Theoretical Computer Science 578 (2015) 100–125

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Early nested word automata for XPath query answering on XML streams

Denis Debarbieux [a,c], Olivier Gauwin [d,e], Joachim Niehren [a,c], Tom Sebastian [b,c,*], Mohamed Zergaoui [b]

[a] Inria Lille, France
[b] Innovimax, France
[c] LIFL, France
[d] LaBRI, France
[e] University of Bordeaux, France

ARTICLE INFO

Article history:
Received 2 November 2013
Received in revised form 23 June 2014
Accepted 14 January 2015
Available online 20 January 2015

Keywords:
Automata
Logic
Trees
Nested words
Streams
Databases
Document processing
XML
XPath
XSLT
XQuery

ABSTRACT

Algorithms for answering XPath queries on XML streams have been studied intensively in the last decade. Nevertheless, there still exists no solution with high efficiency and large coverage. In this paper, we introduce early nested word automata in order to approximate earliest query answering algorithms for nested word automata. Our early query answering algorithm is based on stack-and-state sharing for running early nested word automata on all answer candidates with on-the-fly determinization. We prove tight upper complexity bounds on time and space consumption. We have implemented our algorithm in the QuiXPath system and show that it outperforms all previous tools in coverage on the XPathMark benchmark, while obtaining very high time and space efficiency and scaling to huge XML streams. Furthermore, it turns out that our early query answering algorithm is earliest in practice on most queries from the XPathMark benchmark.[1]

1. Introduction

XML is a major format for information exchange besides Json, also for Rdf linked open data and relational data. Therefore, complex event processing for XML streams has been studied for more than a decade [14,7,26,29,5,24,15,10,25]. Query answering for XPath is a basic algorithmic task on XML streams, since XPath is a language hosted by the W3C standards XSLT and XQuery.

Memory efficiency is essential for processing XML documents of several gigabytes that do not fit in main memory, while high time efficiency is even more critical in practice. Nevertheless, so far there exists no solution for XPath query answering on XML streams with high coverage and high efficiency. The best coverage on the usual XPathMark benchmark [8] is reached by Olteanu's Spex [26] with 22% of the use cases. The time efficiency of Spex, however, is only average, for instance

* Corresponding author.
E-mail address: tom.sebastian@inria.fr (T. Sebastian).

[1] Thanks to the QuiXProc project of Inria and Innovimax and the Cnrs Sosp project.

http://dx.doi.org/10.1016/j.tcs.2015.01.017
0304-3975/

compared to Gcx [29] which often runs in time close to the parsing time. We hope that this unsatisfactory situation can be resolved in the near future by pushing existing automata techniques forwards [14,24,10,25].

In contrast to sliding window techniques for monitoring continuous streams [3,19], the usual idea of answering queries on Xml streams is to buffer only *alive* candidates for query answers. These are stream elements which may be selected in some continuation of the stream and rejected in others. All space-optimal algorithms have to remove non-alive elements from the buffer, by outputting them or by discarding them. Unfortunately, this kind of *earliest query answering* is not feasible in polynomial time for XPath queries [6], as first shown by adapting counter examples from online verification [16]. A second argument is that deciding aliveness is more difficult than deciding XPath satisfiability [10], which is coNP-hard even for small fragments of XPath [4]. The situation is different for queries defined by deterministic *nested word automata* (Nwas) [1,2], for which earliest query answering is feasible with polynomial resources [24,11]. Many practical XPath queries (without aggregation, joins, and negation) can be compiled into small Nwas [10], while relying on non-determinism for modeling descendant and following axes. This, however, does not lead to an efficient streaming algorithm. The problem is that a cubic time precomputation in the size of the *deterministic* Nwa is needed for earliest query answering [11], and that the determinization of Nwas raises huge blow-ups in average (in contrast to finite automata).

Most existing algorithms for streaming XPath evaluation approximate earliest query answering, most prominently: Spex's algorithm on the basis of transducer networks [26], Saxon's streaming Xslt engine [15], and Gcx [29] which implements a fragment of XQuery. The recent XSeq tool [25], in contrast, restricts XPath queries by ruling out complex filters all over. In this way, node selection can always be decided with 0-delay [12] once having read the attributes of the node (which follow its opening event). Such queries are called begin-tag determined [5] if not relying on attributes. In this paper, we propose a new algorithm approximating earliest query answering for XPath queries that is based on Nwas. One objective is to improve on the previous approximations, in order to support earliest rejection for XPath queries with negation, such as for instance:

```
//book[not(pub/text()='Springer')][contains(text(),'Lille')]
```

When applied to an Xml document for an electronic library, as below, all books published from `Springer` can be rejected once its publisher was read:

```
<lib>...<book>...<pub> Springer </pub>
              ...<content>...Lille...</content>...</book>...</lib>
```

Spex, however, will check for all books from Springer whether they contain the string `Lille` and detect rejection only when the closing tag `</book>` is met. This requires unnecessary buffering space.

As a first contribution, we provide an approximation of the earliest query answering algorithm for queries defined by Nwa [11,24], while removing the assumption of determinism imposed there. The main idea to gain efficiency is that selection and rejection should depend only on the current state of an Nwa but not on its current stack. Therefore, we propose *early nested word automata* (eNwas) that are Nwas with two kinds of distinguished states: rejection states and selection states. Selection states are final and must always remain final, so that a nested word can be accepted, once one of its prefixes reaches a selection state. Symmetrically, rejection states can never reach a final state, so that a nested word can be rejected, once all non-blocking runs on a prefix reach a rejection state. We then present a new streaming algorithm for answering eNwa queries in an early manner. The basic idea is to run the eNwa for all possible candidates while determinizing on-the-fly, so that one can see easily whether all non-blocking runs of the nondeterministic automaton reach a rejection state, or whether one of them is selecting. The second idea is to share the stacks and states of runs of buffered candidates in the same state, so that the running time does not depend on the number of buffered candidates, but only on the number of states of the deterministic automaton discovered during the on-the-fly determinization. Our streaming algorithm with stack-and-state sharing for answering eNwas queries is original and nontrivial. It enables tight upper bounds for time and space complexity that we prove (Theorem 13).

As a second contribution, we show how to compile XPath expressions to small eNwa descriptors defining the same query. These descriptors allow to represent eNwas with large finite alphabets in a succinct manner, by replacing labels in eNwa rules by label descriptors. The label descriptor $\neg a$, for instance, stands for the set of all finitely many labels different from a. The target of our XPath-compiler is thus an eNwa descriptor. For instance, the eNwa descriptors that our XPath compiler obtains for the XPath expressions $P_n = \texttt{child::}a_1\texttt{/child::}a_2\texttt{/.../child::}a_n$ are of size $O(n)$, while the described eNwa is of size $O(n^2)$. The latter has n states each of which has n transitions, in order to accept children with all possible letters $a_1, \ldots, a_n$. We will prove a tight time bound for our compiler (Theorem 11). It implies the same bound on the size of the generated eNwa descriptors, and in particular that the eNwa descriptors for any XPath expressions without filters, unions, and with no other axes than *child* axes (such as P_n for instance) can be compiled in time $O(n)$. This improves on the previous compiler to dNwas from [10], which required time $O(n^4)$ for P_n. The main idea of the compiler is to adapt the previous translation to dNwas, so that it produces descriptors of eNwas while distinguishing selection and rejection states. We maintain pseudo-completeness (no run can ever block) as an invariant, so that we can compile negations efficiently in the deterministic case. Otherwise, we treat negation based on eNwa determinization after the instantiation of the eNwa descriptors, even though this is costly in theory and often unfeasible in practice. The XPath operators introducing

nondeterminism are recursive axes such as *descendant*, *following* and *following-sibling*, disjunctions of filters and unions of paths, and also expressions with *child* axes such as `child::a[following::b]` where the subexpression cannot be decided when closing the child. It should also be noticed that the compilation of forward axis requires a more complex treatment of stack symbols during the eNwa construction, which leads to a more tedious correctness statement for our compiler.

The third contribution is an implementation of our algorithms in the QuiXPath 1.3 system, which is freely available for testing on our online demo machine. It improves on all other tools in coverage with 37% of the XPathMark benchmark (the previous best is Spex with 22%), and also outperforms all of them in time efficiency with the exception of Gcx, which runs slightly quicker on few queries, and slightly slower on others. Our approximation of earliest query answering turns out to be tight in practice, in that all supported queries of the XPathMark are treated in an earliest manner. Our algorithms are not earliest on particular XPath queries with valid or unsatisfiable filters, of which there are two in the XPathMark, but these are not supported by QuiXPath 1.3 due to independent implementation limitations. It is shown in follow-up work [23] that our approximation is also tight in theory, in that it is exact for all positive XPath queries without valid or unsatisfiable subfilters. Note that there is still a gap between the tightness results in practice presented here, and those in the theory of the follow-up paper, in that some of the supported queries of the XPathMark use negation.

Summary of the contributions ranging from theory to practice

Early query answering algorithm for eNwa descriptor queries. We present a streaming algorithm with stack-and-state-sharing, that answers queries defined by eNwa descriptors on Xml streams, and prove tight upper complexity bounds for its space and time efficiency (Theorem 13).

Compiler from XPath to eNwa descriptors. We present a compiler from a fragment of XPath to eNwa descriptors that improves the previous compilers to dNwas in efficiency (Theorem 11).

QuiXPath 1.3 tool. We provide an implementation of our algorithms in the QuiXPath 1.3 tool and test it experimentally on the usual XPathMark benchmark. This shows the good coverage and high efficiency of QuiXPath 1.3 in practice, while being earliest in most cases.

The CIAA'2013 conference version [21] provided only a quick sketch of our results, which are now worked out with full proofs and extended experiments. The proposal of eNwa descriptors is new in the journal version, and also the efficiency theorem for the compiler from XPath to eNwa descriptors. Also the experimental section contains many new results. In particular, we introduce the concept of the *parsing free query evaluation time*, and use it to analyze the time efficiency of QuiXPath 1.3 in practice.

Outline Section 2 starts with preliminaries on nested word automata and earliest query answering. Section 3 introduces eNwas. Section 4 recalls the tree logic Fxp which abstracts from Forward XPath. Section 5 provides a compiler from Fxp to eNwa descriptors. Section 6 presents our new query answering algorithm for eNwas with stack-and-state sharing. Section 7 details our implementation and experimental results. Some details on Nwa determinization and the correctness proof of our compiler to eNwa descriptors are deferred to the appendix in [22] that is omitted in the TCS version.

2. Preliminaries

We recall the definitions of nested words and nested word automata and show how they can define node selection queries on data trees, and thus on Xml documents. We also recall what it means to answer node selection queries on data trees in streaming mode in an earliest manner.

Data trees, nested words, and tree suffixes Let Σ and Δ be two finite sets of labels and internal letters respectively. We assume that there is a set $\mathcal{L} \subseteq 2^{\Sigma}$ of label properties. A label property $L \subseteq \Sigma$ can be used for instance for distinguishing labels of different types of nodes.

A *data tree* over Σ and Δ is a finite ordered unranked tree, whose nodes are labeled by a tag in Σ or else they are leaves containing a string in Δ^*, i.e., any data tree t satisfies the abstract grammar $t ::= a(t_1, \ldots, t_n) \mid$ "w" where $a \in \Sigma$, $w \in \Delta^*$, $n \geq 0$, and $t_1, \ldots, t_n$ are data trees. A node π of a tree t is identified with the word of natural numbers that addresses the node when starting at the root. The empty word ϵ is identified with the root, and the word πi with the i-th child of node π. A *marked tree* is a pair (t, π) consisting of a tree t and a node π of t called the mark.

Any data tree t defines a relational structure with the following relation symbols:

$$\{ch, ch^+, ns, fo, fut\}$$
$$\cup\, \mathcal{L} \cup \{contains_w, equals_w, starts\text{-}with_w, ends\text{-}with_w \mid w \in \Delta^*\}$$

The domain is the set of all nodes of t. The relation symbols are interpreted as relations on this domain as follows: The binary relation ch^t relates a node to its children, the binary relation ns^t relates a node to its next sibling to the right, the descendant relation $(ch^+)^t = (ch^t)^+$ is the transitive closure of child relation, the following sibling relation $(ns^+)^t = (ns^t)^+$

is the transitive closure of the next sibling relation, the following relation $fo^t = ((ch^t)^*)^{-1} \circ (ns^t)^+ \circ (ch^t)^*$ relates a node to all its following nodes, and the future relation $fut^t = (ch^t)^* \vee fo^t$, which relates a node to all its future nodes. The label properties $L \in \mathcal{L}$ are used as unary relation symbols and interpreted as the set L^t of node of t whose label satisfies L. The unary relations $contains^t_w$, $equals^t_w$, $starts\text{-}with^t_w$ and $ends\text{-}with^t_w$ for words $w \in \Delta^*$ are satisfied by all nodes π of t whose data values satisfy the respective relation to w. Here, the data value of a node π is the concatenation of all strings contained in the subtree of t rooted by π.

A *nested word* is a word over three disjoint alphabets O, C, and Δ, with opening parenthesis $o \in O$, closing parenthesis $c \in C$ and internal letters in Δ. The positions of nested words are called *events*, of which there are three kinds. For every node of the tree there is an opening and a closing event, and for every letter of a data value, there is an internal event.

In order to linearize data trees over alphabets Σ and Δ into nested words, we need to fix functions $op : \Sigma \to O$ and $cl : \Sigma \to C$. If not stated otherwise, the functions will map letters a of Σ to opening and closing XML parentheses, i.e., $op(a) =$ `<a>` and $cl(a) =$ `</a>`. We can then linearize the data tree $l(b(p(\text{“}ACM\text{”}), c(\ldots)), \ldots)$ into the nested word `<l><b><p>ACM</p><c> ...</c></b>...</l>`. But sometimes, the label of a tag in a nested word may also be used to store information about its past. For instance, we could annotate each opening and closing event by its number as in `<(l,1)><(b,2)><(p,3)>ACM</(p,4)><(c,5)>`. This is supported by our definitions, since we can choose tuples $b = (a, i, j)$ as node labels in Σ and define $op(b) = \langle (a, i) \rangle$ and $cl(b) = \langle / (a, j) \rangle$. It should be noticed, however, that the resulting nested words are not well-formed XML documents, since corresponding opening and closing events are labeled differently. It should also be noted that every *node* of a data tree corresponds to a pair of matching opening and closing events. The matching can be established by a parser in streaming mode (a SAX parser in the case of XML).

Any marked data tree (t, π) can be linearized into the nested word, which is the suffix of the linearization of t starting at the opening event of π. More formally, we define for any tree $t = a(t_1, \ldots, t_n)$ that contains node π a tree suffix $suff(t, \pi)$ as follows:

$$suff\big(a(t_1, \ldots, t_n), \epsilon\big) = op(a) \cdot suff(t_1, \epsilon) \cdot \ldots \cdot suff(t_n, \epsilon) \cdot cl(a)$$
$$suff\big(a(t_1, \ldots, t_n), i\pi\big) = suff(t_i, \pi) \cdot suff(t_{i+1}, \epsilon) \cdot \ldots \cdot suff(t_n, \epsilon) \cdot cl(a)$$
$$suff(\text{“}w\text{”}, \epsilon) = w$$

The marked node of a tree suffix is the marked node of the underlying marked tree. Tree suffixes are well-balanced if and only if the marked node is the root. More generally, tree suffixes lack the opening events for all ancestors of the marked node. A *tree factor* is a prefix of a tree suffix. Note that tree suffixes are tree factors that are complete to the right in that they contain the closing event of the marked node.

XML streams The XML data model provides data trees with 6 different types of nodes: root, element (abbreviated as *el* in examples), attribute, text, comment, and processing-instruction. The label set Σ of XML trees is the set of all pairs of an XML tag and an XML node type. The label properties can test for a label $a = (tag, type) \in \Sigma$ whether *tag* is equal to a fixed constant, or whether *type* is equal to one of the types above. In practice, we add a third component to labels for the treatment of namespaces, but we will omit this here.

An XML stream contains a nested word that is the linearization of a marked data tree in XML format. Any XML data tree must satisfy the following typing restrictions: The root is the only node of type root and all nodes of type attribute, text, comment or processing-instruction are leafs. Furthermore, for any node, all its children of type attribute must precede all its other children. These XML specific typing restrictions are relevant for early query answering for XPATH. For instance, for the query `title[@lang='eng']`, any title node can be rejected, once the first element child was read and there was no `lang` attribute with value `eng` before. However, we will not have the space to discuss the implications of typing in detail.

