Early Nested Word Automata for XPath Query Answering on XML Streams

Denis Debarbieux, Olivier Gauwin, Joachim Niehren, Tom Sebastian, Mohamed Zergaoui

Early Nested Word Automata for XPath Query Answering on XML Streams

Denis Debarbieux[1,3], Olivier Gauwin[4,5], Joachim Niehren[1,3]
Tom Sebastian[2,3], and Mohamed Zergaoui[2]

No Institute Given

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 in a highly efficient manner. We show that this approximation can be made tight in practice for automata obtained from XPath expressions. We have implemented an XPath streaming algorithm based on early nested word automata in the Fxp tool. Fxp outperforms most previous tools in efficiency, while covering more queries of the XPathMark benchmark.[a]

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 [12,7,19,20,5,17,13,9,18]. Query answering for XPath is the most 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 giga bytes 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 [19] with 24% of the use cases. The time efficiency of Spex, however, is only average, for instance compared to Gcx [20], which often runs in parsing time without any overhead (since the cpu can work in parallel with file accesses to the stream). We hope that this unsatisfactory situation can be resolved in the near future by pushing existing automata techniques forwards [12,17,9,18].

In contrast to sliding window techniques for monitoring continuous streams [3,15], 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 non-alive elements should be either output or discarded from the buffer. Unfortunately, this kind of *earliest query answering* is not feasible for XPath queries [6], as first shown

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

by adapting counter examples from online verification [14]. A second argument is that deciding aliveness is more difficult than deciding XPath satisfiability [9], 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 [17,10]. Many practical XPath queries (without aggregation, joins, and negation) can be compiled into small NWAs [9], while relying on non-determinism for modeling descendant and following axis. 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, 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 basis of transducer networks [19], Saxon's streaming Xslt engine [13], and Gcx [20] which implements a fragment of XQuery. The recent XSeq tool [18], in contrast, restricts XPath queries by ruling out complex filters all over. In this way, node selection can always be decided with 0-delay [11] 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 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 processing time and buffering space.

As a first contribution, we provide an approximation of the earliest query answering algorithm for NWAs [10,17]. 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. The query answering algorithm then runs eNWAs for all possible candidates while determinizing on-the-fly, and using a new algorithm for sharing the runs of multiple alive candidates. Our stack-and-state sharing algorithm for multi-running eNWAs is original and nontrivial. As a second contribution, we show how to compile XPath queries to eNWAs by adapting the previous translation to NWAs from [9], mainly by distinguishing selection and rejection states. The third contribution is an implementation of our algorithms in the Fxp 1.1

system, that is freely available. It covers 37% of the use cases in XPathMark, while outperforming most previous tools in efficiency. The only exception is Gcx, which does slightly better on some queries, probably due to using C++ instead of Java. Our approximation of earliest query answering turns out to be tight for XPath in practice: it works in an earliest manner in the above example and for all supported queries from XPathMark with only two exceptions, which contain valid or unsatisfiable subfilters (as for the counter examples in [9]).

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 sketches how to compile Fxp to eNwA queries. Section 6 presents our new query answering algorithm for eNwAs with stack-and-state sharing. Section 7 sketches our implementation and exprimental results. We refer to the Appendix of the long version[b] for missing proofs and further details on constructions and experiments.

2 Preliminaries

Nested Words and XML Streams. Let Σ and Δ be two finite sets of tags and internal letters respectively. 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 *nested word* over Δ and Σ is a sequence of internal letters in Δ, opening tags `<a>`, and closing tags `</a>`, where $a \in \Sigma$, that is well nested so that every opening tag is closed properly. Every data tree can be linearized in left-first depth-first manner into a unique nested word. For instance, $l(b(p("ACM"), c(...)), ...)$ becomes `<l><b><p>ACM</p><c>...</c></b>...</l>`. We will restrict ourselves to nested words that are linearizations of data trees. The positions of nested words are called *events*, of which there are three kinds: opening, closing, and internal, depending on the letter at the event. Note also, that every *node* in a data tree corresponds to a pair of an opening event and a corresponding closing event. The correspondence is established by a parser processing a nested word stream.

