# Complexity of Earliest Query Answering with Streaming Tree Automata

Olivier Gauwin, Anne-Cécile Caron, Joachim Niehren, Sophie Tison

# Complexity of Earliest Query Answering with Streaming Tree Automata

Olivier Gauwin Anne-Cécile Caron Joachim Niehren Sophie Tison

Mostrare Project, INRIA Futurs at LIFL (UMR 8022 of CNRS), Lille, France

## Abstract

We investigate the complexity of earliest query answering for $n$-ary node selection queries defined by streaming tree automata (STAs). We elaborate an algorithm that selects query answers upon reception of the shortest relevant prefix of the input tree on the stream. In general, deciding if a prefix is sufficient for the selection of a $n$-tuple is DEXPTIME-complete (even for $n = 0$). For queries defined by deterministic STAs, this decision problem is in polynomial time combined complexity, as implemented in our earliest query answering algorithm.

***Keywords*** streaming, database theory, tree automata, XML, query languages

## Introduction

Streaming algorithms process data collection that are exchanged over data streams. For large data sizes, memory management becomes the main issue. Only needed fragments of input and output data streams can be kept in memory. Therefore, output data should be produced incrementally as early as possible, while processing input data in parallel.

Most XML processing tasks are relevant for streaming, since XML has become the standard format of today's data exchange. These problems include XML document validation (19), typing (13), and query answering for various XML query languages. Streaming for XPath fragments is often feasible and generally well understood (12; 4; 17). XPath algorithms can be lifted to the larger XQuery language (18) but only to limited extend.

Consider, for instance, an XML document that represents a library of books. When querying for all books that have at least two authors in XPath (//book[authors[count(author) $\geq$ 2]]), one can always decide membership of a book node to the answer set after having inspected the children of its authors child of every book. This way, the query can be answered while keeping at most one book subdocument in memory at every time. This kind of early query answering is essential for streaming, but unfeasible in the presence of XQuery's blocking operations. For instance, sorting the library by authors requires to input all books, before the first book of the sorted library can be output.



*PLAN-X '08* January 2008, San Francisco.



In XQuery or XPath2.0, $n$-ary node selection queries are frequently expressed by composing a set of XPath queries via variables (3; 9). Answering $n$-ary node selection queries in a streaming fashion is an interesting intermediate task (11; 7). Queries of this type define $n$-tuples of nodes in XML documents for which early selection is meaningful, and particularly relevant since the number of answer candidates is exponential in $n$. Selecting $n$-tuples of nodes as early as possible is a necessary condition for optimal memory management. An analogous statement for a fragment of XPath was proved formally in (4).

In this paper, we define formally the framework of earliest query answering for $n$-ary node selection queries in the presence of schema for the input document. For a given $n$-ary query $q$ and tree $t$, this corresponds to the problem of determining if a node $x$ of $t$ is optimal for deciding the membership of some node tuple $(y_1, \ldots, y_n)$ to the answer set $q(t)$. In other words, is $x$ the first node in document order such that any valid following of the current prefix will keep selecting $(y_1, \ldots, y_n)$?

In the general case where queries may be defined by nondeterministic tree automata, we show that it becomes DEXPTIME-hard to decide earliest query answering, depending on the sizes of the tree automaton $A$ and $t$.

We use streaming tree automata (STAs) (10), a reformulation of nested word automata (1) that operate directly on unranked trees. Queries defined by an STA have the same expressiveness as formulas of monadic second-order logic (MSO) with $n$ free variables. If $q$ is defined by an STA $A$, we show how to construct another STA detecting optimal positions, thus proving MSO-definability of the "optimality" query.

