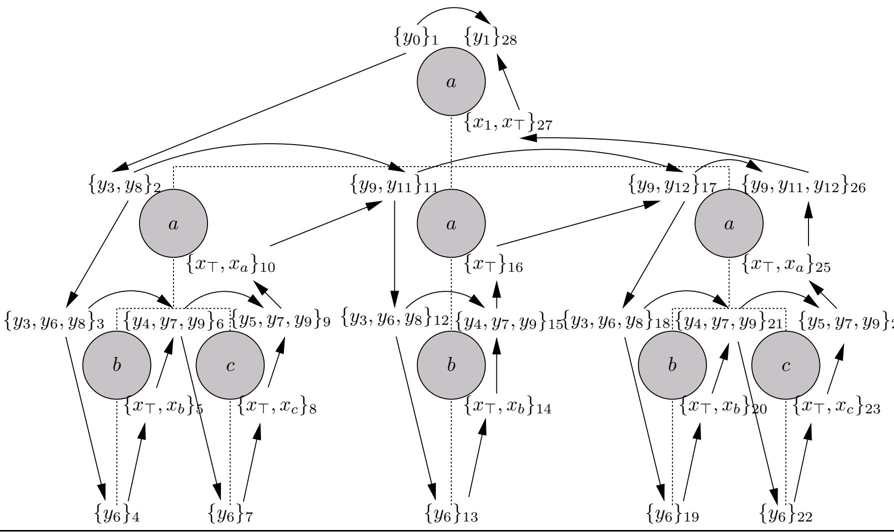


Fig. 12: The run of $A_G^{\vec{}}$ on the input tree

Let R be a set of forest grammar productions, r_0 be a regular expression over non-terminals and f an input forest. A *(non-deterministic, accepting) run* f_R over f for R and r_0, denoted $f_R \in \mathcal{R}uns_{r_0,f}$ is defined as follows:

$$y_0\langle f_1'\rangle \ \dots \ y_{n-1}\langle f_n'\rangle \ y_n\langle\rangle \in \mathcal{R}uns_{r_0,a_1\langle f_1\rangle \ \dots \ a_n\langle f_n\rangle} \text{ iff}$$
$$y_0 = y_{0,0}, y_n \in F_0, \text{ and}$$
$$(y_{i-1}, x_i, y_i) \in \delta_0, x_i \to a_i\langle r_i\rangle, f_i' \in \mathcal{R}uns_{r_i,f_i} \text{ for all } i = 1, \dots, n$$

$$y \in \mathcal{R}uns_{r_0,\varepsilon} \text{ iff } y = y_{0,0}, y \in F_0$$

An example run is given immediately below.

It is straightforward to see that a derivation f' with $(f, f') \in \mathcal{D}eriv_{r_0}$ (defined on page 5) exists iff a run $f_R \in \mathcal{R}uns_{r_0,f}$ exists.

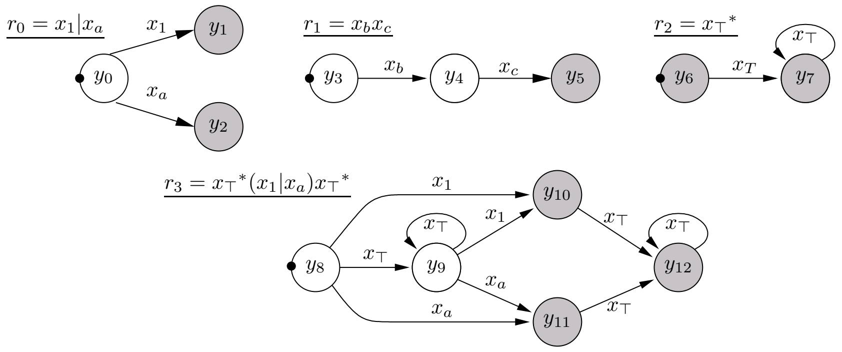
Example 8. Let $G = (R, r_0)$ with R being the set of rules as below:

$$\begin{array}{lll}
(1)\ x_\top \to a\langle x_\top^*\rangle & (4)\ x_1 \to a\langle x_\top^*(x_1|x_a)x_\top^*\rangle & (6)\ x_b \to b\langle x_\top^*\rangle \\
(2)\ x_\top \to b\langle x_\top^*\rangle & (5)\ x_a \to a\langle x_b x_c\rangle & (7)\ x_c \to c\langle x_\top^*\rangle \\
(3)\ x_\top \to c\langle x_\top^*\rangle & &
\end{array}$$

The NFAs for the regular expressions occurring in grammar G with the set are reproduced in Fig. 13.

Consider the input tree depicted t reproduced for convenience in Fig. 14 and one derivation of t' w.r.t. r_0 depicted in Fig. 15. A run corresponding to t' is depicted in Fig. 16 via dotted lines.

The derivation corresponding to a run can be obtained by taking the incoming transitions of the NFA states of the nodes which are not the first in their siblings sequence as



$r_0 = x_1|x_a$
$r_1 = x_b x_c$
$r_2 = x_\top{}^*$
$r_3 = x_\top{}^*(x_1|x_a)x_\top{}^*$
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9 y_{10} y_{11} y_{12}
x_1 x_a x_b x_c $x_\top$ x_T

Fig. 13: NFAs obtained by Berry-Sethi construction for regular expressions in Example 8

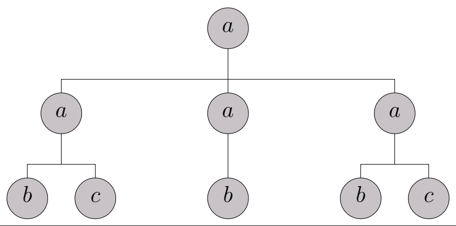


Fig. 14: Input tree t

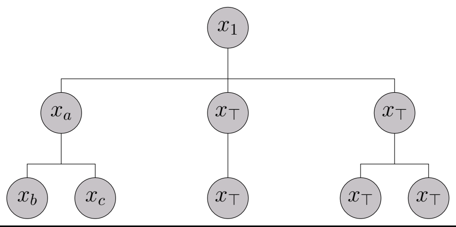
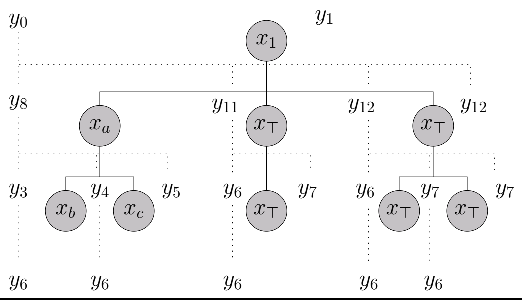


Fig. 15: Derivation t' of t w.r.t r_0



y_0 y_1
x_1
y_8 y_{11} y_{12} y_{12}
x_a $x_\top$ $x_\top$
y_3 y_4 y_5 y_6 y_7 y_6 y_7 y_7
x_b x_c $x_\top$ $x_\top$ $x_\top$
y_6 y_6 y_6 y_6 y_6

Fig. 16: Run corresponding to t'

one can see in Fig. 16. Formally, the following expresses the relation between *derivations* and *runs*:

$$\begin{array}{l}(f, f') \in \mathcal{D}eriv_r \\ \quad \text{iff } \exists f_R \in \mathcal{R}uns_{r,f} \text{ with } L(f') = N(f_R) \\ \qquad \text{and } lab(f'[\pi p]) = in(lab(f_R[\pi(p+1)])) \text{ for all } \pi p \in N(f').\end{array}$$