The Xml data model provides data trees with five different types of nodes: element, text, comment, processing-instruction, and attributes.[c] The latter four are always leaf nodes. Any sequence of children of element nodes starts with a sequence of attribute nodes, followed by a sequence of nodes of the other 4 types. For an Xml data tree t the "child" relation ch^t relates all element nodes to their non-attribute children. Attribute nodes are accessed by the attribute relation $@^t$, which relates all element nodes to their attribute nodes. The "next-sibling" relation ns^t relates non-attributes nodes in t to their non-attribute next-sibling node. In that sense attributes in the Xml data model are unordered. An Xml stream contains a nested word obtained by linearization of Xml data trees.

[b] The long version can be found at http://hal.inria.fr/hal-00676178.

[c] Attributes are nodes of data trees but *not* nodes in terms of the Xml data model.

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. Furthermore, the stack must be empty at the beginning of the stream, and thus it will also be empty at its end. More formally, a nested word automaton is a tuple $A = (\Sigma, \Delta, Q, I, F, \Gamma, \mathit{rul})$ where Σ and Δ are the finite alphabets of nested words, Q a finite set of states with subsets $I, F \subseteq Q$ of initial and final states, Γ a finite set of stack symbols, and rul is a set of transition rules of the following three types, where $q, q' \in Q$, $a \in \Sigma$, $d \in \Delta$ and $\gamma \in \Gamma$:

(open) $q \xrightarrow{\langle a \rangle : \gamma} q'$ can be applied in state q, when reading the opening tag `<a>`. In this case, γ is pushed onto the stack and the state is changed to q'.

(close) $q \xrightarrow{\langle /a \rangle : \gamma} q'$ can be applied in state q when reading the tag `</a>` with γ on top of the stack. In this case, 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 $Q \times \Gamma^*$. A run of an NWA on a nested word over Σ and Δ must start in a configuration with some initial state and the empty stack, and then rewrites this configuration on all events of the nested word according to some rule. A run is successful if it can be continued until the end of the nested word, while reaching some final state. Note that the stack must be empty then. The *language* $\mathcal{L}(A)$ of an NWA A is the set of all data trees with some successful run on their linearization. See Figs. 1 and 2 for an example of an NWA and a successful run on the nested word of a data tree.

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. In the worst case, the resulting automata may have $2^{|Q|^2}$ states. In experiments, we also observed huge size explosions in the average case. Therefore, we will mostly rely on on-the-fly determinization.

Automata Queries. We restrict ourselves to monadic (node selection) queries for data trees with fixed alphabets Σ and Δ. A monadic query over these alphabets is a function P that maps all data trees t over these alphabets to some subset $P(t)$ of nodes of t. We will use NWAs to define monadic queries (as usual for showing that tree automata capture MSO queries). The idea is that an NWA should only test whether a candidate node is selected by the query on a given tree, but not generate the candidate by itself. Therefore, a unique candidate node is assumed to be annotated on the input tree by some external process. We fix a single variable x for annotation and set the tag alphabet of such NWAs to $\{a, a^x \mid a \in \Sigma\}$. Letters a^x are called annotated (or "starred" in the terminology of [17]) while letters a are not. A monadic query P can then be defined by all NWAs that recognize the set of all trees t annotated at some single node belonging to $P(t)$. An example for a deterministic NWA is given in Fig. 1, while