Nested word automata A nested word automaton (NWA) is a pushdown automaton that runs on nested words [2]. The usage of the pushdown of an NWA is restricted: a single symbol is pushed at opening tags, a single symbol is popped at closing tags, and the pushdown remains unchanged when processing internal letters. More formally, a nested word automaton is a tuple $A = (O, C, \Delta, Q, Q_I, Q_F, \Gamma, R)$ where O, C, and Δ are the finite alphabets of nested words, Q a finite set of states with subsets $Q_I, Q_F \subseteq Q$ of initial and final states, Γ a finite set of stack symbols, and R is a set of transition rules of the following three types, where $q, q' \in Q$, $o \in O$, $c \in C$, and $d \in \Delta$:

(open) $q \xrightarrow{o:\gamma} q'$ can be applied in state q, when reading the opening parenthesis o. In this case, γ is pushed onto the stack and the state is changed to q'.

(close) $q \xrightarrow{c:\gamma} q'$ can be applied in state q when reading the closing parenthesis c with γ on top of the stack. Then, γ is popped and the state is changed to q'.

(internal) $q \xrightarrow{d} q'$ can be applied in state q when reading the internal letter d. One then moves to state q'.

A configuration of an NWA is a state-stack pair in $Q \times \Gamma^*$. Let $S \in \Gamma^n$ be a stack of depth $n \geq 0$ and s be a tree factor over Σ and Δ whose marked node is at depth n. An S-run on t must start in a configuration with stack S and some initial state, and then rewrite this configuration on all events of the tree factor according to some of the rules in R. An S-run on

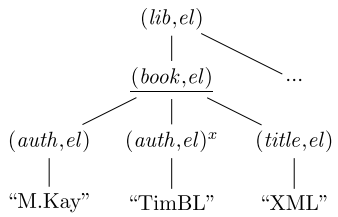

Fig. 1. An example of a data tree for a library, in which the first *book* element is marked, and therefore underlined. The second *auth* child of the marked node is annotated by variable *x*.

a tree suffix is called *successful* if it continues until the end while reaching some final state. Note that the stack will always be empty at the end of tree suffixes, given that the depth of S was equal to the depth of the start node and given the restriction on pushdowns of NWAs (often called "visible"). The *language* $\mathcal{L}_S(A)$ of an NWA A is the set of all tree suffixes over Σ and Δ that permit a successful S-run by A.

An NWA is called *deterministic* or a dNWA if it is deterministic as a pushdown automaton. In contrast to more general pushdown automata, NWAs can always be determinized [2], essentially, since they have the same expressiveness as bottom–up tree automata. For completeness the nontrivial determinization algorithm is given in the appendix of the long version [22]. In the worst case, the resulting deterministic automata may have $2^{|Q|^2}$ states. In experiments, we also observed huge size explosions in the average case. For example, for the XPath query `//a[following-sibling::b[.//c][./d]]/e` we obtain an NWA with 38 states and 7719 transitions, by instantiating the NWA descriptor from Section 5 (which has much fewer transition rules). We were not able to construct the corresponding dNWA even if restricted to accessible states only. The main limiting factor was memory. For the mentioned query we stopped the construction shortly, after having reached 5000 states with more than 20 million transitions, and swapping to the disk. Therefore, we will mostly rely on on-the-fly instantiation and determinization.

Automata queries We consider monadic (node selection) queries on marked trees. The XPATH query `following-sibling::b`, for instance, will select the nodes 2 and 3 of the marked tree $(a(b,b,b),1)$, since the b-node 1 of $a(b,b,b)$ has the next sibling 2 with label b, which in turn has the next sibling 3 again labeled by b. This is an example of a query that does not make much sense when started at the root of a tree, since the root never has any following sibling. For this reason, we apply queries to marked trees and let the query start at the marked node.

Definition 1. A monadic (forward) query over alphabets Σ and Δ is a function P that maps all marked trees (t,π) over Σ and Δ to some subset $P(t,\pi)$ of nodes π' of t opened later or equal to π, i.e., $P(t,\pi) \subseteq \{\pi' \mid fut^t(\pi,\pi')\}$.

These kinds of monadic queries cannot select nodes opened before π. We impose this restriction since we are interested in XPATH queries with forward axes only in the present paper. From a streaming perspective, this means that monadic queries of the above type concern only the tree suffix $suff(t,\pi)$ defined by a marked tree (t,π).

We will use NWAs to define monadic queries (as usual for showing that tree automata capture monadic second-order (MSO) queries). The idea is that an NWA should only test whether a candidate node is selected by the query on a given tree suffix, but not generate the candidate by itself. We fix a single variable x for annotation and set the label alphabet of such NWAs to $\{a, a^x \mid a \in \Sigma\}$. Letters a^x are called annotated (or "starred" in the terminology of [24]) while letters a are not. Then, a unique candidate node is assumed to be annotated on the input tree suffix by some external process. A monadic query P on marked trees can be defined by any NWA that recognizes the set of variants of $suff(t,\pi)$, in which the label of a single selected node in $P(t,\pi)$ is annotated by x.

Example An example for a marked data tree of a library is given in Fig. 1. There, the first *book* element is chosen as the marked node, and therefore underlined. The second *auth* child of the marked *book* element node is annotated by the variable x. The XPATH query

```
book[starts-with(title,'XML')]/auth
```

selects the annotated node when applied to the marked library (without the annotation), since the marked node is a *book* element, whose title starts with "XML" and since the annotated node is an *auth* child of the marked node.

Whether the above XPATH query applied to a marked tree can select a given node, can be verified as follows. First we annotate the given node with x as in Fig. 1, second we anonymize all symbols not occurring in the query by substitution with _, as in Fig. 2, so that the signatures become finite, and third we run the deterministic NWA in Fig. 3 on the marked, annotated, and anonymized tree in Fig. 2.

A successful run on the annotated marked tree from Fig. 2 is depicted in Fig. 4. This S-run starts at the marked *book* node, with the start stack $S = \gamma$ containing a single element, since the marked node has a single ancestor.

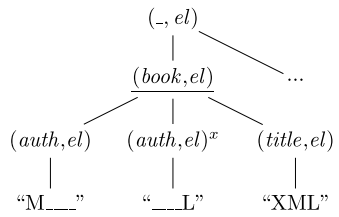

Fig. 2. A data tree with finite signature, obtained by anonymizing letters not occurring in `book[starts-with(title,'XML')]/auth`.

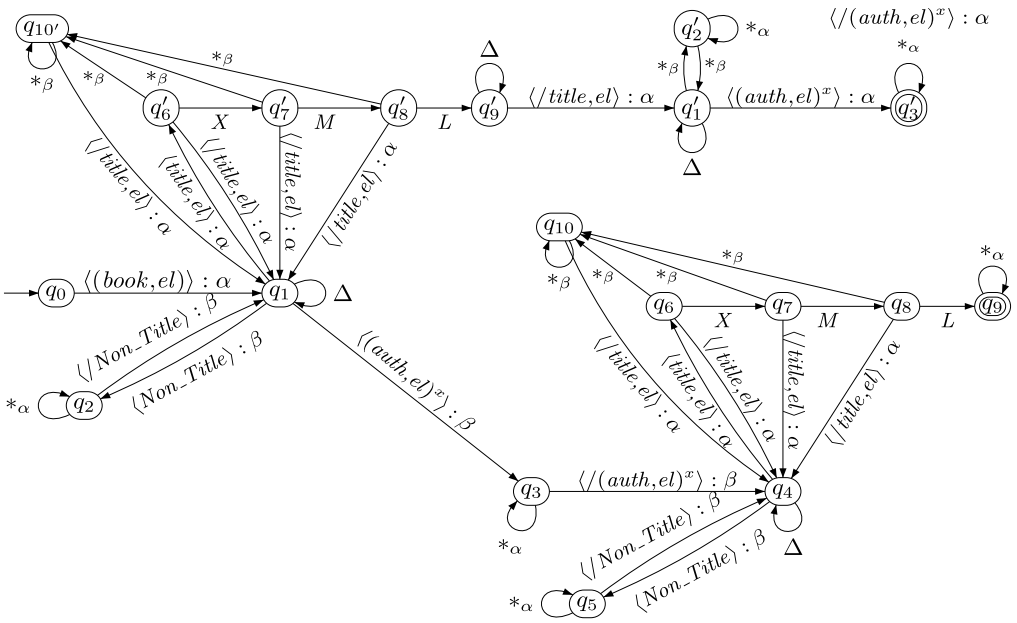

Fig. 3. An NWA for XPATH query `book[starts-with(title,'XML')]/auth` selecting all authors of all books of a library whose title starts with "XML". It can be run on any marked library while starting at its marked node, under the assumption that there is an arbitrary but unique node annotated by x. This is a candidate node for which selection is to be verified by the NWA. The finite signatures, obtained by anonymization of all letters that do not belong to the query to _, are $\Sigma = \{a, a^x \mid a \in \Sigma'\}$ where $\Sigma' = \{(book,el), (title,el), (auth,el), (\_,el)\}$ and $\Delta = \{L, M, X, \_\}$. As shortcuts, we use the sets of transitions $*_\alpha = \{\langle a\rangle : \alpha,\ \langle/a\rangle : \alpha \mid a \in \Sigma'\} \cup \Delta$ and $*_\beta$ defined analogously to $*_\alpha$, and the set of tags $Non\_Title = \Sigma' \setminus \{(title,el)\}$.

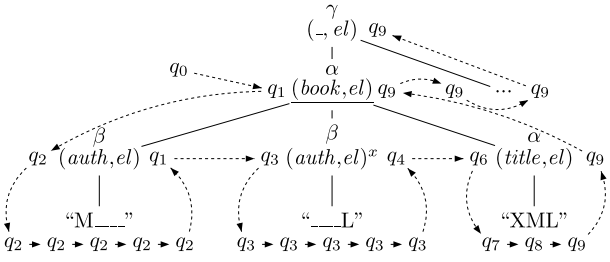

Fig. 4. A successful run of the NWA of Fig. 3 on the annotated library from Fig. 2, in which the first *book* element is chosen as the marked node. Here, the stack $S = \gamma$ provides a stack symbol for the unique ancestor of the marked node.

Earliest query answering Let P be a query, (t, π) a marked tree, π' a node such that $fut^t(\pi, \pi')$ is true, and e an event of $suff(t, \pi')$. We call π' *safe for selection at* e if $\pi' \in P(t', \pi)$ for every t' being a continuation of t beyond e, i.e., such that the prefixes of the linearizations of t and t' until event e are equal. We call π' *safe for rejection at* e if $\pi \notin P(t', \pi)$ for every t' such that t' is a possible continuation of t beyond e. We call π *alive at* e if it is neither safe for selection nor rejection at e. An earliest query answering (EQA) algorithm outputs selected nodes at the earliest event when they become safe for selection, and discards rejected nodes at the earliest event when they become safe for rejection. Indeed, an EQA algorithm buffers only alive nodes. The problem to decide the aliveness of a node is EXPTIME-hard for queries defined by NWAs [11]. For dNWAs it can be reduced to the reachability problem of pushdown machines which is in cubic time [10]. This, however, is too much in practice with NWAs of more than 50 states, 50 stack symbols, and $4 * 50^2 = 10\,000$ transition rules, so that the time costs are in the order of magnitude of $10\,000^3 = 10^{12}$.

3. Early nested word automata

We will introduce early NWAs for approximating earliest query answering for NWAs with high time efficiency. The idea is to avoid reachability problems of pushdown machines, by enriching NWAs with selection and rejection states,[2] so that aliveness can be approximated by inspecting states, independently of the stack. As we will see in Section 5, we can indeed distinguish appropriate selection and rejection states when compiling XPATH queries to ENWAs descriptors.

A subset Q' of states of an NWA A is called an *attractor* if any run of A that reaches a state of Q' can always be continued and must always stay in a state of Q'. Whether runs reaching Q' can always be continued is not that easy to decide syntactically, since it requires to decide accessibility questions for pushdown automata. In practice, however, we will only consider attractors that consist of a single state, which can loop in itself with all possible transitions.

Definition 2. An *early nested word automaton* (ENWA) is a triple $E = (A, Q_S, Q_R)$ where A is an NWA, Q_S is an attractor of A of final states called selection states, and Q_R is an attractor of non-final states called rejection states.

In the example NWA in Fig. 3, we can define $Q_S = \{q_9\}$ and $Q_R = \emptyset$. We could add a sink state to the automaton and to the set of rejection states. Also all selection states can be merged into a single state, and all rejection states can be deleted or merged into a single sink, if one wants to preserve pseudo-completeness (no run can ever block), as needed for efficient complementation in the deterministic case.

An ENWA defines the same language or query as the underlying NWA. Let us consider an ENWA E defining a monadic query and a data tree with some annotated node π. Clearly, whenever *some* run of E on this annotated tree reaches a selection state then π is safe for selection. By definition of attractors, this run can always be continued until the end of the stream while staying in selection states and thus in final states. In analogy, whenever *all* runs of E reach a rejection state, then π is safe for rejection, since none of the many possible runs can ever escape from the rejection states by definition of attractors, so none of them can be successful. For finding the first event, where all runs of E either reach a rejection state or block, it is advantageous to assume that the underlying NWA is deterministic. In this case, if some run reaches a rejection state or blocks, we can conclude that all of them do, as there is at most one run.

We call an ENWA deterministic if the underlying NWA is. We next argue that the determinization procedure for NWAs (see the appendix of the long version [22]) can be lifted to ENWAs. Let $E = (A, Q_S, Q_R)$ be an ENWA and A' the determinization of NWA A. The states of A' are sets of pairs of states of A', and not just sets of states of A, in contrast to more traditional classes of automata. We define the deterministic ENWA $E' = (A', Q_S', Q_R')$ such that Q_S' contains all sets of pairs of states of A, such that the second component of *some* pair belongs to Q_S, while Q_R' contains all sets of pairs of states of A, for which *all* pairs have their second component in Q_R.

Lemma 3. *Let $E = (A, Q_S, Q_R)$ be an ENWA and $E' = (A', Q_S', Q_R')$ be the deterministic ENWA obtained from the above determinization procedure. For any event e of the stream of a tree t, there exists a run of E going into Q_S at event e if and only if there is a run of E' going into Q_S' at e. Likewise all runs of E go into Q_R at event e iff all runs of E' go into Q_R' at e.*

This means that ENWA determinization preserves early selection and rejection. Intuitively, the reason is as follows. If we ignore the first components of the state pairs, then the run of automaton E' on t always reaches the set of states reached by E on t. Hence, whenever some run of E on t goes into a selection state q, the unique run of E' goes into a set of states that contains q and is thus selecting for E'. And whenever all runs of E on t reach a rejection state (or block), then the unique run of E' on t reaches the set of all these rejection states, which is rejecting for E'.

4. FXP logic

Rather than dealing with XPATH expressions directly, we first compile a fragment of XPATH into a variant of the hybrid temporal logic FXP [10]. Even though the translation from the fragment of XPATH without arithmetics, aggregations, joins, and positions to FXP is mainly straightforward, it leads to a great simplification, mainly due to the usage of variables for node selection and since it abstracts from the details of the XML data model.

The XPATH query `book[starts-with(title,'XML')]/auth`, for example, will be compiled to the following FXP formula with one free variable x:

$$el\&book\big(ch\big(el\&title(\textit{starts-with}_{XML})\big)\big) \wedge ch\big(el\&auth\&x(true)\big)$$

In this example, we rely on the label properties from the XML data model (while we may use others elsewhere): the label property *el* stands for all node labels (a, el) with $a \in \Sigma$, the label book for all $(book, T)$ where T is one of the XML types, and similarly *title* and *auth* test whether a node label has the respective XML tag. In addition, we use relations symbols *ch* and $\textit{starts-with}_{XML}$ for talking about the corresponding relations of data trees.

[2] The semantics of our selection states is identical with the semantics of final states in the acceptance condition for NWAs in [2], but should not be confused in general.