Let f be an input forest and $Q = ((R, r_0), T)$ a grammar query. Matches of Q, which were originally defined in terms of *derivations*, can be equivalently defined in terms of runs as it follows:

$$\pi p \in \mathcal{M}_{Q,f} \text{ iff } \exists f_R \in \mathcal{R}uns_{r_0,f} \text{ s.t. } in(lab(f_R[\pi(p+1)])) \in T.$$

Notations

Before proceeding with the proof we further introduce a couple of useful notations. The set of *matches defined by runs with label y at location l* is defined as:

$$\begin{array}{l}\pi p \in \mathcal{M}^{l,y}_{Q,f} \text{ iff } \exists f_R \in \mathcal{R}uns_{r_0,f} \text{ s.t. } in(lab(f_R[\pi(p+1)])) \in T \\ \qquad\qquad\qquad \text{and } lab(f_R[l]) = y\end{array}$$

The set of *l-right-ignoring matches defined by a run with label y at l* is defined as:

$$\pi \in \textit{ri-}\mathcal{M}^{l,y}_{Q,f} \text{ iff } \pi \in \mathcal{M}^{l,y}_{Q,f_2} \forall f_2 \in \mathcal{R}ightIgn_{f,l}$$

A node π' is a *πi-upper-right ignoring match defined by a run with label y at πi* iff for any right-completion f_2 at the parent of πi there is a run defining π' as a match of Q in f_2 which labels πi with y, formally:

$$\pi' \in \textit{uri-}\mathcal{M}^{\pi i,y}_{Q,f} \text{ iff } \pi' \in \mathcal{M}^{\pi i,y}_{Q,f_2} \forall f_2 \in \mathcal{R}ightIgn_{f,\pi}$$

Given a location πi and an NFA state y, a sequence of states is a *suffix run from y at πi* iff the last state in the sequence is a final state and the sequence of siblings to the right of πi allows to visit the sequence of states, formally:

$$\begin{array}{l}y_i, \ldots, y_n \in \mathcal{S}uf_{\pi i,y} \\ \text{ iff } (y_{k-1}, x_k, y_k) \in \delta_j, f_1[\pi k] \in [\![R]\!]\, x_k \text{ with } x_k = in(y_k), \forall k \in i, \ldots, n \text{ and} \\ \quad y_{i-1} = y, y_n \in F_j \text{ where } n = last_{f_1}(\pi)\end{array}$$

To denote the information on top of the stack at the some moment πi we write $\pi i.q$, $\pi i.m$ and $\pi i.ri$ in analogy to attributes of attribute grammars. Similarly to attribute grammars, these are computed by local rules as presented in Section 7.1.

Proof

Theorem 3 is a straightforward corollary of the following theorem:

Theorem 4. *The construction presented in Section 7.1 keeps the following invariant:*

$$\begin{array}{l}\pi' p \in \pi i.m(y), y \in \pi i.q \cap Y_j, \exists c \in \mathcal{S}uf_{\pi i,y} \textit{ and } r_j \in \pi i.ri \\ \textit{iff } \pi' p < \pi i \textit{ and } \pi' p \in \textit{uri-}\mathcal{M}^{\pi i,y}_{Q,f}\end{array} \tag{3}$$

Proof. We proof the two directions of Theorem 4 separately.

Left-to-right We show that (3) holds at all locations in the input by induction using the lexicographic order on locations.

Base case Initially, at location 1, $1.m(y) = \emptyset, \forall y \in 1.q$, thus $\pi' p \in 1.m(y)$ is false, and the left-to-right direction trivially holds.

Induction step Supposing that (3) holds at all locations up to some location l we show that it also holds at the immediately next location.

Start-tag transition We first show that if (3) holds at $\pi i \in N(f)$, so does it at $\pi i 1$.