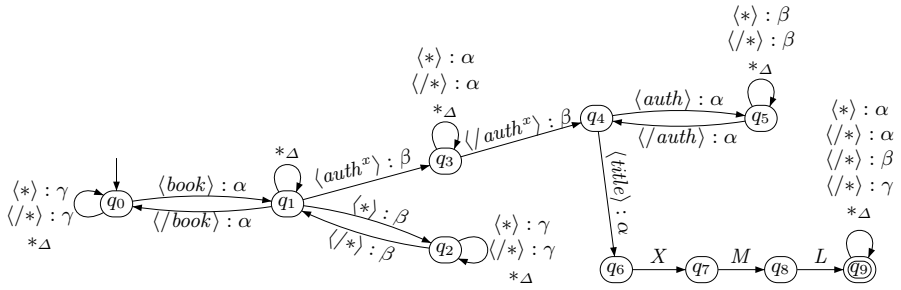

Fig. 1. An NWA for XPATH query `//book[starts-with(title,'XML')]/auth`, which selects all authors of book nodes having a title that starts with "XML". It runs on well-formed libraries as in the introduction, where the children of book nodes contain the sequence of authors followed by the title. We add the special symbol $*$ to Σ that captures all infinitely many other tags non-specified at the same state, and similarly a special symbol $*_\Delta$ to Δ that captures all other internal letters not mentionned there.

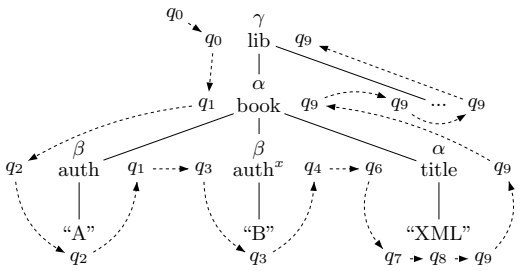

Fig. 2. An example run of the NWA in Fig. 1.

a successful run of this NWA is depicted in Fig. 2, on a library in which the second author of the first book is annotated by x.

Earliest Query Answering. Let P be a query, t a data tree with node π, and e an event of the nested word of t such that the opening event of π is before or equal to e. We call π *safe for selection at* e if π is selected for all data trees t' ($\pi \in P(t')$) whose nested word is a possible continuation of the stream of t at event e, i.e., of the prefix of the nested word of t until e. We call π *safe for rejection at* e if π is rejected for all data trees t' ($\pi \notin P(t')$) such that the nested word of t' is a possible continuation of the stream 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. For dNWAs it can be reduced to the reachability problem of pushdown machines which is in cubic time [9]. 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[d], 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 NWAs.

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'. It is easy to formalize this condition in terms of necessary and impossible transition rules of A.

Definition 1. *An* early nested word automaton *(*ENWA*) is a triple $E = (A, S, R)$ where A is an* NWA*, S is an attractor of A of final states called selection states, and R an attractor of non-final states called rejection states. The* query defined by *E is the query defined by A.*

In the example NWA in Fig. 1, we can define $S = \{q_9\}$ and $R = \emptyset$. We could add a sink state to the automaton and to the set of rejection states. Also all selection and respectively rejection states can be merged into a single state.

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 rejecting 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.

We call an ENWA deterministic if the underlying NWA is. We next lift the determinization procedure for NWAs to ENWAs. Let $E = (A, S, R)$ be an ENWA and A' the determinization of A. The deterministic ENWA $E' = (A', S', R')$ is defined such that S' contains the pair sets, for which *some* pair has the second component in S, while R' contains the pair sets, for which *all* pairs have a second component in R. From the construction of A' it is not difficult to see that S' and R' are attractors of A'. Notice that the selection delay is preserved by ENWA determinization, so that we can decide whether all runs of E reach a rejection state at event e, by running the determinized version until event e.

[d] The semantics of selection states is identical with the semantics of final states in the acceptance condition for NWAs in [2]. The idea of analogous rejection states, however, is original to the present paper to the best of our knowledge.

Formulas	$\mathrm{F} ::= \mathrm{F} \wedge \mathrm{F} \mid \mathrm{F} \vee \mathrm{F} \mid \neg\mathrm{F} \mid \mathit{true} \mid \mathrm{A(F)} \mid \mathrm{L(F)} \mid \mathrm{K(F)} \mid O(\mathrm{T,s})$
Axes	$\mathrm{A} ::= @ \mid \mathit{ch} \mid \mathit{ch}^+ \mid \mathit{ch}^* \mid \mathit{ns} \mid \mathit{fs} \mid \mathit{fo}$
Labels	$\mathrm{L} ::= x \mid a \mid \mathit{nsp}_a$
Types	$\mathrm{K} ::= \mathit{element} \mid \mathit{text} \mid \mathit{comment} \mid \mathit{processing\text{-}instruction}$
Comparisons	$O ::= \mathit{equals} \mid \mathit{contains} \mid \mathit{starts\text{-}with} \mid \mathit{ends\text{-}with}$
Texts	$\mathrm{T} ::= \mathit{text}_x(\mathrm{F})$