Formulas	F	::=	$F \wedge F \mid F \vee F \mid \neg F \mid \mathit{true}$
		|	$A(F) \mid B(F) \mid O_w$
Axes	A	::=	$\mathit{ch} \mid \mathit{ch}^+ \mid \mathit{ns}^+ \mid \mathit{fo}$
Label formulas	B	::=	$x \mid L \mid B\&B$
Comparisons	O	::=	$\mathit{equals} \mid \mathit{contains} \mid \mathit{starts\text{-}with} \mid \mathit{ends\text{-}with}$

Fig. 5. Abstract syntax of Fxp where $x \in \mathcal{V}$ is a variable, $L \in \mathcal{L}$ a label predicate on Σ, and $w \in \Delta^*$ a string data value.

$$\begin{array}{ll}
[\![F_1 \wedge F_2]\!]_{t,\pi,\mu} \Leftrightarrow [\![F_1]\!]_{t,\pi,\mu} \wedge [\![F_2]\!]_{t,\pi,\mu} & [\![B(F)]\!]_{t,\pi,\mu} \Leftrightarrow [\![B]\!]_{t,\pi',\mu} \wedge [\![F]\!]_{t,\pi',\mu} \\
[\![F_1 \vee F_2]\!]_{t,\pi,\mu} \Leftrightarrow [\![F_1]\!]_{t,\pi,\mu} \vee [\![F_2]\!]_{t,\pi,\mu} & [\![x]\!]_{t,\pi,\mu} \Leftrightarrow \pi = \mu(x) \\
[\![\neg F]\!]_{t,\pi,\mu} \Leftrightarrow \neg [\![F]\!]_{t,\pi,\mu} & [\![L]\!]_{t,\pi,\mu} \Leftrightarrow \pi \in L^t \\
[\![\mathit{true}]\!]_{t,\pi,\mu} \Leftrightarrow \mathit{true} & [\![B_1 \& B_2]\!]_{t,\pi,\mu} \Leftrightarrow [\![B_1]\!]_{t,\pi',\mu} \wedge [\![B_2]\!]_{t,\pi',\mu} \\
[\![A(F)]\!]_{t,\pi,\mu} \Leftrightarrow \exists \pi'.\ A^t(\pi,\pi') \wedge [\![F]\!]_{t,\pi',\mu} & [\![O_w]\!]_{t,\pi,\mu} \Leftrightarrow \pi \in O^t_w
\end{array}$$

Fig. 6. Semantics of Fxp formulas F for an Xml data tree t with node π and variable assignment μ to nodes of t.

The abstract syntax of Fxp formulas is given in Fig. 5. It is parameterized by a set of label properties $\mathcal{L}$ of some alphabet Σ, an alphabet Δ for strings, and a set of variables $\mathcal{V}$. Formulas are constructed from the single atomic formula *true*, the usual boolean operators, label formulas $B(F)$, where B imposes a set of label properties $L \in \mathcal{L}$ and a set of variables annotations $x \in \mathcal{V}$, and string comparisons equals_w, $\mathit{contains}_w$, $\mathit{starts\text{-}with}_w$, and $\mathit{ends\text{-}with}_w$ where $w \in \Delta^*$. Furthermore, there are navigation formulas with axis $A(F)$ where A is a relation symbol that will denote one of the binary relations of a data tree.

We define the *conjunction width* $w(F)$ as the number of conjunction operators $\wedge$ in F. The conjunction width of an Fxp formula will be relevant for the complexity analysis of the automata construction in Section 5. Note that conjunctions in label formulas $B\&B'$ are not counted. Furthermore, $B(F)$ can be rewritten equivalently with a conjunction $B \wedge F$, but this would increase the conjunction width.

The formal semantics of Fxp is defined in Fig. 6. A formula F is evaluated to a Boolean with respect to a given marked data tree (t, π) and a variable assignment μ that maps all variables of F to nodes of t opened later than π. The value of a formula F is the Boolean $[\![F]\!]_{t,\pi,\mu}$ defined in Fig. 6. As usual we will write $F \models F'$ if all models of F are also models of F', i.e. if $[\![F]\!]_{t,\pi,\mu}$ is true then also $[\![F']\!]_{t,\pi,\mu}$.

Any formula F with one free variable x defines a monadic query P on marked trees such that $P(t,\pi) = \{\mu(x) \mid [\![F]\!]_{t,\pi,\mu} = \mathit{true}\}$. For compiling XPath expressions to the fragment sketched above, we need only such Fxp formulas. For the general case, as treated in the remainder of the paper, formulas with n free variables will be essential. Since the general case does not raise any additional difficulties, we will not impose any restriction on the number of variables.

5. Compiler from FXP to early nested word automata

We have already seen in the previous examples of Nwas for XPath expressions[3] that the number of similar transitions for different labels may become huge. Furthermore, the alphabet of Nwas are exponential in the number of variables, which may become huge in the n-ary case. Therefore, we will compile Fxp formulas into compact descriptors of eNwas, in which labels are replaced by label descriptors.

5.1. ENWA descriptors

Let $\mathcal{V}_0 \subseteq \mathcal{V}$ be a finite set of variables and $\mathcal{L}_0 \subseteq \mathcal{L}$ a finite set of label predicates over Σ, for instance those used by some fixed Fxp formula.

A descriptor D_{lab} for a letter in Σ is an expression $E_1\& \ldots \&E_n$ where all E_i belong to $\{L, \neg L \mid L \in \mathcal{L}_0\}$ and $n \geq 0$. We write $D_{lab} = \mathit{true}$ if $n = 0$. We define the denotation $[\![D_{lab}]\!] \subseteq \Sigma$ as follows:

$$[\![L]\!] = L \quad \text{and} \quad [\![\neg L]\!] = \Sigma \setminus L \quad \text{and} \quad [\![E_1\& \ldots \&E_n]\!] = [\![E_1]\!] \cap \ldots \cap [\![E_n]\!].$$

A descriptor D_{var} for a subset of $\mathcal{V}_0$ is a pair (V, V') of subsets of $\mathcal{V}_0$. Its denotation is defined as follows:

$$[\![(V_1, V_2)]\!] = \left\{ V \;\middle|\; V_1 \subseteq V \subseteq (\mathcal{V} \setminus V_2) \right\}$$

We define the conjunction of two such descriptors $(V_1, V_1')\&(V_2, V_2')$ as $(V_1 \cup V_2, V_1' \cup V_2')$. Clearly, $[\![(V_1, V_1')\&(V_2, V_2')]\!]$ is equal to $[\![(V_1, V_1')]\!] \cap [\![(V_2, V_2')]\!]$.

A descriptor D_{tup} for a triple in $\Sigma \times 2^{\mathcal{V}_0} \times 2^{\mathcal{V}_0}$ is a triple (D_1, D_2, D_3) consisting of a descriptor D_1 for a letter in Σ, and descriptors D_2 and D_3 for subsets of $\mathcal{V}_0$. The denotation is $[\![(D_1, D_2, D_3)]\!] = [\![D_1]\!] \times [\![D_2]\!] \times [\![D_3]\!]$.

A descriptor of an eNwa is an eNwa itself, whose alphabets of opening and closing tags contain descriptors. An eNwa descriptor thus uses opening and closing tags $\langle D \rangle$ and $\langle /D \rangle$ where D is a descriptor of a label in $\Sigma \times 2^{\mathcal{V}_0} \times 2^{\mathcal{V}_0}$ rather

[3] See Fig. 3 and also the expressions P_n in the introduction.

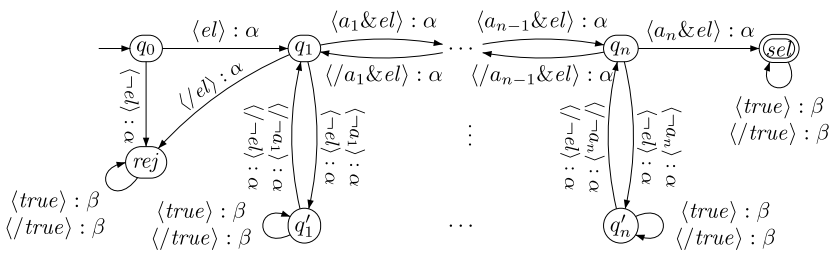

Fig. 7. A descriptor of an ENWA for XPATH filter `[child::`a_1`/child::`a_2`/.../child::`a_n`]` with selection state *sel* and rejection state *rej* (without transitions for texts and other XML types for simplicity).

than the label itself. The intuition for why we consider labels in $\Sigma \times 2^{\mathcal{V}_0} \times 2^{\mathcal{V}_0}$ is that we construct automata, who need to check a label in Σ, an annotation in $2^{\mathcal{V}_0}$, and who needs to check the possibility for annotations in $2^{\mathcal{V}_0}$ of future nodes of the data tree. This will become clearer in the following. The ENWA described is obtained from an ENWA descriptor by instantiating all occurrences of letters D in transition rules by all possible values of $[\![D]\!]$. We also need similar descriptors for letters in Δ but omit the details here. Note that it is possible that a rule is described twice by an ENWA descriptor, while the described automaton is still deterministic.

In Fig. 7, we illustrate the ENWA descriptor for XPATH filter `[child::`a_1`/.../child::`a_n`]`, i.e., the FXP expression $ch(el(a_1(ch(\ldots ch(el(a_n))))))$. Here we need conjunctive descriptors such as $a_i\&el$ for expressing simultaneous type and tag restrictions for the same node, and negative descriptors such as $\neg el$ and $\neg a_i$ for handling else cases. Thanks to the latter, the size of this ENWA descriptor is in $O(n)$, even though the size of the described ENWA is in $O(n^2)$, since for each of the n states q_i there are n outgoing edges, one for each a_i.

5.2. When variables must be bound

Consider the FXP formula $ch(x(true))$. The linearization of an annotated tree must be immediately rejected if x got assigned to the start node, since then x cannot be assigned to any child of the start node anymore. What is relevant here is which variables must be bound in order to make a subformula true.

Definition 4. Let F be an FXP formula and x a variable. We say F must bind x if $F \models fut(x)$ and that F cannot bind x if $F \models \neg fut(x)$.

In order to approximate $F \models fut(x)$ syntactically, as needed for our automata construction to define rejection states, we define the predicate $F \vdash fut(x)$ as the least binary relation such that:

1. $F_1 \wedge F_2 \vdash fut(x)$ if $F_1 \vdash fut(x)$ or $F_2 \vdash fut(x)$
2. $F_1 \vee F_2 \vdash fut(x)$ if $F_1 \vdash fut(x)$ and $F_2 \vdash fut(x)$
3. $A(F) \vdash fut(x)$ if $F \vdash fut(x)$
4. $L(F) \vdash fut(x)$ if $F \vdash fut(x)$
5. $x(F) \vdash fut(x)$ is true

Given a formula F one can compute in linear time the set $\{x \in \mathcal{V} \mid F \vdash fut(x)\}$. The following soundness lemma for syntactic binding is obvious.

Lemma 5. *For any* FXP *formula* F *and variable* x: $F \vdash fut(x) \Rightarrow F \models fut(x)$.

The converse, i.e. completeness, does not hold in general but still in many interesting cases (see [23]). The above soundness result will be sufficient to justify the correctness of our automata construction. In cases where completeness fails our ENWA may fail to be earliest.

5.3. Construction of ENWA descriptors

Let $\mathcal{V}_0 \subseteq \mathcal{V}$ be a finite subset of variables. For any tree t and mapping $\alpha : \mathcal{V}_0 \to \text{nodes}(t)$ we define the annotated tree $t * \alpha$, by replacing in t for any node π the label l of π by $(l, \mu^{-1}(\pi))$, i.e., by annotating the label of any node by the set of variables that are mapped to it.

Let F be an FXP formula with variables in $\mathcal{V}_0$ and $n \geq 0$. We define the language of tree suffixes of F with marks at tree depth n as follows:

$$\mathcal{L}_n(F) := \big\{ suff(t * \mu, \pi) \;\big|\; [\![F]\!]_{t,\pi,\mu} \text{ is true},\ \pi \text{ is a node of } t \text{ at depth } n \\ \mu(\mathcal{V}_0) \subseteq fut^t(\pi),\ dom(\mu) = \mathcal{V}_0 \big\}$$

This language contains all tree suffixes of annotated trees $t * \mu$, such that F evaluates to true for a node π of t at depth n, and variable assignment μ. For any finite set $\mathcal{E}$ of "external" stack symbols, that will be fixed by the context in which F will be used, we construct an eNwa $E_F(\mathcal{E})$, such that for all stacks $S \in \mathcal{E}^n$:

$$\mathcal{L}_S\big(E_F(\mathcal{E})\big) = \mathcal{L}_n(F)$$

Note that this equality will hold for all stacks S of height n, so that it is independent of the precise content of the stack.

We call a tree suffix s with opening and closing parentheses <l> and </l> where $l \in \Sigma \times 2^{\mathcal{V}_0}$ *canonical*, if each variable of $\mathcal{V}_0$ is annotated to exactly one node of s. Whether a tree suffix is canonical can be decided by a streaming algorithm, that runs the following deterministic finite word automaton C on the linearization of s. The state set of C is $2^{\mathcal{V}_0}$, the initial state is $\mathcal{V}_0$, and its final state is $\emptyset$. The rules are $V \xrightarrow{\langle (a,V')\rangle} V \setminus V'$ if $V' \subseteq V$, $V \xrightarrow{\langle /(a,V')\rangle} V$, and $V \xrightarrow{w} V$ for $a \in \Sigma$, $w \in \Delta^*$, and $V, V' \in 2^{\mathcal{V}_0}$. Note that C gets stuck at the earliest event, when the variable annotation gets in conflict with canonicity.

Given a tree suffix of a marked tree $s = \mathit{suff}(t * \alpha, \pi)$, let $\mathit{Can}(t * \alpha, \pi)$ be obtained from s by annotating all events by the subset of variables in $\mathcal{V}_0$ that were not bound in the past or at the current event by α, so that they can still be bound in the future. More formally $\mathit{Can}(t * \alpha, \pi)$ is the suffix at node π of the tree obtained from $t * \alpha$ by replacing for any node π' the label $(l, V) \in \Sigma \times 2^{\mathcal{V}_0}$ by (l, V, V'), where $V' = \{v \in \mathcal{V}_0 \mid \alpha(v) \in \mathit{fut}^t(\pi') \setminus \{\pi'\}\}$. Note that corresponding opening and closing events of $\mathit{Can}(t * \alpha, \pi)$ have the form $\langle (a, V, V_1)\rangle$ and $\langle /(a, V, V_2)\rangle$ where V_1 may be a proper subset of V_2, so the label of corresponding events need not be the same. Such nested words are still linearizations of trees but with the functions $\mathit{op}(a, V, V_1, V_2) = $ <(a,∇,∇1)> and $\mathit{cl}(a, V, V_1, V_2) = $ </(a,∇,∇2)>.

First we obtain $\mathit{Can}(s)$ by running the DFA C on the input tree suffix s, whose current state will always be the subset of variables in $\mathcal{V}_0$ that were not yet bound. Second, on the stream $\mathit{Can}(s)$, the automaton $E = E_F(\mathcal{E})$ will run the eNwa described by $D = D_F(\mathcal{E})$ constructed below. Whenever the run of C blocks, the eNwa described by D will go into a rejection state.

5.4. Construction of the eNwa descriptor $D_F(\mathcal{E})$

In order to compile an Fxp formula to the eNwa descriptor $D_F(\mathcal{E})$, we follow the same fundamental approach as for compiling tree logics such as Mso into tree automata, as used before in the context of XPath [11,24]. The most novel part here is the distinction of appropriate selection and rejection states, and the usage of label descriptors. It should also be noticed that our compiler will heavily rely on non-determinism in order to compile formulas with recursive axes such as $ch^+(F)$, $ns^+(F)$, or $fo(F)$, and disjunctions $F_1 \vee F_2$. However, we will try to preserve determinism of the described eNwa as much as possible, so that we can compile many formulas $\neg F$ without having to determinize the described eNwa for F. The compiler will rely on the so-called head $h(D)$, that contains the subset of all opening rules of D that start from an initial state, and the subset of closing rules of D that end in a selection or rejection state with a stack symbol pushed by such an opening rule. Without disjunctions and conjunctions, the heads will always remain of constant size. This is the reason why formulas such as $ch(a(ch(\ldots ch(a)\ldots)))$ can be compiled to eNwa descriptors of linear size in linear time. When adding disjunction but no conjunction, the heads are still of amortized constant size, as we will show later on. This is the reason why we will need an amortized size and time analysis.

The construction of $D_F(\mathcal{E})$ is by induction on the structure of F as follows.