As last contribution we answer complexity questions left open previously. We start with a concrete algorithm for earliest query answering for $n$-ary queries $q$ defined by some STA $A$ that extends the above automaton construction. If $A$ is deterministic, our algorithm decides for a given tree $t$ the optimality of an opening or closing event at some node $x$ of $t$ for some node tuple $(y_1, \ldots, y_n)$ of $t$ in polynomial time, depending on the number of nodes of $t$ preceeding $x$ in document order and the size of automaton $A$. Our algorithm outputs all $n$-tuples $(y_1, \ldots, y_n)$ at their optimal event, either opening or closing $x$, and refutes membership of all other $n$-tuples at the optimal event when no valid continuation can keep selecting this candidate. Based on determinization of $A$ in deterministic exponential time $O(2^{|A|^2})$, and applying the above algorithm, it follows that earliest query answering is DEXPTIME-complete in the general case.

*Related work.* Kumar, Madhusudan and Viswanathan (13) investigate earliest query answering by nested word automata, but for a restricted class of monadic queries which allow for immediate node selection at opening time. Benedikt and Jeffrey (5) consider immediate node selection at opening and closing time, for filters

expressed in an XPath dialect with a restriction to depth bounded documents.

Berlea's (6) earliest query answering algorithm via preorder automata (originating from pushdown forest automata) is closest to ours. It has the advantage to work in quadratic time, but runs on more restricted queries by forest grammars which are purely top-down devices. Furthermore, his polynomial time results depend on the assumption of an infinite label set, and a particular form of automata rules with wildcard, so that universality of such automata can be decided in linear time, without presupposing determinism.

Olteanu (17) presents an earliest query answering algorithm for *Forward XPath*. For every node of the input document, this algorithm finds the optimal event in the input stream (opening or closing some other node) from which it is safe to select or unselect the current node, independently of how the content of the stream will evolve in the future. This algorithm decides for every node and subsequent event, whether the event is optimal for deciding membership of a node to the answer set. It does it in polynomial time depending of the size of the input stream and the size of the query.

Bar-Yossef, Fontoura and Josifovski (4) prove a lower space bound for streaming query answering that applies to monadic queries defined in Forward XPath. This bound is expressed in terms of the concurrency of the query for the document, which is the maximal number of closed answer candidates, for which selection depends on the possible futures of the document on the input stream.

# 1. Earliest Query Answering with Schemas

## 1.1 Trees, traversals, events, and prefixes

Let an *alphabet* $\Sigma$ be a finite set. An *unranked tree* $t \in T_\Sigma$ is either a constant $a \in \Sigma$ or a pair $a(t_1, \ldots, t_k)$, where $a \in \Sigma$, and $(t_1, \ldots, t_k)$ is a sequence of unranked trees in $T_\Sigma$ with $k \geq 0$. The set of *nodes* of a tree $t$ is defined by $\mathtt{nodes}(a) = \{\epsilon\}$ for constant trees $a$ and $\mathtt{nodes}(a(t_1, \ldots, t_k)) = \{\epsilon\} \cup \{i.\pi \mid \pi \in \mathtt{nodes}(t_i)\}$ otherwise. The *root* of a tree is the empty word, written $\epsilon$. We denote by $\mathtt{label}^t(\pi)$ the label of node $\pi$ in tree $t$.

To every tree $t$, we associate a totally ordered set of events produced by the pre-order traversal over $t$:

$$\mathtt{events}(t) = \{\mathtt{start}\} \cup (\{\mathtt{open}, \mathtt{close}\} \times \mathtt{nodes}(t))$$

The set of events consists in an initial event and an opening and a closing event for every node. Let $\prec^t$ be the total order on $\mathtt{events}(t)$, and for every $e \in \mathtt{events}(t) - \{\mathtt{start}\}$ let $\mathtt{pred}(e)$ be the immediate predecessor of $e$ in that order. Pre-order traversals equally define a total order $<^t$ on $\mathtt{nodes}(t)$ which satisfies $\pi <^t \pi'$ if $(\mathtt{open}, \pi) \prec^t (\mathtt{open}, \pi')$ for all $\pi, \pi' \in \mathtt{nodes}(t)$. This order is frequently called the document order of $t$.