Let $\pi' p \in \pi i 1.m(y_0)$, $y_0 \in Y_j$ and suppose $\exists c \in \mathcal{S}uf_{\pi i 1, y_0}$ and $r_j \in \pi i 1.ri$. Since $\pi' p \in \pi i 1.m(y_0)$, it follows by our construction (conform to (1) on page 15) that $\exists y \in \pi i.q$ with $(y, x, y') \in \delta_k$, $x \rightarrow a\langle r_j \rangle$, $rightIgn(y')$, $r_k \in \pi i.ri$ and either (i) $\pi' p \in \pi i.m(y)$ or (ii) $\pi' p = \pi i$ and $x \in T$.

In case (i) it follows from (3) at πi that $\pi' p < \pi i$ and $\pi' p \in$ *uri*-$\mathcal{M}_{Q,f}^{\pi i, y}$. Thus, obviously $\pi' p < \pi i < \pi i 1$ and it remains to show that $\pi' p \in$ *uri*-$\mathcal{M}_{Q,f}^{\pi i 1, y_0}$. This follows directly from $\pi' p \in$ *uri*-$\mathcal{M}_{Q,f}^{\pi i, y}$, $c \in \mathcal{S}uf_{\pi i 1, y_0}$ and $rightIgn(y')$ by grafting the run over the children of πi corresponding to c into the run corresponding to *uri*-$\mathcal{M}_{Q,f}^{\pi i, y}$.

In case (ii), $\pi' p = \pi i < \pi i 1$. The proof will use in this case the following lemma (also used later on):

Lemma 1.1: If there is a suffix run within a right ignoring content model, then, independently of what follows in the input after the enclosing element, there is a run over the input forest containing that suffix. Formally, if $y \in \pi i.q \cap Y_k$, $r_k \in \pi i.ri$ and $\exists c \in \mathcal{S}uf_{\pi i, y}$ then $\forall f_2 \in \mathcal{R}ightIgn_{f, \pi}$ $\exists f_R \in \mathcal{R}uns_{r_0, f_2}$ with $c = lab(f_R[\pi i]), \ldots, lab(f_R[\pi\ last_{f_R}(\pi)])$.

Proof. The proof is by straightforward induction on the locations in the input forest. The assertion trivially holds at location 1. For the induction step, let $\pi i \in N(f)$. We show that if the assertion holds at the location πi, it also holds at (i) $\pi i 1$ and (ii) at $\pi(i+1)$. In case (i) $\exists y \in \pi i.q$ with $(y, x, y') \in \delta_k$, $x \rightarrow a\langle r_j \rangle$, $rightIgn(y')$, $r_k \in \pi i.ri$. The required run is obtained by grafting the run over the children of πi corresponding to c into the run $y, y', \ldots$ corresponding to the induction hypothesis. In case (ii) the existence of the suffix run at $\pi(i+1)$ implies the existence of a run at πi and our conclusion follows by the induction hypothesis.

We continue now with the proof of Theorem 4.

Since $c \in \mathcal{S}uf_{\pi i 1, y_0}$ it follows (straightforwardly by definition) that $f[\pi i] \in [\![R]\!]\, x$. Given that $(y, x, y') \in \delta_k$ and $rightIgn(y')$ it follows that there is thus a suffix run $c \in \mathcal{S}uf_{\pi i, y}$ with $c = y, y', \ldots$. With $r_k \in \pi i.ri$ it follows by Lemma 1.1 that $\forall f_2 \in \mathcal{R}ightIgn_{f, \pi}$ $\exists f_R \in \mathcal{R}uns_{r_0, f_2}$ with $lab(f_R[\pi i]) = y'$. Since $rightIgn(y')$ it follows that $\exists f_R \in \mathcal{R}uns_{r_0, f_2}$ for any $f_2 \in \mathcal{R}ightIgn_{f, \pi i}$ and $lab(f_R[\pi i]) = y'$. With $c \in \mathcal{S}uf_{\pi i 1, y_0}$ it follows that $\exists f'_R \in \mathcal{R}uns_{r_0, f_2}$ (obtained by grafting the run over the children of πi corresponding to c into f_R) with $lab(f'_R[\pi i]) = y'$ and $lab(f'_R[\pi i 1]) = y_0$. Thus $\pi' p \in$ *uri*-$\mathcal{M}_{Q,f}^{\pi i 1, y_0}$.