Fig. 3. Abstract syntax of FXP where $x \in \mathcal{V}$ is a variable, $a \in \Sigma$ is a label, attribute, or namespace, and $s \in \Delta^*$ a string data value.

Lemma 1. *For any event e of the stream of a tree t, there exists a run of E going into S at event e if and only if there is a run of E' going into S' at e. Likewise all runs of E go into R at event e iff all runs of E' go into R' at e.*

4 FXP Logic

Rather than dealing with XPATH expressions directly, we first compile a fragment of XPATH into the hybrid temporal logic FXP [9]. Even though the translation from XPATH to FXP is mainly straightforward, it leads to a great simplification, mainly due to the usage of variables for node selection. We are going to compile a larger fragment of XPATH than previously [9], since supporting node types, attributes, strings data values and patterns, and all forward axes of XPATH, we also need to extend FXP accordingly. The XPATH query `//book[starts-with(title,'XML')]/auth`, for example, will be compiled to the following FXP formula with one free variable x:

$$\mathit{ch}^*(\mathit{book}(\mathit{starts\text{-}with}(\mathit{text}_y(\mathit{ch}(\mathit{title}(y(\mathit{true})))), XML) \wedge \mathit{ch}(\mathit{auth}(x(\mathit{true})))))$$

FXP formulas will talk about data trees t satisfying the XML data model based on its typed relations: attribute $@^t$, child ch^t, descendant $(\mathit{ch}^+)^t = (\mathit{ch}^t)^+$, descendant-or-self $(\mathit{ch}^*)^t = (\mathit{ch}^t)^*$, next-sibling ns^t, following-sibling $\mathit{fs}^t = (\mathit{ns}^t)^*$, and following $\mathit{fo}^t = ((\mathit{ch}^t)^*)^{-1} \circ (\mathit{ns}^t)^+ \circ (\mathit{ch}^t)^*$. The abstract syntax of FXP formulas with alphabets Σ, Δ and a set of variables $\mathcal{V}$ is given in Fig. 3. There is a single atomic formula *true*. A non-atomic formula can be constructed with the usual boolean operators, or be a test for a variable $x(F)$, a node label $a(F)$, a namespace $\mathit{nsp}_a(F)$, or an XML node type $K(F)$. There are also formulas $A(F)$ for navigating with any typed relation A^t supported by the XML data model. Finally there are various comparisons $O(T, s)$ between string data values $\mathit{text}_y(F)$ accessed from the y-node in the data tree and string constants $s \in \Delta^*$, but no more general comparisons as needed for join operations. The formal semantics of FXP is defined in Fig. 4. Given an XML data tree t, a node π of t, and a variable assignment μ to nodes of t, a formula F evaluates to a truth value $[\![F]\!]_{t,\pi,\mu}$. Formulas F with one free variable define monadic queries. For compiling XPATH, we restrict ourselves to formulas where all subformulas contain at most one free variable. Also there may be some bound variables y introduced by $\mathit{text}_y(F)$.