For every $e \in \mathtt{events}(t) - \{\mathtt{start}\}$, we define the fragment $t^{\leq e}$ of $t$ to be the tree which contains all nodes of $t$ opened before $e$:

$$\mathtt{nodes}(t^{\leq e}) = \{\pi \in \mathtt{nodes}(t) \mid (\mathtt{open}, \pi) \preceq^t e\}$$

and satisfying $\mathtt{label}^{t^{\leq e}}(\pi) = \mathtt{label}^t(\pi)$ for all $\pi \in \mathtt{nodes}(t^{\leq e})$. Note that $t^{\leq (\mathtt{close}, \pi)}$ contains all proper descendants of $\pi$ in $t$, while $t^{\leq (\mathtt{open}, \pi)}$ does not.

For trees $t, t' \in T_\Sigma$ and $e \in \mathtt{events}(t)$ we define $\mathtt{equal}_e(t, t')$ by $e \in \mathtt{events}(t) \cap \mathtt{events}(t')$ and $t^{\leq e} = t'^{\leq e}$.

## 1.2 Queries

We are interested in $n$-ary node selection queries $q$ in unranked trees where $n \geq 0$. Such queries select a set of $n$-tuples of nodes for every tree $t \in T_\Sigma$:

$$q(t) \subseteq \mathtt{nodes}(t)^n$$

Boolean queries are subsumed by the case $n = 0$. Note that the output of an $n$-ary query is not a tuple of serialized nodes (including their descendants as in XPath semantics) but a set of $n$-tuples of identifiers of nodes.

We identify $n$-ary query $q$ in trees of $T_\Sigma$ with their canonical language of annotated trees $\mathtt{Can}_q \subseteq T_{\Sigma \times \mathbb{B}^n}$, where $\mathbb{B} = \{0, 1\}$ is the set of Booleans. For every tree $t \in T_\Sigma$ and tuple $\sigma \in \mathtt{nodes}(t)^n$, we define a characteristic tree $t' = \mathtt{ch}(t, \sigma)$ in $T_{\Sigma \times \mathbb{B}^n}$ that has the same structure as $t$, i.e., $\mathtt{nodes}(t') = \mathtt{nodes}(t)$, while annotating the node labels of $t$ by bit vectors, such that $\mathtt{label}^{t'}(\pi) = (\mathtt{label}^t(\pi), v)$ for all nodes $\pi \in \mathtt{nodes}(t)$, where $v = (b_1, \ldots, b_n)$, $\sigma = (\pi_1, \ldots, \pi_n)$ and $b_i = 1 \Leftrightarrow \pi = \pi_i$ for all $1 \leq i \leq n$. The canonical language of an $n$-ary query $q$ is the set of all characteristic trees for this query:

$$\mathtt{Can}_q = \{\mathtt{ch}(t, \sigma) \mid \sigma \in q(t)\}$$

A variant of Thatcher and Wright's theorem (22) for unranked trees (16) shows that a query $q$ is MSO-definable iff its canonical language $\mathtt{Can}_q$ is recognizable by a tree automaton for unranked trees.

Throughout this paper, we will deal with MSO definable $n$-ary queries $q$ in trees of $T_\Sigma$. These will be represented by tree automata $A$ over $\Sigma \times \mathbb{B}^n$ which recognize the canonical language of $q$, i.e., $L(A) = \mathtt{Can}_q$. The representation language thus contains all tree automata over $\Sigma \times \mathbb{B}^n$ whose languages are canonical, i.e. all trees $t \in L(A)$ are characteristic, in that for $1 \leq i \leq n$ there exists exactly one $\pi \in \mathtt{nodes}(t)$ such that $\mathtt{label}^t(\pi) = (a, b_1, \ldots, b_n)$ with $b_i = 1$. We write $q(A)$ for the query represented by some automaton $A$ of this class.

At the time being, we have not specified a concrete tree automata notion. The setting presented so far is broadly applicable: either to standard tree automata for binary trees, to hedge automata for unranked trees, or stepwise tree automata for unranked trees (8). In section 3, we define Streaming Tree Automata (STAs), a class of automata for unranked trees we use for earliest query answering.

The size of an automaton is the number of its states and rules, i.e., $|A| = |\mathtt{states}(A)| + |\mathtt{rules}(A)|$. The size of the signature of $A$ is not taken into account.