Case $F = F_1 \wedge F_2$**.** Let $D_{F_i}(\mathcal{E}) = (A_i, Q_{Si}, Q_{Ri})$ be the eNwa descriptors for F_i where A_i has state set Q_i and $i \in \{1, 2\}$. We define the eNwa descriptor $D = (A, Q_S, Q_R)$ such that A is the product of A_1 and A_2. We choose $Q_S = Q_{S1} \times Q_{S2}$, since a node is safe for selection for $F_1 \wedge F_2$ iff it is safe for selection for both F_1 and F_2. For rejection states we take $Q_R = (Q_{R1} \times Q_2) \cup (Q_1 \times Q_{R2})$, which may lead to a proper approximation of earliest query answering. Furthermore, note that a large number of conjunctions may lead to an exponential blow-up of the number of states.

Case $F = F_1 \vee F_2$**.** Let $D_{F_i}(\mathcal{E}) = (A_i, Q_{Si}, Q_{Ri})$ be the eNwa descriptor for the subexpressions where A_i has state set Q_i and $i \in \{1, 2\}$. We define $D = (A, Q_S, Q_R)$ such that A is the union of A_1 and A_2, which introduces nondeterminism in contrast to products. Furthermore, we define $Q_S = Q_{S1} \cup Q_{S2}$ and $Q_R = Q_{R1} \cup Q_{R2}$.

Case $F = \neg F_1$**.** If $D_1 = D_{F_1}(\mathcal{E})$ describes a deterministic eNwa, then we obtain D by flipping selection and rejection states of D_1. This is correct, since we maintain pseudo-completeness (i.e. no run can ever block [9], instead it goes into a rejection state) of the described eNwa as an invariant. There is no approximation here, since a node is safe for selection for $\neg F_1$ iff it is safe for rejection for F_1, and conversely. Otherwise, we first compute the eNwa described by D and determinize it in a first step, which is also free of approximation by Lemma 3, and second apply the previous construction.

Case $F = ch(F_1)$ **where** F_1 **contains neither recursive axes nor disjunctions.** Since F_1 neither contains recursive axes nor disjunctions, $D_1 = D_{F_1}(\mathcal{E})$ describes a deterministic eNwa. Furthermore, selection and rejection can be decided no later than when closing the start node. In this case, we can construct D such that it runs D_1 on all children of the start node, deterministically one by one, until a selection state is reached or the start node was closed. This can be done by adding 2 states and stack symbols to D_1 only, based on a recomputation trick for the stack symbol pushed at the start node by D_1 [9].

The eNwa described by $D = D(\mathcal{E})$ first reads the opening event of the start node, say $\langle (a, V', V'')\rangle$ and goes into a rejection state if there exists $x \in V'$ such that $F_1 \vdash \mathit{fut}(x)$ or if $V' \cap V'' \neq \emptyset$. In the latter case, C will block. Otherwise,

D behaves as D_1 but stacks the Boolean 0 (stating that no previous child was tested successfully) instead of what D_1 would stack. The missing stack symbol will then be recomputed when closing the root of the child. In order to construct D alike, one needs to iterate over the head of D_1, but does not have to touch the rest of D_1. This can be done in time $|h(D_1)|$. The selection states of D are those of D_1, while D introduces a new rejection state. Whenever a run of D_1 goes into a rejection state of D_1 (which is not a rejection state of D), then automaton D may stop any further child tests if x occurs in a set of γ and $F_1 \vdash \mathit{fut}(x)$, in which case D goes into its rejection state. If D reaches the closing event of the start node (that is when the Boolean 0 is popped), then D also goes into the rejection state (this is correct again since F_1 contains no following axes, so that rejection of $ch(F_1)$ can be decided there). Therefore, no external stack symbol from $\mathcal{E}$ will ever be read by D (they may appear only after selection or rejection).

Case $F = A(F_1)$ **where** $A \in \{ch^+, ns^+, fo\}$ **or** $F = ch(F_1)$ **where** F_1 **contains recursive axes or disjunctions.** Let $D_1 = D_{F_1}(\mathcal{E} \cup \Gamma_{new})$ be the ENWA descriptor for F_1, where Γ_{new} is the set of stack symbols that $D = D_F(\mathcal{E})$ introduces. A nondeterministic ENWA is described by D, which guesses an A-successor of the start node and runs D_1 starting there. There is a main run of D, which for all non-A successors of the start node goes into a skip state q_{skip}, and which goes into a state q that generates tests for F_1 for potential following A-successors: For $A = ch$ the main run goes to state q such that the next potential opening event is a child of the start node, and it stays in q_{skip} for the subtrees of children of the start node. For $A = ch^+$ the main run goes and stays in state q for all descendants of the start node (no skip state q_{skip} needed). For $A = ns^+$ the main run goes to state q for all siblings to the right of start node, and it stays in q_{skip} for the subtree of the start node and any of the subtrees of its siblings to the right. For $A = fo$ the main run skips the subtree of the start node and after stays in q for the rest of the stream, which is done via closing rules $q \xrightarrow{(true):\beta} q$ for all $\beta \in \mathcal{E}$. The main run of D starts tests for F_1 by adding a rule $q \xrightarrow{\langle a\rangle:\alpha} p$ for each initial rule $i \xrightarrow{\langle a\rangle:\alpha} p$ of D' with $i \in Q_I$ of D'. This main run will continue until:

- either x occurs in a set of γ and $F_1 \vdash \mathit{fut}(x)$, in which case D goes into a rejection state, or
- the closing event of the start node of P arrives (for $A = ch$ and $A = ch^+$),
- the closing event of the parent of the start node of P arrives (for $A = ns^+$),
- the end of the stream arrives (for $A = fo$), indicated by a unique top most stack symbol.

Automaton descriptor D inherits its selection and rejection states from D_1. Note that here it matters again that a candidate can be rejected only if *all* runs of D on this candidate go into a rejection state.

Case $F = B(F_1)$**.** Without loss of generality, let $B = L_1\&\ldots\&L_m\&x_1\&\ldots\&x_n$ where $n+m \geq 1$ such that $L_i \in \mathcal{L}$ and $x_i \in \mathcal{V}_0$. Let $D_{F_1}(\mathcal{E})$ be the ENWA descriptor for F_1. We build D from D_1, by restricting all label descriptors of initial rules of D_1 by B: Descriptors $(E_1\&\ldots\&E_k, (V_1, V_2), (V_1', V_2'))$ are replaced by $(E_1\&\ldots\&E_k\&L_1\&\ldots\&L_m, (V_1 \cup \{x_1, \ldots, x_n\}, V_2), (V_1', V_2'))$. Furthermore, in order to obtain pseudo-completeness, we add one new rule to D for each conjunct of B and each initial state of D_1 going into a rejection state of D_1: For $1 \leq i \leq m$ the new initial rule has label descriptor $(\neg L_i\&L_{i+1}\&\ldots\&L_m, (\emptyset, \emptyset), (\emptyset, \emptyset))$, which describe the complement of $L_1\&\ldots\&L_m$ deterministically. The initial rule for $1 \leq i \leq n$ has label descriptor $(\mathit{true}, (\emptyset, \{x_i\}), (\emptyset, \emptyset))$ describing all variable subsets that do not contain x_i. D inherits selection and rejection states from D_1.

Case $F = O_w$**.** The ENWA descriptor D for O_w has to compute the concatenation of all strings of text nodes contained in the subtree of the start node, and has to compare it to w with respect to O. This is done by creation of a deterministic finite state automaton B that accepts all strings w' such that (w', w) is in the relation induced by O. The idea is that D runs automaton B on all strings of text nodes in the subtree of the start node. D maintains a copy of states of B, called *copy states*. D's main run r remains in copy states of B whenever not reading strings of text nodes of the subtree of the start node, while r switches to corresponding states of B at text nodes. The main run starts in the initial copy state of B and stays there until the first text node arrives at which it changes to the initial state of B. The string of the text node is consumed by B, while reaching some state q which is not necessarily a final state of B. At the corresponding closing event of the text node, the main run goes into the corresponding copy state q_{copy} for q and stays there until the next text node arrives, where r continues as before. The main run continues until the closing event of the start node, where O_w can be decided. Whenever B blocks, the main run moves into a rejection state, and whenever a final state of B is reached, it moves into a selection state.

Case $F = \mathit{true}$**.** The ENWA descriptor D for true has five plus $2|\mathcal{E}|$ rules with label descriptor $(\mathit{true}, (\emptyset, \emptyset), (\emptyset, \emptyset))$: one opening initial rule to a selection state, opening and closing rules looping in the selection and the rejection state, and $2|\mathcal{E}|$ closing rules to close the stream in the selection and rejection state.

Proposition 6 *(Correctness). Let $\mathcal{E}$ be a set of "external" stack symbols, $n \geq 0$, $S \in \mathcal{E}^n$, F an* FXP *formula with variables in a finite set $V \subseteq \mathcal{V}_0$, and $D = D_F(\mathcal{E})$ be the constructed* ENWA*. Then*

$$\mathcal{L}_S\big(D_F(\mathcal{E})\big) = \mathit{Can}\big(\mathcal{L}_n(F)\big)$$

The proof is straightforward by induction on the structure of FXP-formulas, as shown in the appendix of the long version [22].

Definition 7. We call an Fxp formula *determinization free* if it does not contain any disjunction or any axis from ch^+, ns^+, and *fo* below a negation.

For the complexity analysis of our compiler, we will need to assume that the input Fxp formula is *simple*, in that all label subformulas are of one of the following three forms: $B(O_w)$, $B(A(F))$ or $B(true)$. We can transform all Fxp formulas into equivalent simple Fxp formulas by applying the rewrite rule $B(F) \to B(true) \wedge F$ everywhere. This procedure, however, would increase the conjunction width, on which the efficiency of our compiler depends. Instead, we will assume that the input formulas is simply labeled in the following sense.

Definition 8. We call an occurrence of an operator in a formula *labeled* if it has some ancestor, which is a label formula B but without having any axis A in between. We call a formula *simply labeled* if it does not contain any labeled disjunctions and negations.

Lemma 9. *Any simply labeled* Fxp *formula can be transformed into a simple* Fxp *formula with the same conjunction width in linear time.*

Proof. We apply the following two rewrite rules exhaustively in a bottom–up manner. This can be done in linear time since no subformulas are copied, and it does not affect the number of conjunctions.

$$B\big(F \wedge F'\big) \to B(F) \wedge F'$$
$$B\big(B'(F)\big) \to B\&B'(F)$$

Both rules preserve simple labeling, since whenever one of them applies, all unlabeled disjunctions and negations of F and F' must be located below some axis. Therefore, they will remain unlabeled. If none of these rules applies any more, the resulting Fxp formula is simple: subformulas $B(F \wedge F')$ and $B(B'(F))$ would be rewritten, $B(F \vee F')$ and $B(\neg(F))$ are excluded since not simply labeled, and all other three possible forms $B(true)$, $B(A(F))$ and $B(O_w)$ are simple. □

Lemma 10 *(Size of automaton descriptor). There exists a constant c' such that for any* Fxp*-formula F, that is determinization free and simply labeled, and for any finite set $\mathcal{E}$ of external stack symbols: $|D_F(\mathcal{E})| \leq (|F|(c' + 2|\mathcal{E}|))^{w(F)+1}$.*

Proof. By Lemma 9 we can make F simple in a preprocessing step in linear time without changing the conjunction width $w(F)$. For the treatment of recursive axis ch^+, ns^+, and *fo*, we need nondeterministic automata with multiple initial rules. If we always had a single initial rule, then the proof of the lemma was straightforward. With multiple initial rules, we need an amortized cost analysis. Therefore, we define the amortized size of an eNwa descriptor D by $AS(D) = |D| + |init(D)|$ where $|D|$ is the size of D and $init(D)$ the set of initial rules of D, i.e. rules that depart an initial state. With the constant $c' = 7$, we can show for all determinization free Fxp formulas F that:

$$AS\big(D_F(\mathcal{E})\big) \leq \big(|F|\big(c' + 2|\mathcal{E}|\big)\big)^{w(F)+1}$$

The lemma will then follow from $|D_F(\mathcal{E})| \leq AS(D_F(\mathcal{E}))$. The proof is by induction on the structure of F. Let $D = D_F(\mathcal{E})$. Let $\mathcal{E}'$ be the extension of $\mathcal{E}$ and F_1 and possibly F_2 the subformulas of F, such that D was constructed from $D_1 = D_{F_1}(\mathcal{E}')$ and possibly $D_2 = D_{F_2}(\mathcal{E}')$. We write ω, ω_1, and ω_2 for the respective conjunction widths of $w(F)$, $w(F_1)$, and $w(F_2)$, and I, I_1 and I_2 for the number of initial rules of D, D_1, and respectively D_2.

Case $F = F_1 \wedge F_2$**.** The eNwa descriptor D is the product D_1 and D_2 where $\mathcal{E} = \mathcal{E}'$. Hence $|D| = |D_1||D_2|$ and $I = I_1 I_2$, so that:

$$\begin{aligned}
AS(D) &\leq AS(D_1)\, AS(D_2) \\
&\leq \big(|F_1|\big(c' + 2|\mathcal{E}|\big)\big)^{\omega_1+1} \big(|F_2|\big(c' + 2|\mathcal{E}|\big)\big)^{\omega_2+1} \quad \text{(ind. hypo.)} \\
&\leq \big(|F|\big(c' + 2|\mathcal{E}|\big)\big)^{\omega_1+1} \cdot \big(|F|\big(c' + 2|\mathcal{E}|\big)\big)^{\omega_2+1} \\
&= \big(|F|\big(c' + 2|\mathcal{E}|\big)\big)^{\omega_1+\omega_2+2} \\
&= \big(|F|\big(c' + 2|\mathcal{E}|\big)\big)^{\omega+1}
\end{aligned}$$

Case $F = F_1 \vee F_2$**.** The eNwa descriptor D is the union of the eNwa descriptors D_1 and D_2 where $|\mathcal{E}| = |\mathcal{E}|'$, so $|D| = |D_1| + |D_2|$ and $I = I_1 + I_2$. It follows that $AS(D) = AS(D_1) + AS(D_2)$ and thus smaller than in the case of conjunction.

Case $F = \neg F_1$**.** Since F is determinization free, D_1 describes a deterministic automaton, which furthermore can never get stuck, so that we only need to swap rejection and selection states of D_1 in order to obtain D. Hence, by induction hypothesis: $AS(D) = AS(D_1) \leq (|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} \leq (|F|(c' + 2|\mathcal{E}|))^{\omega+1}$.

Case $F = ch(F_1)$ where F_1 contains neither recursive axes nor disjunctions. Since F_1 contains no nondeterministic constructs, the automaton described by D_1 will be deterministic. The eNwa described by D will thus run the eNwa described by D_1 (where $\mathcal{E} = \mathcal{E}'$) on all children of the marked node, until one of these runs succeeds. The rules of the head of D_1 are rewritten for recomputing the stack symbols that D_1 used at the roots of all children of the marked node, but the number of rules is not increased thereby. In addition, $c' = 7$ new rules were added for the treatment of the marked node. Therefore, $|D| = |D_1| + c'$. The size of the head of D is smaller than that of D_1. Hence, $AS(D) \leq c' + AS(D_1) \leq c' + (|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} \leq (|F|(c' + 2|\mathcal{E}|))^{\omega+1}$.

Case $F = A(F_1)$ where $A \in \{ch^+, ns^+, fo\}$ or $F = ch(F_1)$ where F_1 contains recursive axes or disjunctions. Now $\mathcal{E}'$ is obtained by $\mathcal{E}$ by adding at most c' new stack symbols. The eNwa descriptor D has a generator state, from which it starts all initial rules I_1 of D_1, when being at an A-successor of the marked node. For $A = fo$ maximum $|\mathcal{E}|$ rules are added in order to find all following nodes in any future. In addition, $c' = 7$ new rules were added for the treatment of the main run of D. Thus, $|D| \leq |D_1| + I_1 + c' + |\mathcal{E}|$, so that:

$$\begin{aligned}
AS(D) &= |D_1| + I_1 + c' + |\mathcal{E}| \\
&= AS(D_1) + c' + |\mathcal{E}| \\
&\leq \big(|F_1|\big(c' + 2|\mathcal{E}|\big)\big)^{\omega_1+1} + c' + |\mathcal{E}| \quad \text{(ind. hypo.)} \\
&\leq \big(\big(|F_1| + 1\big)\big(c' + 2|\mathcal{E}|\big)\big)^{\omega_1+1} \\
&= \big(|F|\big(c' + 2|\mathcal{E}|\big)\big)^{\omega+1}
\end{aligned}$$

Case $F = B(F_1)$. Since F is simply labeled, F_1 has the form $A(F_2)$ or O_w or *true*. Hence, descriptor D_1 for F_1 has only one initial state. The size $|D|$ of the eNwa descriptor D is the size of D_1, plus $|B|$ opening rules to the initial state of D_1, with which the complement of B is described (for pseudo-completeness). Hence $|D| = |D_1| + |B|$ and the number of initial rules I of D is $I_1 + |B|$. Therefore $AS(D) = |D_1| + |B| + I = AS(D_1) - I_1 + |B| + I_1 + |B| \leq (|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} + 2|B| \leq (|F|(c' + 2|\mathcal{E}|))^{\omega+1}$, since $2 \leq c'$ and $|F| = |F_1| + |B|$.