$[\![F_1 \wedge F_2]\!]_{t,\pi,\mu} \Leftrightarrow [\![F_1]\!]_{t,\pi,\mu} \wedge [\![F_2]\!]_{t,\pi,\mu}$
$[\![F_1 \vee F_2]\!]_{t,\pi,\mu} \Leftrightarrow [\![F_1]\!]_{t,\pi,\mu} \vee [\![F_2]\!]_{t,\pi,\mu}$
$[\![\neg F]\!]_{t,\pi,\mu} \Leftrightarrow \neg\, [\![F]\!]_{t,\pi,\mu}$
$[\![true]\!]_{t,\pi,\mu} \Leftrightarrow true$
$[\![A(F)]\!]_{t,\pi,\mu} \Leftrightarrow \exists \pi' \in [\![A]\!]_{t,\pi,\mu}\, s.t.\, [\![F]\!]_{t,\pi',\mu}$
$[\![L(F)]\!]_{t,\pi,\mu} \Leftrightarrow [\![F]\!]_{t,\pi,\mu} \wedge [\![L]\!]_{t,\pi,\mu}$
$[\![K(F)]\!]_{t,\pi,\mu} \Leftrightarrow [\![F]\!]_{t,\pi,\mu} \wedge [\![K]\!]_{t,\pi,\mu}$

$[\![O(T,s)]\!]_{t,\pi,\mu} \Leftrightarrow O([\![T]\!]_{t,\pi,\mu}, s)$
$[\![A]\!]_{t,\pi,\mu} = A^t(\pi)$
$[\![x]\!]_{t,\pi,\mu} \Leftrightarrow \pi = \mu(x)$
$[\![a]\!]_{t,\pi,\mu} \Leftrightarrow$ label of π in t is a
$[\![nsp_a]\!]_{t,\pi,\mu} \Leftrightarrow$ namespace of π in t is a
$[\![K]\!]_{t,\pi,\mu} \Leftrightarrow$ π has type K in t
$[\![text_x(F)]\!]_{t,\pi,\mu} =$ data value of $\mu(x)$: $[\![F]\!]_{t,\pi,\mu}$

Fig. 4. Semantics of FXP formulas F for an XML data tree t with node π and variable assignment μ to nodes of t.

5 Compiler from FXP to Early Nested Word Automata

We sketch a compiler from FXP formulas to ENWAs, that follows the usual approach of compiling tree logics such as MSO into tree automata. Compared to previous compilers into NWAs in [10,17], the most novel part is the distinction of appropriate selection and rejection states. It should also be noticed, that our compiler will heavily rely on non-determinism, in order to compile formulas $A(F)$ where A is a recursive axis such as the descendant or following axis. However, we will try to preserve determinism as much as possible, so that we can compile many formulas $\neg F$ without having to determinize the ENWA for F.

The construction is by recursion on the structure of F. Given an FXP formula F with n free variables, the compiler produces an ENWA with node labels in $\Sigma \times 2^{\mathcal{V}}$ that defines the same n-ary query. With $n = 1$ as for FXP formulas obtained from XPATH, this yields ENWAs defining monadic queries by identifying $\Sigma \times 2^{\{x\}}$ with $\{a, a^x \mid a \in \Sigma\}$. Let F and F' be two formulas that were compiled to $E = (A, S, R)$ and $E' = (A', S', R')$ with state sets Q and Q' respectively. The NWA for a **conjunction** $F \wedge F'$ is the product of A and A'. We choose selection states $S \times S'$, since a node is safe for selection for $F \wedge F'$ iff it is safe for selection for both F and F'. As rejection states we choose $(R \times Q') \cup (Q \times R')$, which may lead to a proper approximation of earliest query answering. Also a large number of conjunctions may lead to an exponential blow-up of the states. The NWA of a **disjunction** $F \vee F'$ is the union of A and A'. As selection states we use $S \cup S'$ which is exact, and as rejection states $R \cup R'$. Note that we compile conjunctions and disjunctions differently, since unions may introduce non-determinism while products do not. For **negations** $\neg F$, where E is deterministic, we simply swap the final states of E, and exchange selection and rejection states. This is correct since we maintain pseudo-completeness as an invariant (see [9]), and remains exact, since a node is safe for selection for $\neg F$ iff it is safe for rejection for F, and conversely. Otherwise, we determinize E in a first step, which is exact by Lemma 1, and second apply the previous construction. The ENWAs for **navigation** formulas $A(F)$ for the various axes A guess an A-successor of the root and then run E starting from there. The selection and rejection states remain unchanged except for the treatment of the root. A better