In the sequel, we write $a_v$ for the annotated node label $(a, v) \in \Sigma \times \mathbb{B}^n$.

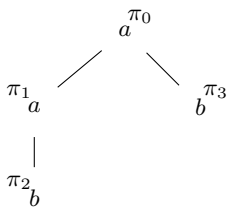
## 1.3 Optimal Events

Let us introduce optimal events with the example of the monadic query $q_0$ that selects all nodes without next siblings, which can be defined by the MSO-formula $\neg\exists y.\mathtt{next\_sibl}(x, y)$. A streaming query answering algorithm reads the events of some tree $t$ in document order from the input stream, and writes the selected nodes in $q_0(t)$ to the output stream.

The root of $t$ is selected and can be output, once opened. Without any schema, membership $\pi \in q_0(t)$ cannot be decided for all other nodes $\pi \in \mathtt{nodes}(t) - \{\epsilon\}$ at opening time, so the algorithm needs to memorize these nodes until encountering the opening event of the next sibling of $\pi$ ($\pi$ is not selected) or the closing event of the father of $\pi$ ($\pi$ is selected). Thus, the optimal event associated with selected node $\pi$ different from the root is the closing event of the father of $\pi$. This is the first event from which it is safe to select $\pi$.

If the schema $D_0 = \{a \to a^*b,\ b \to \epsilon\}$ is assumed to be satisfied for all input trees $t$, i.e. $t \in L(D_0)$, the query may select all nodes labeled by $b$ at opening time, and refuse all nodes labeled by $a$ except the root at opening time too. So, the optimal event for all selected nodes $\pi$ is its opening event. Next, we define optimal events more formally.

**Definition 1** (Sufficient and optimal events)**.** *For every $n$-ary query $q$ in trees of $T_\Sigma$, tree language $L \subseteq T_\Sigma$, and tree $t \in L$, we define*




(a) A tree $t \in T_{\{a,b\}}$

$\mathtt{opt}_{q_0}(t)$=$\{(\pi_0, (\mathtt{open}, \pi_0)), (\pi_2, (\mathtt{close}, \pi_1)), (\pi_3, (\mathtt{close}, \pi_0))\}$

(b) Corresponding optimal events without schema restriction

$\mathtt{opt}_{q_0}^{L(D_0)}(t)$=$\{(\pi_0, (\mathtt{open}, \pi_0)), (\pi_2, (\mathtt{open}, \pi_2)), (\pi_3, (\mathtt{open}, \pi_3))\}$

(c) Optimal events with schema $D_0 = \{a \to a^*b,\ b \to \epsilon\}$

**Figure 1.** An input tree and its optimal events for $q_0$ with and without schema restriction

*a relation* $\mathtt{suff}_q^L(t)$ *between tuples* $\sigma \in \mathtt{nodes}(t)^n$ *and events* $e \in \mathtt{events}(t) - \{\mathtt{start}\}$ *as follows:*

$$(\sigma, e) \in \mathtt{suff}_q^L(t) \Leftrightarrow \begin{cases} \sigma \in \mathtt{nodes}(t^{\leq e})^n \wedge \\ \forall t' \in L,\ \mathit{equal}_e(t, t') \Rightarrow \sigma \in q(t') \end{cases}$$

*The relation* $\mathtt{opt}_q^L(t)$ *relates selected* $n$*-tuples to optimal events:*

$$(\sigma, e) \in \mathtt{opt}_q^L(t) \Leftrightarrow e = \mathit{min}_{\preceq^t}\{e' \mid (\sigma, e') \in \mathtt{suff}_q^L(t)\}$$

*If* $L = T_\Sigma$ *then we write* $\mathtt{suff}_q(t)$ *and* $\mathtt{opt}_q(t)$ *without subscript.*

In order to illustrate these relations, we reconsider query $q_0$ defined above. Figure 1 explicits these relations on an input tree in two cases, with and without the schema. It shows that nodes $\pi_2$ and $\pi_3$ can be selected earlier in the presence of the schema.