Case $F = O_w$. The size of D is the sum of at most $3 \cdot |w|$ many rules for the finite state automaton B (that accepts all strings w' such that (w', w) is in the relation induced by O), $3 \cdot |w|$ many rules for moving in and out of copy states of B as described above, and at most c' many rules for the treatment of Ds main run. In addition D needs $2|\mathcal{E}|$ rules to close the stream in selection and rejection states. Hence the size of D is smaller or equal to $6|w| + c' + 2|\mathcal{E}|$, while the head of D is constant and can be bounded by c'. As $|F| = |w|$ it follows that $AS(D) \leq (|F|(c' + 2|\mathcal{E}|))^{\omega+1}$.

Case $F = true$. The size $|D|$ of the eNwa descriptor D for *true* is 5 and contains $2|\mathcal{E}|$ many rules to close the stream. D's amortized size is thereby smaller than $(|F|(c' + 2|\mathcal{E}|))^{\omega+1}$. □

Theorem 11 *(Compilation time). There exist $c, c' > 0$ such that for any determinization free and simply labeled* Fxp*-formula F and any finite set $\mathcal{E}$ of external stack symbols, the time to compute the* eNwa *descriptor $D_F(\mathcal{E})$ is at most $2c(|F|(c' + 2|\mathcal{E}|))^{w(F)+1}$.*

Proof. We fix a set $\mathcal{E}$ and constants $c_1 = 2$, $c' = 7$ and $c = c_1 + c' = 9$. We again need an amortized analysis. We define the amortized time of an eNwa descriptor $D = D_F(\mathcal{E})$ constructed by our algorithm as follows, where $T(D)$ is the construction time:

$$AT(D) = T(D) + \big|h(D)\big| + |\mathcal{E}|$$

The proof is by induction on the structure of Fxp formulas F, under the assumption that they are determinization free and simply labeled. Let F_1 and possibly F_2 be the arguments of the top-level operator of F, and $D_i = D_{F_i}(\mathcal{E}')$ be the subautomata from which D was constructed. We will also write H for $|h(D)|$, H_i for $|h(D_i)|$, and similarly T for $T(D)$, T_i for $T(D_i)$.

Case $F = F_1 \wedge F_2$. The time to compute an eNwa descriptor $D = D_F(\mathcal{E})$ for the product F is the sum of the times for computing the eNwa descriptors $D_1 = D_{F_1}(\mathcal{E})$ and $D_2 = D_{F_2}(\mathcal{E})$, plus the time to compute eNwa descriptor D for F, which can be done in time $c_1 \cdot |D_1| \cdot |D_2|$. Hence

$$\begin{aligned}
AT(D) &= T_1 + T_2 + c_1|D_1||D_2| + H + |\mathcal{E}| \\
&\leq AT(D_1) + AT(D_2) + c_1|D_1||D_2| + H \\
&\leq AT(D_1) + AT(D_2) + (c_1 + 1)|D_1||D_2| \\
&\leq 2c\big(|F_1|\big(c' + 2|\mathcal{E}|\big)\big)^{\omega_1+1} + 2c\big(|F_2|\big(c' + 2|\mathcal{E}|\big)\big)^{\omega_2+1} \\
&\quad + (c_1 + 1)|D_1||D_2|
\end{aligned}$$

$$
\begin{aligned}
&\leq 2c\big(\big(|F_1|\big(c'+2|\mathcal{E}|\big)\big)^{\omega_1+1}+\big(|F_2|\big(c'+2|\mathcal{E}|\big)\big)^{\omega_2+1}\\
&\quad +|D_1||D_2|\big) && (c_1+1\leq c)\\
&\leq 2c\big(\big(|F_1|\big(c'+2|\mathcal{E}|\big)\big)^{\omega_1+1}+\big(|F_2|\big(c'+2|\mathcal{E}|\big)\big)^{\omega_2+1}\\
&\quad +\big(|F_1|\big(c'+2|\mathcal{E}|\big)\big)^{\omega_1+1}\big)\big(|F_2|\big(c'+2|\mathcal{E}|\big)\big)^{\omega_2+1} && \text{(Lemma 10)}\\
&\leq 2c\big(\big(|F_1|+|F_2|\big)\big(c'+2|\mathcal{E}|\big)\big)^{\omega_1+\omega_2+2} && \big(a^m+b^n+a^mb^n\leq(a+b)^{m+n}\big)\\
&\leq 2c\big(|F|\big(c'+2|\mathcal{E}|\big)\big)^{\omega+1}
\end{aligned}
$$

Case $F=F_1\vee F_2$**.** The time to compute an ENWA descriptor D is the sum of the times to compute D_1 and D_2. The size of the head of D is the sum of the head of D_1 and the head of D_2. Hence

$$
\begin{aligned}
AT(D)&=T+H+|\mathcal{E}|=T_1+T_2+H+|\mathcal{E}|\\
&=AT(D_1)-H_1-|\mathcal{E}|+AT(D_2)-H_2-|\mathcal{E}|+H+|\mathcal{E}|\\
&\leq AT(D_1)+AT(D_2)
\end{aligned}
$$

and thus smaller than in the case of conjunction.

Case $F=\neg F_1$**.** Since F is determinization free, D_1 describes a deterministic automaton, which furthermore can never get stuck, so that we only need to swap rejection and selection states of D_1 in order to obtain D. Hence, by induction hypothesis: $AT(D)=AT(D_1)\leq 2c(|F_1|(c'+2|\mathcal{E}|))^{\omega_1+1}\leq 2c(|F|(c'+2|\mathcal{E}|))^{\omega+1}$.

Case $F=ch(F_1)$ **where** F_1 **contains neither recursive axes nor disjunctions.** Since F_1 contains no nondeterministic constructs, the automaton described by D_1 will be deterministic. The time to compute D is the time to compute D_1 (where $\mathcal{E}=\mathcal{E}'$), plus the time to rewrite the head of D_1 for the recomputation of stack symbols pushed at root of the children of the marked node. In addition D takes time $2|\mathcal{E}|$ to delete closing stream rules for the rejection state of D_1 and to add these to the new rejection state of D_1, and time at most c' to add at most c' new rules for the treatment of the marked node. The head of D contains only 5 rules. Hence

$$
\begin{aligned}
AT(D)&=AT(D_1)-H_1-|\mathcal{E}|+H_1+2|\mathcal{E}|+c'+H+|\mathcal{E}|\\
&=AT(D_1)+2|\mathcal{E}|+c'+5\\
&\leq 2c\big(|F_1|\big(c'+2|\mathcal{E}|\big)\big)^{\omega_1+1}+2|\mathcal{E}|+c'+5 && \text{(ind. hypo.)}\\
&\leq 2c\big(\big(|F_1|+1\big)\big(c'+2|\mathcal{E}|\big)\big)^{\omega_1+1} && 5\leq c\\
&=2c\big(|F|\big(c'+2|\mathcal{E}|\big)\big)^{\omega+1}
\end{aligned}
$$

Case $F=A(F_1)$ **where** $A\in\{ch^+,ns^+,fo\}$ **or** $F=ch(F_1)$ **where** F_1 **contains recursive axes or disjunctions.** The time to compute D is the time to compute D_1 where $\mathcal{E}'$ is obtained from $\mathcal{E}$ by adding at most c' new stack symbols. Furthermore D needs time to compute I_1 many rules for starting the test for F_1 from Ds generator state, plus $|\mathcal{E}|$ closing rules to reach all following nodes of the stream for the case that $A=fo$, plus at most c' rules for the treatment of the main run. The head $H=H_1-I_1+2$, as the head of D contains all closing rules of the head of D_1, but none of the initial rules belonging to the head of D_1, such that D needs two new initial rules. Therefore $AT(D)=T(D)+h(D)+|\mathcal{E}|$.

$$
\begin{aligned}
AT(D)&=T_1+I_1+|\mathcal{E}|+c'+h(D)+|\mathcal{E}|\\
&=AT(D_1)-H_1-|\mathcal{E}|+I_1+|\mathcal{E}|+c'+H_1-I_1+2+|\mathcal{E}|\\
&=AT(D_1)+c'+2+|\mathcal{E}|
\end{aligned}
$$

and thus smaller than in the case of $F=ch(F_1)$ where F_1 contains no nondeterministic constructs.

Case $F=B(F_1)$**.** Since we assumed that F is simply labeled, it follows that F_1 is either a comparison O_w, an axis $A(F_2)$, or *true*. In all cases, the ENWA descriptor D_1 for F_1 has only one initial state, and its head H_1 is of size at most c'. Therefore the time T for computing D is the time to compute D_1, plus the time to intersect label properties and variable restriction in B with the label descriptors of initial rules of D_1 (of which there are at most H_1 many), plus the time to add an opening rule for the initial state of D_1 to some rejection state of D_1 to describe the complement of B. The head of D contains all rules from the head of D_1, plus $|B|$ rules for the initial state in order to maintain pseudo-completeness. Hence

$$\begin{aligned}
AT(D) &= T + H + |\mathcal{E}| = T_1 + c' + |B| + H + |\mathcal{E}| \\
&= AT(D_1) - H_1 - |\mathcal{E}| + c' + |B| + H_1 + |B| + |\mathcal{E}| \\
&= AT(D_1) + c' + 2|B| \\
&\leq 2c\big(|F_1|\big(c' + 2|\mathcal{E}|\big)\big)^{\omega_1+1} + c' + 2|B| \quad \text{(ind. hypo.)} \\
&\leq 2c\big(|F|\big(c' + 2|\mathcal{E}|\big)\big)^{\omega_1+1} \qquad\qquad |F| = |F_1| + |B|
\end{aligned}$$

Case $F = O_w$**.** The time to compute D is the time to compute at most $3 \cdot |w|$ many rules for the finite state automaton B (that accepts all strings w' such that (w', w) is in the relation induced by O), the time to compute $3 \cdot |w|$ many rules for moving in and out of copy states of B as described above, the time to compute at most c' many rules for the treatment of Ds main run, plus $2|\mathcal{E}|$ rules to close the stream in selection and rejection states. D's head contains only four rules. With $|F| = |w|$ the amortized time to compute D is therefore at most $6|w| + c' + 2|\mathcal{E}| + 4 + |\mathcal{E}| \leq 2c(|F|(c' + 2|\mathcal{E}|))^{\omega+1}$.

Case $F = true$**.** The time to compute D is the time to compute the $5 + 2|\mathcal{E}|$ many rules. The head of D has size 3 (initial rule plus 2 closing rules in for the selection and rejection state). D's amortized time is therefore $5 + 2|\mathcal{E}| + 3 + |\mathcal{E}| = 8 + 3|\mathcal{E}|$, which is smaller than $2c(|F|(c' + 2|\mathcal{E}|))^{\omega+1}$. □

6. Early query answering

We show how to use eNwa descriptors for evaluating monadic queries on Xml streams. The main idea is to generate on the fly all possible answer candidates, and to run the described eNwa on all of them in parallel in a streaming manner. The nondeterminism and the automata descriptors are resolved by on-the-fly instantiation and determinization. We then improve this streaming algorithm in an important manner, so that the stacks and states of the runs of multiple answer candidates in the same state may be shared.

6.1. On-the-fly instantiation and determinization

Let D be a descriptor of an eNwa E' that defines a monadic query, i.e., with tag alphabet $\{a, a^x \mid a \in \Sigma\}$ where x is a fixed variable. We are interested in running the determinization E of E'. This can be done while generating the needed part of E from D on the fly (and thus without ever constructing E'). At any time point, we store the subset of the states and transitions of E that were used before. If a missing transition rule is needed for some state $\{(q_1, q'_1), \ldots, (q_k, q'_k)\}$ of E and some label l then for each $1 \leq i \leq k$ one computes transitions of D from state q_i that describe label l (if not already computed once). These transition rules are then used to compute the result state of E by applying the determinization procedure. It should be noticed that each transition of E can be computed in polynomial time (but not in linear time). For example, to compute a missing closing transition of the deterministic automaton by on-the-fly determinization, our algorithm needs time $O(|Q|^4 \cdot |\Sigma|)$, where Q are the states of E' (see [22]). Recall also that the states of E are the sets of pairs of states of E'. For efficiency reasons, we will substitute such sets by integers, so that the known transitions of E can be executed as efficiently as if a deterministic eNwa E' was given at beforehand. Therefore, we can safely separate the aspects of on-the-fly instantiation and determinization from what follows.

6.2. Streaming algorithm for deterministic ENWAs

We present a streaming algorithm that answers monadic queries defined by a deterministic eNwas E in an early manner. This algorithm can then be lifted to eNwa descriptors by on-the-fly determinization and instantiation, as explained above. We will also assume that the deterministic eNwa E is pseudo-complete, so that it has a unique complete run on any document.

Buffering possibly alive candidates Suppose that we are given a stream containing a nested word of some data tree, and that we want to compute the answers of the query defined by E on this data tree in an early manner. That is, we have to find all nodes of the data tree that can be annotated by x, so that E can run successfully on the annotated data tree. At any event e of the stream, our algorithm maintains a finite set of candidates in a so-called buffer. A candidate is a triple that contains a value for x, a state of E that is neither a selection nor a rejection state, and a stack of E. The value of x can either be a node opened before the current event e, or "unknown" which we denote by $\bullet$. At the start event, there exists only a single candidate in the buffer, which is $(\bullet, q_0, \bot)$ where q_0 is the unique initial state of E and $\bot$ the empty stack. At any later event, there will be at most one candidate containing the $\bullet$.

Lazy candidate generation New candidates are generated lazily under supervision of the automaton. This can happen at all opening events for which there exists a candidate with the unknown value $\bullet$ (which then is unique). Consider the <a> event of some node π and let $(\bullet, q, S)$ be the candidate with the unknown value in the buffer at this event. The algorithm then computes the unique pair (γ, q') such that $q \xrightarrow{\langle a^x \rangle : \gamma} q'$ is a transition rule of E. If q' is a selection state, then π is

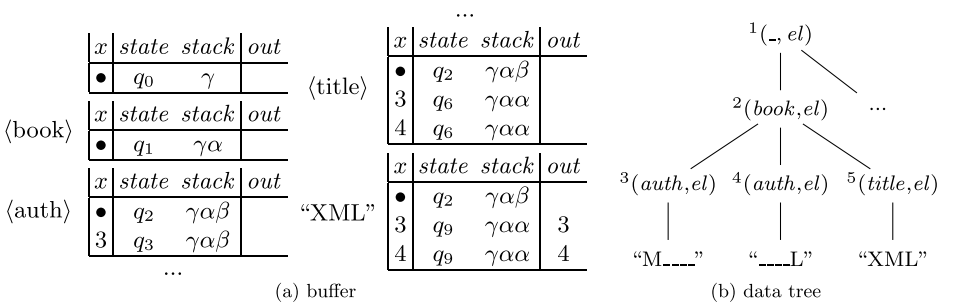

Fig. 8. Evolution of the buffer (a) for the eNwa from Fig. 3 when answering the XPath query `book[starts-with(title,'XML')]/auth` on the suffix of the data tree (b) with start node book.

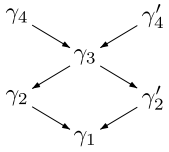

Fig. 9. Dag representing stacks.

an answer of the query, so we can output π directly. If q' is a rejection state, then π is safe for rejection (since E is deterministic), so we can ignore it. Otherwise, π may still be alive, so we add the candidate $(\pi, q', S\gamma)$ to the buffer.

Candidate updates At every event, all candidates in the buffer must be updated except for those that were newly created. First, the algorithm updates the configuration of the candidate by applying the rule of E with the letter of the current event to the configuration of the candidate. If a selection state is reached, the node of the candidate is output and discarded from the buffer. If a rejection state is reached, the candidate is also discarded from the buffer. Otherwise, the node may still be alive, so the candidate is kept in the buffer.