construction preserving determinism is available for formulas $ch(F)$ under the condition that F contains only the axis ch and ch^* (see [9] again). There is also an optimized construction for **attribute** access $@(F)$ which uses internal transitions only. Further optimizations are possible based on **node typing** of the XML data model (such as that attribute children precede all children of other node types). Node **label** and **type** testers $L(F)$ and $K(F)$ work as usual. They may add new rejection states to R while preserving the selection states S. ENWAs for **string comparisons** at the root $O(text_y(y), s)$ can be obtained from a DFA with accepting and rejecting states that recognizes all strings s' such that $O(s', s)$. General string comparisons $O(text_y(F), s)$ can be reduced to the previous case, since they are equivalent to $F[y/O(text_z(z), s)]$.

6 Early Query Answering

We show how to use ENWAs for evaluating monadic queries on XML streams. Our basic algorithm generates all possible answer candidates, and runs the ENWA on them based on on-the-fly determinization. We then improve this algorithm so that configurations and runs of multiple answer candidates may be shared.

On-the-fly Determinization. Let E be an ENWA that defines a monadic query, i.e., with tag alphabet $\{a, a^x \mid a \in \Sigma\}$ where x is a fixed variable. Rather than running E, we want to run its determinization E'. This can be done while generating E' on the fly. At any time point, we store the subset of the states and transitions of E' that was used before. If a missing transition rule is needed then one we compute it from E and adds it to E'. It should be noticed that each transition can be computed in polynomial time (but not in linear time). Recall also that the states of E' are 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 E was deterministic at beforehand. Therefore, we will assume in the sequel that E is deterministic. We will also assume that it is pseudo-complete, so that runs can never get blocked.

Buffering Possibly Alive Candidates. Suppose we are given a stream containing a nested word of some data tree, and that we want to compute the answer 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 selecting nor rejecting, 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 bullet candidate (which then is unique). Consider the `<a>` event of

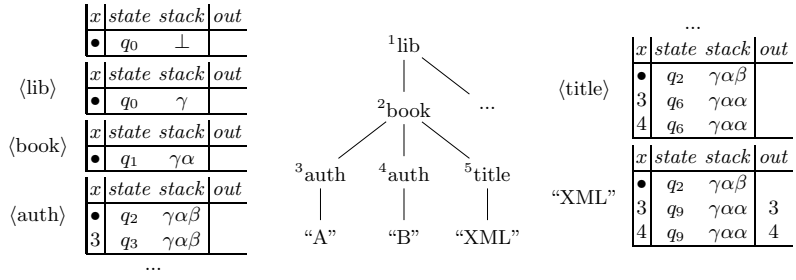

Fig. 5. Evolution of the buffer for the eNwa from Fig. 1 when answering the XPath query `//book[starts-with(title,'XML')]/auth` on a sample document.

some node π and let $(\bullet, q, S)$ be the bullet candidate 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 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 selecting 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. 5 on the eNwa from Fig. 1 and the document from Fig. 2. Initially the buffer contains a single candidate with an unknown node $\bullet$, that starts in the initial state of the eNwa with an empty stack. According to opening tags $\langle lib \rangle$ and $\langle book \rangle$ we launch open transitions and apply state and stack changes. At the opening event of node 3, i.e. when reading the open tag $\langle auth \rangle$ 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.

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 $\bullet$ candidate. It may happen though that the number of candidates grows linearly with the size of the document. An example is the XPath query `/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.

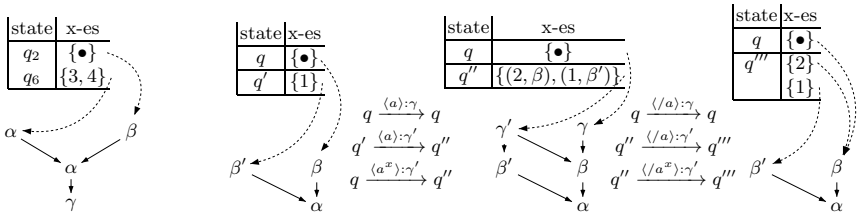

Fig. 6. Buffer of `<title>`

Fig. 7. Data structures for the state sharing algorithm.