End-tag transition We next show that if (3) holds at l $\forall l < \pi(i+1)$, so does it at $\pi(i+1)$.

Let $\pi'p \in \pi(i+1).m(y'')$, $y'' \in Y_k$ and suppose $\exists c \in \mathcal{S}uf_{\pi(i+1),y''}$ and $r_k \in \pi(i+1).ri$. Since $\pi'p \in \pi(i+1).m(y'')$, it follows by our construction (conform to (2) on page 16) that $\exists y \in \pi i.q$, $y' \in \pi i(n+1).q$ with $y' \in F_j$, $x \to a\langle r_j\rangle$, $(y, x, y'') \in \delta_k$ and either (i) $\pi'p \in \pi i.m(y)$, or (ii) $\pi'p \in \pi i(n+1).m(y')$, or (iii) $\pi'p = \pi i$ and $x \in T$.

In case (i) our conclusion follows directly from (3) at πi.

We continue with the cases (ii) and (iii). Given c and $r_k \in \pi(i+1).ri$ it follows by Lemma 1.1 that $\forall f_2 \in \mathcal{R}ightIgn_{f,\pi}$ $\exists f_R \in \mathcal{R}uns_{r_0,f_2}$ s.t. $lab(f_R[\pi(i+1)]) = y''$. Further we use the following lemma (also employed later on):

Lemma 1.2: If $y \in \pi n.q \cap \mathcal{F}_j$ then $\exists f_R \in \mathcal{R}uns_{r_j,f[\pi 1]\dots f[\pi n]}$ with $lab(f_R[\pi(n+1)]) = y$.

Proof. The proof is straightforward by induction on the depth of $f[\pi]$.

In case (ii), $\pi'p$ was found either before or while visiting the content of πi, that is either $\pi'p \leq \pi i$ or $\pi i < \pi'p < \pi(i+1)$, respectively. In the first case our conclusion follows directly from (3) at πi. In the second case $\pi'p < \pi(i+1)$ we further need the following lemma:

Lemma 1.3: If $y \in \pi n.q \cap \mathcal{F}_j$, $\pi'p \in \pi n.m(y)$ and $\pi 1 \leq \pi'p \leq \pi n$ then $\exists f_R \in \mathcal{R}uns_{r_j,f[\pi 1]\dots f[\pi n]}$ with $lab(f_R[\pi(n+1)]) = y$ and $in(lab(f_R[\pi'(p+1)])) \in T$ where $n = last_f(\pi)$.

Proof. The proof is by induction on the depth of $f[\pi]$. By Lemma 1.2 $\exists f'_R \in \mathcal{R}uns_{r_j,f[\pi 1]\dots f[\pi n]}$ with $lab(f'_R[\pi(n+1)]) = y$.

For depth 1 it directly follows that $\pi'p = \pi i$ for some $1 \leq i \leq n$ and $in(lab(f'_R[\pi'(p+1)])) \in T$. Therefore $f_R = f'_R$ is the sought after run. If the depth is more than 1, then either (A) $\pi'p = \pi i$ for some $1 \leq i \leq n$ and $in(lab(f_R[\pi'(p+1)])) \in T$ as above or (B) $\exists y' \in \pi in'.q \cap \mathcal{F}_k$, $\pi'p \in \pi in'.m(y)$ and $\pi i1 \leq \pi'p \leq \pi in'$ for some $1 \leq i \leq n$ and $n' = last_f(\pi i)$. In case (B) f_R in our conclusion can be constructed by grafting the run over the children of πi existent by the induction hypothesis into f'_R.