Example We illustrate the basic algorithm in Fig. 8 on the eNwa from Fig. 3 and the suffix of the document from Fig. 2 with the book as start node. Initially the buffer contains a single candidate with the unknown node •, that starts in the initial state of the eNwa. According to opening tag ⟨*lib*⟩ we launch the open transition and apply state and stack changes. At the opening event of node 3, i.e., when reading the open tag ⟨*auth*⟩ in state q_1, a new candidate is created. This is possible, since there exists the transition rule $q_1 \xrightarrow{\langle auth^x\rangle:\beta} q_3$ in the eNwa and since q_3 is neither a rejection nor a selection state. Similarly a new candidate will be created for node 4 at its opening event. Only after having consumed the text value of the title node 5, a selection state is reached for the candidates with node 3 and 4, such that they can be output and removed from the buffer.

6.3. Adding stack-and-state sharing

For most queries of the XPathMark benchmark, the buffer will contain only 2 candidates at every event, of which one is the candidate with the unknown value •. It may happen though that the number of candidates grows linearly with the size of the document. An example is the XPath query `/child::a[following::b]` on a document whose root has a large list of only *a*-children. There the processing time will grow quadratically in the size of the document. All candidates (of which there are $O(n)$ for documents of size n) must be touched for all following events on the stream (also $O(n)$). A quadratic processing time is unfeasible even for small documents of some megabytes, so this is a serious limitation.

We next propose a data structure for state and stack sharing, that allows to solve this issue. The idea is to share the work for all candidates in the same state, by letting their stacks evolve in common. Thereby the processing time per event for running the eNwa on all candidates will become linear in the number of states and stack symbols of the eNwa, instead of linear in the number of candidates in the buffer. In addition to this time per event, the algorithm must touch each candidate at most three times, once for creation, output, and deletion. We will use a directed acyclic graph (Dag) with nodes labeled in Γ for sharing multiple stacks. For instance a Dag for stacks $\gamma_1\gamma_2\gamma_3\gamma_4$ and $\gamma_1\gamma'_2\gamma_3\gamma'_4$ share nodes for stack symbols γ_1 and γ_3, as depicted in Fig. 9.

In addition, we use a table $B : Q \times \Gamma \to Aggreg$ relating a state and a root of the Dag through an aggregation of candidates. Table B aims at storing enough information when sharing at opening events, so that one can undo the sharing properly at closing events. Formally, *Aggreg* is the least set such that (1) it contains all sets of candidates, and (2) all subsets of $\{(a, \gamma) \mid a \in Aggreg, \gamma \in \Gamma\}$. For instance, the Dag and its associated B table at the `<title>`-event in Fig. 8 is illustrated in Fig. 10. Here we have $B(q_6, \alpha) = \{3, 4\}$ and $B(q_2, \beta) = \{\bullet\}$. In this case, the aggregations are just sets of candidate nodes. In general, an aggregation contains information on how to unshare candidates related to a root in the Dag which represents

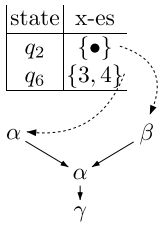

Fig. 10. Buffer of <title>.

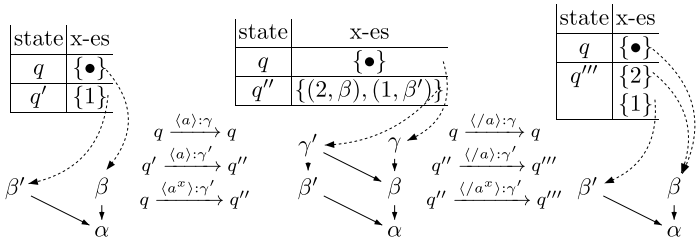

Fig. 11. Data structures for the state sharing algorithm.

more than one stack. This means one needs more information than just a set of candidates, and for this reason we introduce a nested structure. Let $B(q_i, \beta_i) = A_i$ for aggregations A_i in state q_i with stack symbol β_i on top of the stack, for $1 \le i \le k$ (i.e. there are k roots).

At an opening event the automaton may have transitions that lead into the same state q pushing the same stack symbol β for some of the states q_i, for instance q_2 and q_3. Since the corresponding aggregations A_2 and A_3 originate from different roots in the DAG, we add this information in a new aggregation $A = \{(A_2, \beta_2), (A_3, \beta_3)\}$. Consider for instance the situation depicted in Fig. 11. From the first configuration, we reach the second with the <a>-event. There the candidate 2 is created from the $\bullet$-candidate whose configuration has β on top of the stack, goes into state q'', and pushes γ'. However, there is also the candidate 1 which will go into the same state q'' while pushing the same stack symbol γ', but from a configuration with β' on top of the stack. The pairs $(2, \beta)$ and $(1, \beta')$ must be stored in the aggregation, so we define $B(q'', \gamma') = \{(2, \beta), (1, \beta')\}$.

At closing events for aggregations that contain pairs (A_i, β_i) of an aggregation A_i and a stack symbol β_i, the unsharing is done by relating aggregation A_i to the new root β_i in the DAG, after closing. For instance the closing event is read when moving from the second to the third situation in Fig. 11. There we have to undo the sharing. We decompose the aggregate and update the data structure to $B(q''', \beta) = \{2\}$, $B(q''', \beta') = \{1\}$ and $B(q, \beta) = \{\bullet\}$.

At every time point, B contains each buffered candidate exactly once. Whenever a selection state is reached in the B-table, the candidates in the aggregate of this state will be output and the aggregate will be deleted from the data structure. For rejection states, we only have to discard the aggregate. Note that rejected or selected candidates get deleted entirely from the data structure this way, since no candidate may appear twice in different aggregates, again due to determinism.

Proposition 12 *(Time per event). For every deterministic* eNWA *E defining a monadic query P and data tree t the time complexity per event of our query answering algorithm with stack-and-state sharing to compute $P(t)$ is in $\mathcal{O}(|Q|)$, where Q is the set of states of E.*

We can even reduce $|Q|$ to the maximal number of states per event that are assigned to the buffered candidates. This bound is at most 2 for all queries in our practical experiments. In practice, therefore, the runtime should grow linearly in the size of the document, and be independent of the size of the query. This will indeed be confirmed by the experiments in Fig. 18, which also indicate that the costs for on-the-fly instantiation and determinization are irrelevant in practice.

Proof. We start with the case of opening events. Since E is deterministic there is only one opening transition per state and label. According to the tag alphabet $\{a, a^x \mid a \in \Sigma\}$, the query answering algorithm has therefore two choices for opening transitions for an opening event a. Hence at most $2|Q|$ edges to new roots in the DAG may be added. For candidates in table B additional information may have to be stored (which needs only constant time). This happens, when candidates in different states related to different roots in the DAG reach the same state at an opening event. The evolution of the DAG at opening events can therefore be done in time $\mathcal{O}(|Q|)$.

For internal events, the DAG structure is not altered. At most $|Q|$ many state changes occur in table B.

We finish with the case of closing events. Closing transitions are applied according to the label of the closing event e' and to table B, which relates states and roots of the DAG to an aggregation of candidates. At first glance it seems as there could be $|Q||\Gamma|$ (hence quadratically many) possible transitions. Nevertheless the number of possible closing transitions at e' is bounded by

1. the number of opening transitions for opening event e that corresponds to e' (of which there are maximum $2|Q|$ many), plus
2. those closing transitions for potential candidates that were created between events e and e'.

(1) is obvious, in that at event e table B contained maximum $2|Q|$ entries to aggregations C of candidates. As E is deterministic at event e' there are thus $2|Q|$ many possibilities for these aggregations C to be related with stack symbols on top of the Dag. For (2) let us consider the creation of new candidates between events e and e'. All these candidates were instantiated from the unique $\bullet$ candidate, such that they share parts of the stack. At the closing event all potentially created candidates therefore are related to the unique stack symbol γ in the top of the Dag belonging to the $\bullet$ candidate. These new candidates may be in at most $|Q|$ many different states. Hence with (1) and (2) maximum $3|Q|$ closing transition may be applied. For these only a constant number of operations are needed to unshare the aggregations of candidates. Hence all together the run-time per event is in $\mathcal{O}(|Q|)$. □

Theorem 13 *(Time and space). For any deterministic* eNwa E *with state set* Q *defining a monadic query* P *and data tree* t*, the time complexity of our streaming algorithm with stack-and-state sharing to compute* $P(t)$ *is in* $\mathcal{O}(|E| + |Q||t|)$ *and its space complexity in* $\mathcal{O}(|E| + depth(t)|Q| + C)$*, where* C *is the maximal number of buffered candidates of* P *on* t *at any event.*

Proof. By Proposition 12, the run-time per event is linear in the number of states Q. With the construction of E we thereby obtain an overall run-time complexity which is in $\mathcal{O}(|E| + |Q||t|)$. Regarding space complexity, we have to keep E in memory all the time. The maximum number of buffered candidates at any event is C, while the Dag data-structure uses space in $depth(t)|Q|$. □

As we did in the time analysis, we can reduce $|Q|$ in our space analysis to the maximal number per event of states of the buffered candidates. Therefore, we can replace $|Q|$ by 2 in our practical experiments. So if the depth of the tree is bounded in addition (which is the case in our experiments), the space requirement is determined by the maximal number of buffered candidates per event, and by the size of the automaton. This will indeed be confirmed by our experiments in Table 2.

7. QuiXPath

We present an implementation of our algorithms in the QuiXPath 1.3 system and analyze its time and space performance experimentally.

7.1. Implementation, tools, and applications

We have implemented QuiXPath 1.3, a streaming query evaluator for a fragment of XPath 2.0. The implementation consists of 3 parts. First, a streaming algorithm with stack-and-state sharing for answering eNwa queries, second, a compiler from Fxp to eNwas, and third, a compiler from the supported fragment of XPath 2.0 to Fxp. All our implementation is done in Java 1.6, while relying on the free XPath parser from Saxon.

In addition to what we presented here, QuiXPath 1.3 supports top-level aggregation, which is implemented on basis of binary Fxp queries. Furthermore, there is a support for backwards axes, which are eliminated at the cost of forward axes and regular axes. Regular axes P^* are supported by QuiXPath 1.3, but only if P has 0-delay. This is enough for most backwards axes in the usual benchmarks. We would also like to notice that conditional regular axes are *not* sufficient for eliminating general backwards axes, as proven in [20].

QuiXPath 1.3 can be tested on the QuiX-Tools demo machine, which is freely available online at https://project.inria.fr/quix-tool-suite/demo. Furthermore, all newer versions of QuiXPath are available there, as well as the newest released version of all other QuiX-Tools, which are based on QuiXPath. Note that no installation effort is needed. All our experiments can be run there. We used another machine though, which features an Intel Core i7-2720QM processor at 2.20 GHz, 3.8 GB of RAM, and an SSD hard drive. The operating system is a 64-bit version of ubuntu 12.04 LTS.

7.2. Benchmarks

We will analyze the coverage and the time and space performance of QuiXPath 1.3, and compare it to other streaming XPath evaluators, such as Spex, Saxon 9.5, and Gcx. For this, we need an XPath benchmark collection, that consists of a collection of XPath queries and a collection of documents, so that both of them can be scaled in size.

We mainly use the usual XPathMark benchmark collection in its revised version (and not the one from the original paper [8]). These are queries about XMark documents that contain a table of bids and a table of bidders. The size of these documents can be scaled, basically by adding more bids and bidders. Some of the queries are parameterized by an integer $i \geq 1$, so that these queries can be scaled in size.

In Fig. 12 we have selected those queries from the XPathMark benchmark that can be answered by some of the available tools (QuiXPath 1.3, Spex, Saxon 9.5, and Gcx) and left out all others. We keep the 16 queries `A1–A8`, `B1–B7`, and `C2`, and

```
A1:   /site/closed_auctions/closed_auction/annotation/description/text
      /keyword
A2:   //closed_auction//keyword
A3:   /site/closed_auctions/closed_auction//keyword
A4:   /site/closed_auctions/closed_auction
      [annotation/description/text/keyword]/date
A5:   /site/closed_auctions/closed_auction[descendant::keyword]/date
A6:   /site/people/person[profile/gender and profile/age]/name
A7:   /site/people/person[phone or homepage]/name
A8:   /site/people/person
      [address and (phone or homepage) and (creditcard or profile)]/name
B1:   /site/regions/*/item[parent::namerica or parent::samerica]/name
B2:   //keyword/ancestor::listitem/text/keyword
B3:   /site/open_auctions/open_auction/bidder[following-sibling::bidder]
B4:   /site/open_auctions/open_auction/bidder[preceding-sibling::bidder]
B5:   /site/regions/*/item[following::item]/name
B6:   /site/regions/*/item[preceding::item]/name
B7:   //person[profile/@income]/name
B11(i): //open_auction(/bidder/..)^i/interval
B12(i): //item(/@id/..)^i/name
B13(i): //keyword(/ancestor::parlist/descendant::keyword)^i
B14(i): //bidder(/following-sibling::bidder/preceding-sibling::bidder)^i
B15(i): //keyword(/following::keyword/preceding::keyword)^i
C2:   /site/open_auctions/open_auction[bidder/increase = current]/interval
```

Fig. 12. XPathMark queries from http://goo.gl/qVugQc with parameter $i \geq 1$.

```
O1: /site[closed_auctions/closed_auction/type]//item
O2: /site[c or not(c)]//bidder
```

Fig. 13. Additional queries for illustrating the buffering behavior.

the 5 parameterized queries B11(i)–B15(i), since only these are supported by at least one of the above tools. With $i = 1$ these are 39% of the XPathMark queries only. We added 2 further queries O1 and O2 to our collection, which are given in Fig. 13. These queries do apply to the same XMark documents, but illustrate a different performance behavior since requiring to buffer much more candidates.

It should be noticed that neither Saxon 9.5 nor Gcx support XPath directly, but through Xslt respectively XQuery programs. Therefore we could not run the queries from the XPathMark directly, but had to embed them in an Xslt or XQuery program. This kind of rewriting is not fully trivial, if one does not want the experiments to be biased by different output modes. For Spex we used their internal output mode "count_results" and for QuiXPath 1.3 we counted the materialized answer nodes. For Saxon 9.5 we use a style sheet which counts the number of query answers. For Gcx we were not able to count the number of answers without producing them. Therefore, we decided not to produce any output (rather than using their default output mode, which is to print the subtrees of the selected nodes).

There is a further problem when comparing to Gcx, which is that Gcx does not support Xml attributes. Since we did not want to take Gcx out of the race, we had to adapt the benchmark in the case of Gcx as follows: in documents, we replaced attributes by elements, and in queries we replaced attribute axes by child axes. These adaptations are not fully neutral. Therefore, we will also compare Gcx and QuiXPath 1.3 on the same documents and queries in a separate study.

7.3. Performance comparison

In Table 1 we compare the different tools for their XPath "conceptual" coverage on the XPathMark since only 4 tools were available for testing in practice. For all others [13,25,27,28], we can still compare the coverage in terms of the concepts that are supported, and also in the percentage of queries that require these features. When presenting such percentages, we restrict ourselves to the XPathMark queries with parameter $i = 1$. This is an advantage for our competitor tools, since these cannot deal with $n \geq 2$ anyway. Later on, we will also experiment with QuiXPath 1.3 on queries with larger parameters $i \geq 2$ separately.

Only Gcx is able to deal with some queries with data joins such as C2. What is not covered by any of the tools are queries with data positions, full aggregations, and full negation. These features are used by the remaining 63% of queries of the XPathMark. Spex and QuiXPath 1.3 cover mostly the same features except for top-level aggregation. Nevertheless, QuiXPath 1.3 covers 37% of the XPathMark queries, while Spex can only deal with 22%. The other available tools cover even fewer queries, Gcx has 19% and Saxon 9.5 is at 6%.

The advantage in coverage of QuiXPath 1.3 over Spex is mostly due to the fact that Spex does not support all queries that are produced by backward axes elimination. Saxon's low coverage for XPath in streaming mode may be motivated by their focus on Xslt. The coverage of Gcx is lowered since it does not support backward axes. While the 37% coverage of QuiXPath 1.3 on the XPathMark benchmark is better than all previous tools, a much better result of 95% is achieved by QuiXPath 2.0, in follow-up work relying on all the algorithms presented here.

Table 1
System features.