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 in the obvious manner. In addition, we use a table $B : Q \times \Gamma \to \mathit{Aggreg}$ relating a state and a root of the DAG through an aggregation of nodes or $\bullet$. The shared representation of the buffer at the `<title>`-event in Fig. 5 is illustrated in Fig. 6 for instance. Here we have $B(q_6, \alpha) = \{3, 4\}$, $B(q_2, \beta) = \{\bullet\}$. In this case, the aggregations are set of candidate nodes or the $\bullet$, but this will not be enough in general (see example below). Whenever a selection state is reached in the B-table, the nodes 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 nodes get deleted entirely from the data structure this way, since no node may appear twice in different aggregates, again due to determinism.

The precise functioning of our DAG-based buffering is illustrated by example in Fig. 7. There one has to store enough information when sharing at opening events, so that one can undo the sharing properly at closing events. From the first configuration, we reach the second with the `<a>` event for node 2, for which a new candidate will be buffered. This candidate 2 will be 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 for node 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 $B(q'', \gamma') = \{(2, \beta), (1, \beta')\}$. The next event has the letter `</a>`, where we have to undo the sharing. Now we decompose the aggregate, to update the data structure to $B(q''', \beta) = \{2\}$, $B(q''', \beta') = \{1\}$ and $B(q, \beta) = \{\bullet\}$.

Theorem 1. *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 to compute* $P(t)$ *is in* $\mathcal{O}(|E| + |Q|\ |t|)$ *and its space complexity in* $O(|E| + \mathit{depth}(t)\ |Q| + C)$*, where* C *is the maximal number of buffered candidates of* P *on* t *at any event.*

	A1	A2	A3	A4	A5	A6	A7	A8	B1	B2	B3	B4	B5	B6	B7	B11	B12	B13	B14	B15
Fxp	**2.7**	2.5	2.4	**2.3**	**3.5**	**3.4**	**3.4**	2.4	**2.8**	**2.2**	**3.3**	**3.7**	**2.1**	**2.4**	1.9	**1.9**	**2.2**	**2.2**	**2.0**	**1.6**
Spex	0.7	1.5	1.1	0.9	0.9	0.9	0.8	0.9	0.9	1.1	0.4	0.8	–	–	–	–	–	0.6	–	–
Saxon	1.7	1.8	1.8	–	–	1.6	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Gcx	2.5	**3.0**	**2.9**	–	–	–	–	**3.3**	–	–	–	–	–	2.3	**3.3**	–	–	–	–	–

Table 1. Throughput on XPathMark queries in millions of events per second.

7 Implementation and Experimental Results

The Fxp tool is released under the version 1.1 and is available under the GPL licence at http://fxp.lille.inria.fr. A compiler from XPath to Fxp is freely available in the QuiXPath tool at http://code.google.com/p/quixpath. It covers a slightly larger fragment of XPath than discussed here. In particular it supports top-level aggregations, which are reduced to earliest query answering for n-ary queries (and not only monadic queries). QuiXPath also supports backwards axes such as Spex. We eliminated them at the cost of forward axis and regular closure. As noticed in [16] conditional regular axes are not enough.

We also implemented the static determinization algorithm for Nwas, which explodes for most practical queries, even if restricted to accessible states, but do not need it for evaluating the queries of the XPathMark benchmark. In contrast, on-the-fly determinization explores only small fragments of the determinized Nwas. One should also mention that we obtain high efficiency results also due to projection, where parts of the input documents are projected according to the content of the query. This precise projection algorithm is new and of interest but out of scope of this present paper.