Our conclusion results now for the case (ii) $\pi i < \pi'p < \pi(i+1)$ by grafting the run corresponding to the children which defines the match (as implied by Lemma 1.3) into f_R.

In case (iii) $\pi'p = \pi i < \pi(i+1)$ and it remains to show that $\pi'p \in$ *uri*-$\mathcal{M}^{\pi(i+1),y''}_{Q,f}$. We have by Lemma 1.2 that $\exists f'_R \in \mathcal{R}uns_{r_j,f[\pi i1]\dots f[\pi in]}$ and $in(lab(f'_R[\pi'(p+1)])) \in T$. From f_R and f'_R it results (by grafting f'_R into f_R at π) that $\exists f''_R \in \mathcal{R}uns_{r_0,f_2}$ s.t. $in(lab(f''_R[\pi'(p+1)])) \in T$ and $lab(f''_R[\pi(i+1)]) = y''$, thus $\pi'p \in$ *uri*-$\mathcal{M}^{\pi(i+1),y''}_{Q,f}$.

Right-to-left Let $\pi'p < \pi i$ and $\pi'p \in$ *uri*-$\mathcal{M}^{\pi i,y}_{Q,f}$. Let f_2 be a right-completion of f at π obtained by adding on every level from the root to π inclusively an arbitrary number of right siblings $\star\langle\rangle$, as depicted in Fig. 17, where $\star$ is a symbol not occurring in any of

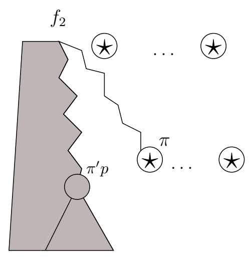


Fig. 17: Right completion of f at π

the rules in the grammar. Since $\pi' p \in$ *uri*-$\mathcal{M}^{\pi i,y}_{Q,f}$ it follows that $\exists f_R \in \mathcal{R}uns_{G,f_2}$ s.t. $in(lab(f_R[\pi'(p+1)])) \in T$ and $lab(f_R[\pi i]) = y$.

Also, since $\star$ does not occur in any rule f_R must label all the ancestors of the πi node with right-ignoring states, i.e. $rightIgn(lab(f_R[\pi_1(k+1)]))\forall \pi_1 k \in ancestors_f(\pi i)$, where *ancestors*$_f : N(f) \mapsto N(f)$ is defined as follows:

$$\begin{array}{lcl} ancestors_f(i) & = & \emptyset \\ ancestors_f(\pi i) & = & \{\pi\} \cup ancestors_f(\pi) \end{array}$$

It follows that $\exists f'_R \in \mathcal{R}uns_{G,f}$ s.t. $in(lab(f'_R[\pi'(p+1)])) \in T$ and $lab(f'_R[\pi i]) = y$. Suppose that $y \in Y_j$. Since y is part of a run (f'_R), it obviously holds that $\exists c \in \mathcal{S}uf_{\pi i,y}$.

Also, since $rightIgn(lab(f_R[\pi_1(k+1)]))\forall \pi_1 k \in ancestors_f(\pi i)$, we obtain by using the NFA transitions in f'_R at the corresponding steps in our construction that all content models of the elements enclosing πi are right ignoring, thus $r_j \in \pi i.ri$.

Given that $\pi' p < \pi i$ it follows that there is an ancestor of $\pi' p$ which is either (i) a sibling of an ancestor a of πi or (ii) an ancestor a of πi. In any case it follows by using the NFA transitions in f'_R at the corresponding steps in our construction that $\pi' p$ is propagated down at location a until πi, thus $\pi' p \in \pi i.m(y)$.

We have proven thus that $\pi' p \in \pi i.m(y)$, $y \in \pi i.q \cap Y_j$, $\exists c \in \mathcal{S}uf_{\pi i,y}$ and $r_j \in \pi i.ri$.

This completes the proof of Theorem 4.

Theorem 3 follows now directly from Theorem 4.