Name	QuiXPath 1.3	Gcx	LNFA	Saxon 9.5	Spex	XMLtk	XSeq	Xsq
Reference		[29]	[27]	[15]	[26]	[13]	[25]	[28]
Language	XPath	XQuery	XPath	Xslt	XPath	XPath	XSeq	XPath
Available	yes	yes	no	yes	yes	yes	no	no
Coverage of XPathMark	37%	19%	18%	6%	22%	5%	7%	11%
Downward axis	✓	✓	✓	✓	✓	✓	✓	✓
Complex filters	✓	✓	✓	✗	✓	✗	✗	✓
Backward axis	◡	✗	✗	✗	✓	✗	✗	✗
Negation	◡	◡	✗	✗	◡	✗	✗	✗
Top level aggreg.	✓	✓	✗	✓	✗	✗	✗	✓
Nested aggreg.	✗	◡	✗	✗	✗	✗	✗	✗
Arithmetics	✗	✗	✗	✓	✗	✗	✗	✗
0-delay reg. axis	✗	✗	✗	✗	✗	✗	✓	✗
Regular axis	✗	✗	✗	✗	✗	✗	✗	✗
Join	✗	◡	✗	✗	✗	✗	✗	✗
Multi inputs	✗	✗	✗	◡	✗	✗	✗	✗

supported: ✓ partially supported: ◡ not supported: ✗

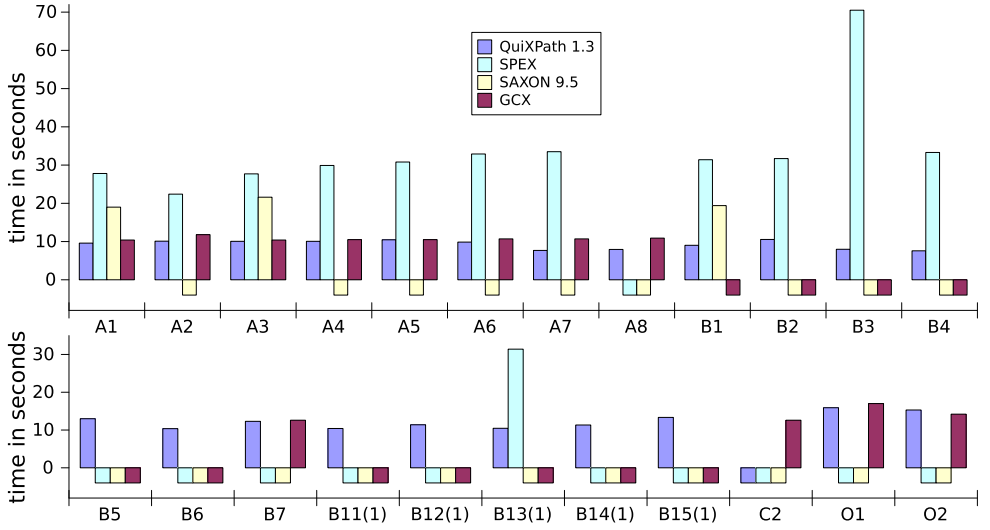

Fig. 14. Runtime in seconds of QuiXPath 1.3, Spex, Saxon 9.5, and Gcx for the benchmark queries obtained on a 1.1 GB XMark document. Whenever the query was supported, we depict a negative runtime. The runtimes were averaged over three runs, if there were no outliers (outliers were discarded).

In Fig. 14, we compare the runtime performance of QuiXPath 1.3 against Spex, Saxon 9.5, and Gcx on the benchmark queries. A negative runtime here means that the query was not supported by the respective tool. The document was fixed to a 1.1 GB XMark file. Generally, QuiXPath 1.3 and Gcx show very good performance, outperforming Saxon 9.5 and Spex.

The parametric queries `B11`(i)–`B15`(i) cannot be treated by neither Spex nor Gcx and also stresses QuiXPath 1.3 to its limits. The problem comes with backwards axes, which must be rewritten into forward axes,[4] such that the rewritten queries are supported by the query evaluator. Indeed, only QuiXPath 1.3 is able to execute some of the rewritten queries, as we will discuss in Section 7.5.

In Fig. 15 we give a more detailed comparison of the runtime performance of QuiXPath 1.3 and Gcx on the same 1.2 GB document without attributes with the queries adapted accordingly. It turns out that the parsing of Gcx is considerable higher than that of QuiXPath 1.3 which inherits the parser from Saxon, even though Gcx is implemented in C++ and the others in Java. On the other hand side, the overhead of Gcx with respect to the parsing time is lower than that of QuiXPath 1.3.

[4] In the general case, backwards axes elimination may need to introduce regular axes [20], but not for the XPathMark queries considered here.

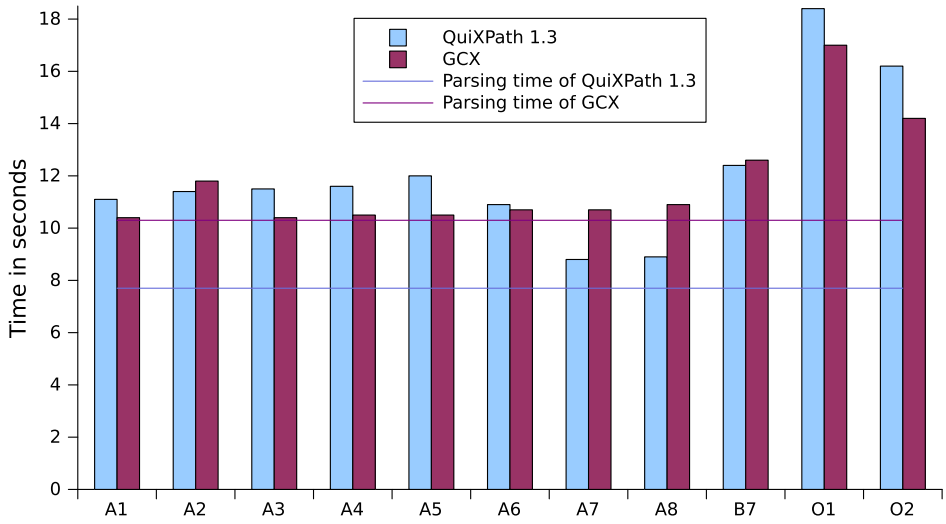

Fig. 15. Runtime comparison of Gcx and QuiXPath 1.3 for Gcx compatible queries on a compatible 1.2 GB XMark document.

7.4. Performance analysis

In order to understand the time efficiency of QuiXPath 1.3 on the different queries of the benchmark, it is essential to separate the parsing time from the time for pure query evaluation. This is relevant to all languages embedding XPath queries, since there, many XPath queries must be executed at the same time, while the document is parsed only once.

We propose to measure the *parsing-free evaluation time* $T_{parsefree}(P, d)$ of a query P on a document d as follows. We launch the query several times in parallel on the document, while parsing the document only once. When scaling the copy number, the parsing time should become irrelevant in the limit. Let $T(P, d, n)$ be the runtime that is needed to evaluate n copies of P in parallel on document d, and let $T_{parse}(d)$ be the parsing time for document d. We then define the parsing-free runtime $T_{parsefree}(P, d)$ of query P on document d as follows:

$$T_{parsefree}(P, d) = \lim_{n \to \infty} \frac{T(P, d, n) - T_{parse}(d)}{n}$$

The limit converged quickly for all our benchmark queries. Already for $n = 10$ the difference to the previous value was less than 50 ms for a 1.1 GB XMark document. In Fig. 16 we display the parsing-free evaluation times for all benchmark queries on the 1.1 GB XMark document, and related it to the overall runtime and to the parsing time.

The result shows that parser works in parallel with the query evaluator. The reason is that the CPU can work in parallel with file accesses to the stream. However, the degree of parallelism remains quite low for QuiXPath 1.3. It should also be noticed that we were not able to compute the parsing-free evaluation time for the other tools, so that we do not know whether they exploit more parallelism or not.

Our next objective is to explain the difference in the parsing-free evaluation times for the various queries. The most relevant parameter is the workload of a query on a document, which is the number of pairs of states and non-projected events (see the next paragraph), such that some buffered candidate was in this state at the event. Fig. 17 shows that the workload correlates nicely to the parsing free evaluation time for all queries of our benchmark, except for queries that store big numbers of candidates, whose rejection or selection requires additional time.

The QuiXPath 1.3 implementation supports 2 kinds of projections (on which the workload depends). First, if a query does not depend on text data, all text events are projected away. This is indeed the case for all queries in the benchmark except for query `C2` with data joins (that QuiXPath 1.3 cannot deal with). The second kind is depth projection for queries that do not use the descendant axis, so that they consider the nodes until a certain depth only. The queries with depth projection are exactly those with the lowest parsing-free evaluation time: `A7`, `A8`, `B3`, `B4` (depth projection at depth 4), `A6`, `B1` (depth projection at depth 5), and `A1`, `A4` (depth projection at depth 7). The next best queries are those with 0-delay, since this reduces the workload too: `A2`, `A3`, `B2`, `B6`, `B12`(1), `B13`(1). The next class contains queries without projection, but only few buffered candidates `A5`, `B5`, `B7`, `B11`(1), `B14`(1), and `B15`(1). The highest parsing-free evaluation time is obtained for the queries `O1` and `O2`, which buffer thousands of candidates for many events, but all of them in the same state.

QuiXPath 1.3 scales up to large document sizes. Thanks to stack and state sharing, the runtime remains linear in the size of the document as for all benchmark queries in Fig. 18 on larger documents ranging from 1 GB until 28 GB in size. This even holds for queries such as `O1` and `O2`, where the number of buffered candidates grows linearly. But since all of them are in the same state, the running times remain linear (and not quadratic). For document sizes greater than 17 GB the candidates needed to buffer for query `O2` did not fit into memory, such that no evaluation times could be reported.

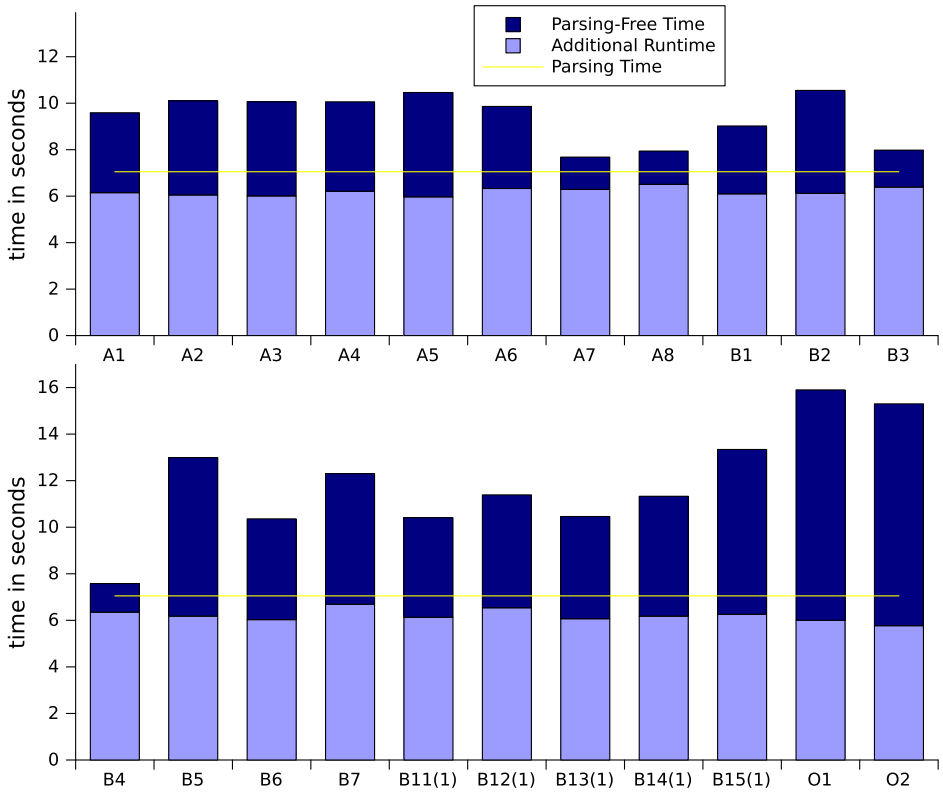

Fig. 16. Parsing-free time of QuiXPath 1.3 in relation to the additional runtime for the benchmark queries on the 1.1 GB XMark document.

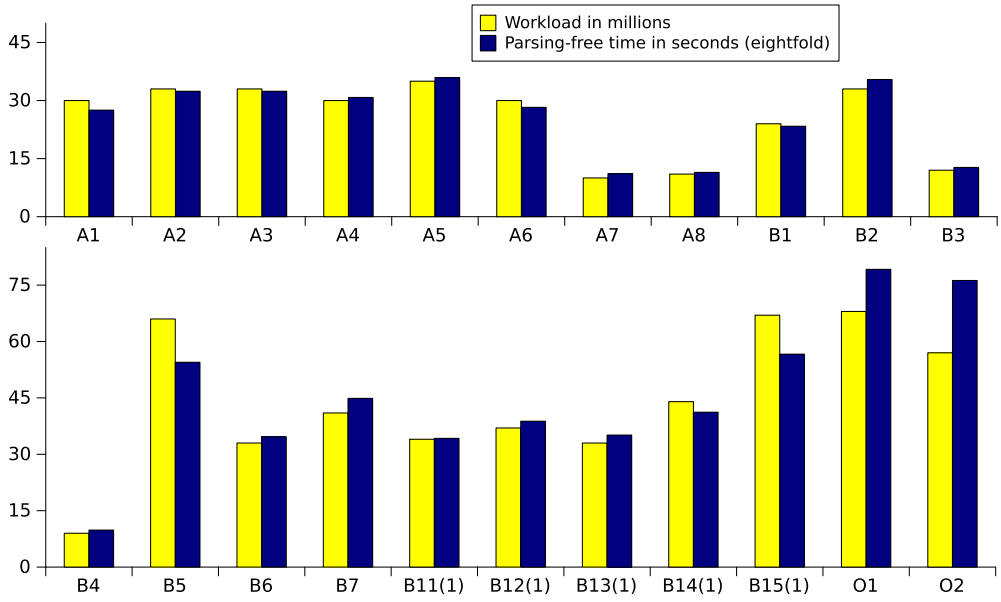

Fig. 17. Workload compared to parsing-free evaluation time.

Query O2 is the only query from our benchmark collection, where QuiXPath 1.3 is not earliest. In general QuiXPath 1.3 is not earliest with the presence of valid respectively unsatisfiable filters (as in the examples showing the hardness of earliest query answering [10]). It was also proven in follow-up work [23], that our early query answering algorithm is indeed earliest for a large subset of positive XPath queries.

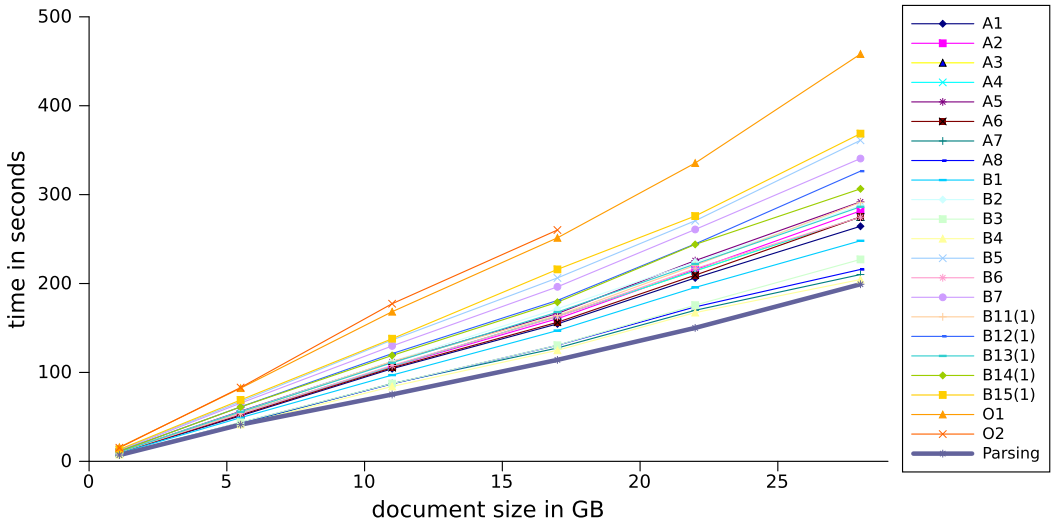

Fig. 18. Runtimes of the benchmark queries for large documents.