We tested our system against the revised[e] version of XPathMark query set [8]. It turns out that all queries are answered in an earliest manner with two exceptions, that use valid or unsatisfiable subfilters. The query from the introduction is also treated in an earliest manner, so Fxp improves on Spex in this respect. We have also compared our Fxp tools to various systems on XPathMark, such as Spex, Saxon, and Gcx. Input documents were produced by the XMark generator. We give in Table 1 a collection of XPath queries, where we report for each system the *throughput* obtained on a 1.1GB XMark file. There "–" states that the query was not supported. Notice however that the Gcx system competes very well. Nevertheless we believe that we obtain good results with respect to that the Gcx system was done in C++, in contrast to Fxp, developed in Java.

Conclusion and Future Work. We have shown how to approximate earliest query answering for XPath on Xml streams by using eNwas. An implementation of our algorithms is freely available in the Fxp system. Our practical solution outperforms existing algorithms in performance and coverage. In follow-up work, we extended the coverage of our XPath fragment by aggregate queries, arithmetic operations, and float comparisons. For this we propose *networks of*

[e] http://users.dimi.uniud.it/~massimo.franceschet/xpathmark/index.html

automata registrations, such that each of them can evaluate one subquery in a query decomposition. For future work we hope that we can extend this approach to cover database joins, and thereby reach over 90% coverage of XPathMark.

References

1. R. Alur and P. Madhusudan. Visibly pushdown languages. In *36th ACM Symposium on Theory of Computing*, pages 202–211. ACM-Press, 2004.
2. R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009.
3. D. Barbieri, D. Braga, S. Ceri, E. Della Valle, M. Grossniklaus. C-SPARQL: a continuous query language for RDF data streams. *Semantic Computing*, 4(1):3–25, 2010.
4. M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM*, 55(2):1–79, 2008.
5. M. Benedikt and A. Jeffrey. Efficient and expressive tree filters. In *FSTTCS*, vol. 4855 of *LNCS*, pages 461–472. 2007.
6. M. Benedikt, A. Jeffrey, and R. Ley-Wild. Stream Firewalling of XML Constraints. In *ACM SIGMOD*, pages 487–498. 2008.
7. M. Fernandez, P. Michiels, J. Siméon, and M. Stark. XQuery streaming à la carte. In *ICDE*, pages 256–265, 2007.
8. M. Franceschet. XPathMark: An XPath benchmark for the XMark generated data. In *3rd International XML Database Symposium*, 2005.
9. O. Gauwin and J. Niehren. Streamable fragments of forward XPath. In *CIAA*, vol. 6807 of *LNCS*, pages 3–15. 2011.
10. O. Gauwin, J. Niehren, and S. Tison. Earliest query answering for deterministic nested word automata. In *FCT*, vol. 5699 of *LNCS*, pages 121–132. 2009.
11. O. Gauwin, J. Niehren, and S. Tison. Queries on XML streams with bounded delay and concurrency. *Information and Computation*, 209:409–442, 2011.
12. A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. *SIGMOD Conference*, pages 419–430. 2003.
13. M. Kay. A streaming XSLT processor. In *Balisage: The Markup Conf.*, vol 5, 2010.
14. O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
15. D. Le Phuoc, M. D. Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *International Semantic Web Conference (1)*, vol. 7031 of *LNCS*, pages 370–388, 2011.
16. C. Ley and M. Benedikt. How Big Must Complete XML Query Languages Be? In *12th International Conference on Database Theory*, pages 183–200. 2009.
17. P. Madhusudan and M. Viswanathan. Query automata for nested words. In *MFCS*, vol. 5734 of *LNCS*, pages 561–573. 2009.
18. B. Mozafari, K. Zeng, and C. Zaniolo. High-performance complex event processing over XML streams. In *ACM SIGMOD*, pages 253–264. 2012.
19. D. Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. on Know. Data Eng.*, 19(7):934–949, 2007.
20. M. Schmidt, S. Scherzinger, C. Koch. Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In *ICDE*, 2007.

21. T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, Dec. 2004.
22. M. Onizuka. Processing XPath queries with forward and downward axes over XML streams. EDBT '10, pages 27–38, New York, NY, USA, 2010. ACM.
23. F. Peng and S. S. Chawathe. XSQ: A streaming XPath engine. *ACM Transactions on Database Systems*, 30(2):577–623, 2005.