Table 2
Detailed analysis for non-parametric XPathMark queries. The queries are compiled to Fxp formulas F of size $|F|$ and conjunction width $w(F)$. N is the number of states of the eNwa descriptor constructed for F and $\mathrm{KB}_{\mathrm{desc}}$ the size of the memory needed for its storage. D is the number of states of the deterministic eNwa constructed by on-the-fly instantiation and determinization. $\mathrm{KB}_{\mathrm{run}}$ is the memory required at runtime for storing the deterministic eNwa and the candidate buffer, i.e., the overall memory needed is at least $\mathrm{KB}_{\mathrm{run}} + \mathrm{KB}_{\mathrm{desc}}$. T_{det} is the time in seconds needed to compute all D states by on-the-fly instantiation and determinization.

	A1	A2	A3	A4	A5	A6	A7	A8	B1	B2	B3	B4
$\|F\|$	25	13	17	31	24	34	37	115	43	41	19	19
$w(F)$	0	0	0	1	1	2	2	12	0	2	0	0
N	17	9	12	27	17	35	21	129	24	39	13	13
$\mathrm{KB}_{\mathrm{desc}}$	201	102	131	237	177	307	212	1186	276	411	136	141
D	14	10	14	27	23	32	23	127	14	49	16	15
$\mathrm{KB}_{\mathrm{run}}$	192	232	256	256	224	256	552	672	192	384	192	192
T_{det}	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.03	0.00	0.02	0.00	0.00

	B5	B6	B7	B11(1)	B12(1)	B13(1)	B14(1)	B15(1)	O1	O2
$\|F\|$	24	103	21	17	16	37	62	69	23	34
$w(F)$	1	4	1	1	0	2	0	3	1	2
N	13	58	15	12	10	32	33	34	25	18
$\mathrm{KB}_{\mathrm{desc}}$	161	571	122	120	99	389	326	372	251	193
D	32	28	20	17	11	28	19	29	25	12
$\mathrm{KB}_{\mathrm{run}}$	256	320	336	304	256	352	224	288	39 432	112 460
T_{det}	0.00	0.01	0.00	0.00	0.00	0.01	0.00	0.01	0.00	0.00

7.5. Detailed analysis

We provide a detailed analysis of the performance of QuiXPath 1.3 including measurements of its space consumption in particular.

We start with the analysis for the non-parametric XPathMark queries reported in Table 2. The first step is to compile the XPath queries into simple Fxp formulas F, while eliminating backwards axes. The size of the formula and its conjunction width becomes apparent only after this precompilation phase. The maximal values are reached for the simple Fxp formula of query `A8`, with a size of 112 and a conjunction width of 12. These quite large values are due to backwards axes elimination and the following rewriting into *simple* Fxp formulas.

The next compilation step is to convert F into a descriptor of the nondeterministic eNwa. The number N of the states of this descriptor remains moderately small with at most 129 states for `A8` and also the memory $\mathrm{KB}_{\mathrm{desc}}$ for its storage with at most 1186 KB. The number D of states visited by deterministic eNwa during on-the-fly instantiation and determinization also remains small with at most 127 for `A8`. The results confirm that the on-the-fly determinization inspects only a very small subset of the state space of the determinized automaton, and that the time required for on-the-fly instantiation and determinization (T_{det}) can be ignored in practice. The memory $\mathrm{KB}_{\mathrm{run}}$ needed for storing the deterministic automaton and the candidate buffer is small for all queries with few candidates with at most 672 KB for `A8`. It grows however linearly with the number of buffered candidates, as one can observe for queries `O1` and `O2`.

Table 3
Detailed analysis for the parametric queries B13(i)–B15(i). The backward axes elimination of QuiXPath 1.3 infers an (extended) Fxp formula F of size $|F|$ and conjunction width $w(F)$ up to 912! The Fxp formulas are then compiled into eNwa descriptors with N states, which can be stored in a memory of size KB_{desc}. On-the-fly instantiation and determinization yields a deterministic eNwa of which D states are visited. The memory for storing this deterministic automaton and the candidate buffer is KB_{run}. T_{det} is the time in seconds needed to for on-the-fly instantiation and determinization.

	B13(1)	B13(2)	B13(3)	B14(1)	B14(2)	B14(3)	B14(4)	B15(1)	B15(2)	B15(3)
$\|F\|$	37	173	863	62	374	2116	11 804	69	527	4149
$w(F)$	2	10	56	0	16	136	912	3	32	299
N	32	325	4273	33	237	1959	17 611	34	347	4294
KB_{desc}	389	4896	76 514	326	2560	22 291	230 733	372	4448	64 098
D	14	120	503	19	29	36	43	29	136	2625
KB_{run}	352	1683	32 747	224	384	1821	16 868	288	1448	394 780
T_{det}	0.01	0.76	113.02	0.00	0.02	1.13	21.05	0.00	0.54	971.88

The minimal memory required to evaluate a query in reasonable time is the sum of $KB_{desc} + KB_{run} + 2000$ KB, where the 2 MB serve for running the java virtual machine itself. Our runtime experiments were performed with 3 GB of memory in order to reduce overheads by repeated garbage collections. Less memory implies larger overheads. For query A1 we conducted a small experiment to estimate these overheads. We restricted the memory to 5, 15, 30, 60, and 120 MB and obtain as runtimes 14.0, 12.7, 10.7, 10.4, and 10.1 seconds respectively.

We next analyze the parametric queries B11(i)–B15(i). These contain backwards axes, which must be rewritten into forward axes. QuiXPath 1.3 can do this, but the sizes of the Fxp formulas obtained by backwards axes elimination may be large, as shown in Table 3. For instance, the formula for B14(4) has size 11 804 and conjunction width 912. As a consequence, the corresponding eNwa descriptors will be huge, so that they quickly do no more fit into main memory (for $i \geq 5$ for all B13(i), B14(i), B15(i)), or may not leave enough space for on-the-fly determinization (for B13(4) and B15(4)).

Fig. 19 shows the runtimes for the parametric queries that can still be evaluated by QuiXPath 1.3 with 3 GB of memory. It should be noticed that the queries B11(i) are equivalent for all $i \geq 1$, and similarly for B12(i), B13(i), B14(i), and B15(i) respectively. Nevertheless, the sizes of the corresponding Fxp formulas grow with parameter i and also the conjunction width (since our compiler ignores query equivalence). Despite this, our compiler produces the same eNwa descriptor for all B11(i) independently of i, and similarly for all B12(i), while the eNwa descriptors for B13(i), B14(i), and B15(i) do grow quickly with i. The surprisingly good treatment of the families B11(i) and B12(i) can be explained as follows. The Fxp formulas obtained after backward axes elimination contain conjunctions of the same filter over and over (the filter [./bidder] for B11(i) and the filter [@id] for B12(i)). The automata for such conjunctions are independent of the number of times that the filter is conjoined with itself.

The queries B13(i), B14(i), B15(i) with $i \geq 3$ illustrate that the costs of on-the-fly determinization may become relevant for the runtime in extreme cases. Otherwise the runtime should be independent of the size of the eNwa descriptor, as explained earlier. In extreme cases, the deterministic automata have thousands of states (D), each of which contains thousands of states of the nondeterministic automaton (N). This is problematic since the determinization time is not linear but highly polynomial: overall it is in time $\mathcal{O}(DN^4|\Sigma|)$. Indeed, the overall time for on-the-fly determinization (T_{det}) grows importantly in these cases. Furthermore, the backward axes elimination for B13(i), B14(i), and B15(i) produces Fxp formulas with many disjunctions, so that the candidates admit almost all of the N states of the nondeterministic eNwa. For this reason, the on-the-fly determinization requires high memory costs in these cases (included in KB_{run}), raising out-of-memory errors for B13(4) and B15(4).

8. Conclusion and future work

We have shown how to approximate earliest query answering for XPath on Xml streams by using early nested word automata. An implementation of our algorithms is freely available in the QuiXPath 1.3 system, which outperforms all existing tools in coverage and while the performance is similar to the best existing system Gcx.

Meanwhile, we developed the work presented here much further. The current version of QuiXPath, QuiXPath 2.0, covers a much larger fragment of XPath 2.0. In particular, it features data joins, arithmetics, and general aggregates. More than 95% of the XPathMark benchmark is covered. This requires further algorithms for implementation, which we did not publish so far. These algorithms will be based on networks of eNwas. The algorithms presented here are the special case with a singleton eNwa network.

QuiXPath 2.0 serves as a platform, on which we implemented the QuiX-Tool suite (see https://project.inria.fr/quix-tool-suite). Currently it provides streaming implementations of Xslt and XSchematron besides XPath, and in the near future, streaming implementation of XQuery and XProc will be added, as proposed by [18]. All the QuiX-Tools can be tested on the QuiX-Tools demo machine, which is freely available online at:

https://project.inria.fr/quix-tool-suite/demo.

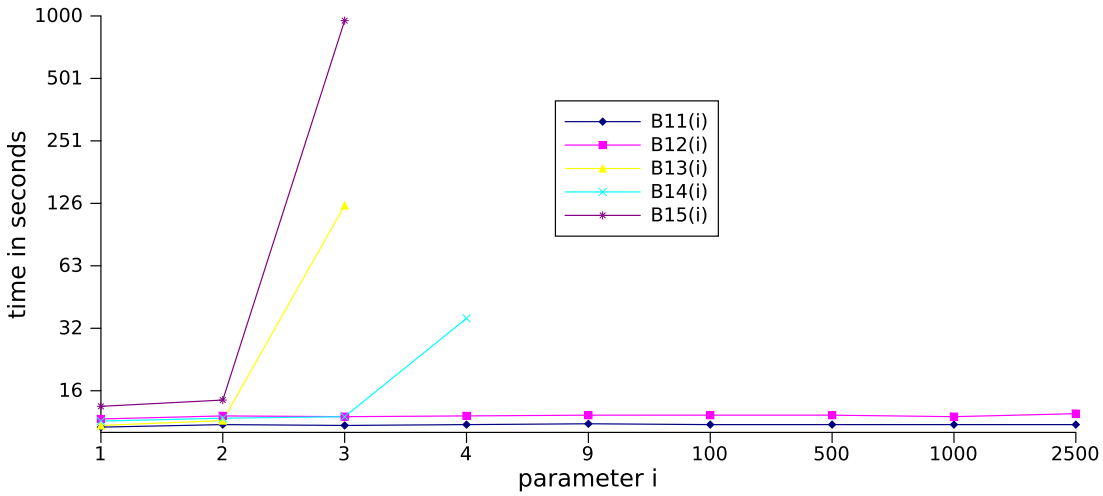

Fig. 19. Runtime for parametric queries B11(*i*)–B15(*i*) of the XPathMark queryset on a 1.1 GB XMark document. When for some parameter *i* no runtime is displayed, then the corresponding eNwa descriptor did not fit into memory (for B14(5)) or it could not be evaluated on-the-fly (for B13(4) and B15(4)).

Another application can be found there, were we apply our streaming methods for querying Twitter streams in Json format. This is another format for data trees, that can be easily converted to Xml on-the-fly in a streaming manner. What is relevant in these applications is the extension of XPath 2.0 by recursive axis.

An important open end of this work is to extend our streaming algorithms such that they can deal with multiple streams, so that one can ask queries with data joins like `$x[@a=$y/@b]`, where `$x` and `$y` refer to nodes selected on two different input streams. Another problem is to integrate these algorithms into a streaming implementation of X-Fun [18], so that multi-stream input can be made available for all QuiX-Tools, in order to obtain hyper-streaming implementations of Xslt, XProc, and XQuery, and XSchematron [17].

References

[1] R. Alur, P. Madhusudan, Visibly pushdown languages, in: 36th ACM Symposium on Theory of Computing, ACM-Press, 2004, pp. 202–211.
[2] R. Alur, P. Madhusudan, Adding nesting structure to words, J. ACM 56 (3) (2009) 1–43.
[3] D.F. Barbieri, D. Braga, S. Ceri, E. Della Valle, M. Grossniklaus, C-SPARQL: a continuous query language for RDF data streams, Int. J. Semantic Comput. 4 (1) (2010) 3–25.
[4] M. Benedikt, W. Fan, F. Geerts, XPath satisfiability in the presence of DTDs, J. ACM 55 (2) (2008) 1–79.
[5] M. Benedikt, A. Jeffrey, Efficient and expressive tree filters, in: Foundations of Software Technology and Theoretical Computer Science, in: LNCS, vol. 4855, 2007, pp. 461–472.
[6] M. Benedikt, A. Jeffrey, R. Ley-Wild, Stream firewalling of XML constraints, in: ACM SIGMOD International Conference on Management of Data, ACM-Press, 2008, pp. 487–498.
[7] M. Fernandez, P. Michiels, J. Siméon, M. Stark, XQuery streaming à la carte, in: 23nd International Conference on Data Engineering, 2007, pp. 256–265.
[8] M. Franceschet, XPathMark: an XPath benchmark for the XMark generated data, in: 3rd International XML Database Symposium, 2005, Revised version available at http://goo.gl/qVugQc.
[9] O. Gauwin, Streaming tree automata and XPath, PhD thesis, Université Lille 1, 2009.
[10] O. Gauwin, J. Niehren, Streamable fragments of forward XPath, in: International Conference on Implementation and Application of Automata, in: LNCS, vol. 6807, 2011, pp. 3–15.
[11] O. Gauwin, J. Niehren, S. Tison, Earliest query answering for deterministic nested word automata, in: International Symposium on Fundamentals of Computer Theory, in: LNCS, vol. 5699, 2009, pp. 121–132.
[12] O. Gauwin, J. Niehren, S. Tison, Queries on XML streams with bounded delay and concurrency, Inform. and Comput. 209 (2011) 409–442.
[13] T.J. Green, A. Gupta, G. Miklau, M. Onizuka, D. Suciu, Processing XML streams with deterministic automata and stream indexes, ACM Trans. Database Syst. 29 (4) (Dec. 2004) 752–788.
[14] A.K. Gupta, D. Suciu, Stream processing of XPath queries with predicates, in: ACM SIGMOD Conference, 2003, pp. 419–430.
[15] M. Kay, A streaming XSLT processor, in: Balisage: The Markup Conference 2010, in: Balisage Series on Markup Technologies, vol. 5, 2010.
[16] O. Kupferman, M.Y. Vardi, Model checking of safety properties, Form. Methods Syst. Des. 19 (3) (2001) 291–314.
[17] P. Labath, J. Niehren, A functional language for hyperstreaming XSLT, Technical report, INRIA Lille, 2013.
[18] P. Labath, J. Niehren, X-Fun: a universal programming language for processing data trees and XML, Research report, Feb. 2014.
[19] D. Le Phuoc, M.D. Tran, J.X. Parreira, M. Hauswirth, A native and adaptive approach for unified processing of linked streams and linked data, in: International Semantic Web Conference (1), in: LNCS, vol. 7031, 2011, pp. 370–388.
[20] C. Ley, M. Benedikt, How big must complete XML query languages be?, in: 12th International Conference on Database Theory, ACM-Press, 2009, pp. 183–200.
[21] D. Debarbieux, O. Gauwin, J. Niehren, T. Sebastian, M. Zergaoui, Early nested word automata for XPath query answering on XML streams, in: CIAA Conference, 2013, pp. 292–305.
[22] D. Debarbieux, O. Gauwin, J. Niehren, T. Sebastian, M. Zergaoui, Early nested word automata for XPath query answering on XML streams, Extension of this TCS article by an Appendix, Available online at http://hal.inria.fr/hal-00966625.
[23] A. Lick, J. Niehren, Early = Earliest?, Technical report, http://hal.inria.fr/hal-00873742, Oct. 2013.
[24] P. Madhusudan, M. Viswanathan, Query automata for nested words, in: 34th International Symposium on Mathematical Foundations of Computer Science, in: LNCS, vol. 5734, 2009, pp. 561–573.

[25] B. Mozafari, K. Zeng, C. Zaniolo, High-performance complex event processing over XML streams, in: SIGMOD Conference, ACM, 2012, pp. 253–264.
[26] D. Olteanu, SPEX: streamed and progressive evaluation of XPath, IEEE Trans. Knowl. Data Eng. 19 (7) (2007) 934–949.
[27] M. Onizuka, Processing XPath queries with forward and downward axes over XML streams, in: 13th International Conference on Extending Database Technology, EDBT, ACM, 2010, pp. 27–38.
[28] F. Peng, S.S. Chawathe, XSQ: a streaming XPath engine, ACM Trans. Database Syst. 30 (2) (2005) 577–623.
[29] M. Schmidt, S. Scherzinger, C. Koch, Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation, in: 23rd IEEE International Conference on Data Engineering, 2007, pp. 236–245.