From sufficient events for $n$-ary queries, we can derive two ($n$+1)-ary queries defining sufficient nodes for selecting $n$-tuples with respect to opening or closing actions. For $\alpha \in \{\mathtt{open}, \mathtt{close}\}$ let:

$$\mathtt{suff}_q^{L,\alpha}(t) = \{(\sigma, \pi) \mid (\sigma, (\alpha, \pi)) \in \mathtt{suff}_q^L(t)\}$$

In Section 4, we will prove that $\mathtt{suff}_q^{L,\alpha}$ is MSO definable for all MSO definable queries $q$ and regular schemas $L$.

## 2. Lower Complexity Bound

We infer a lower bound for complexity of the *optimality problem*, which is the problem to decide whether $(\sigma, e) \in \mathtt{opt}_{q(A)}(t)$ for some possibly nondeterministic tree automaton $A$. To keep argument simple, we represent queries in binary trees by standard tree automata. What is essential for the hardness result is that we do allow for non-determinism.

**Definition 2** (Optimality problem)**.** *The problem can be defined for either standard tree automata or STAs. In the first case, only binary trees are to be considered, while unranked trees are permitted in the second one. In both cases, we have the following parameters, inputs, and outputs:*

*PARAMETERS: a signature* $\Sigma$*, a natural number* $n \geq 0$*, a tree automaton* $D$ *over* $\Sigma$ *(representing the schema).*

*INPUTS: a tree automaton* $A$ *over* $\Sigma \times \mathbb{B}^n$ *whose language is canonical (representing the query), a tree* $t \in T_\Sigma$*, an* $n$*-tuple* $\sigma \in \mathtt{nodes}(t)^n$*, and an event* $e \in \mathtt{events}(t) - \{\mathtt{start}\}$*.*

*OUTPUT: the truth value of* $(\sigma, e) \in \mathtt{opt}_{q(A)}^{L(D)}(t)$*.*

Every choice of the parameters specifies a decision problem. Hardness does not depend on this choice ; we can fix the parameters mostly arbitrarily but nontrivial. In particular, the optimality problem remains hard for $\Sigma = \{a\}$, $n = 0$, and every $D$ with $L(D) = T_\Sigma$. Hardness is invariant under the choice of the particular automata notions too.

**Proposition 1.** *The optimality problem is DEXPTIME-hard for Boolean queries in binary trees represented by possibly nondeterministic tree automata.*

*Proof.* We use reduction from the universality problem for (nondeterministic) tree automata for trees built from a single constant $a$ and a single binary function symbol called $a$ as well. Automata rules have the form $a(q_1, q_2) \to q$ or $a \to q$ for states $q, q_1, q_2 \in \mathtt{states}(A)$. Universality for this kind of tree automata is DEXPTIME complete (21).

Let $A$ be a tree automaton for binary $a$-trees. The language $L(A)$ is canonical for $n = 0$, so it defines a Boolean query $q(A)$. This query selects the empty tuple in all trees $t \in L(A)$. Thus, for all binary trees $t$ build from $a$'s only:

$$((), (\epsilon, \mathtt{open})) \in \mathtt{opt}_{q(A)}(t) \text{ iff } L(A) \text{ is universal.} \qquad \square$$

Since, the optimality problem is DEXPTIME-hard for Boolean queries, it is also clearly DEXPTIME-hard for $n$-ary queries with $n > 0$.

## 3. Streaming Tree Automata

Streaming tree automata (STAs) operate on unranked trees in streaming order, and recognize exactly the set of regular languages of unranked trees. They unify three previous automata notions used for streaming: nested word automata (1), visibly pushdown automata (2), and pushdown forest automata (15). A comparison of these three families is the subject of (10).

**Definition 3.** *An STA* $A$ *consists of 6 finite sets, a signature* $\Sigma$ *of tree labels, a signature* $\Gamma$ *of stack symbols, a states set* $\mathtt{states}$*, subsets* $\mathtt{init}, \mathtt{final} \subseteq \mathtt{states}$ *of initial (respectively final) states and a set* $\mathtt{rules} \subseteq \{\mathtt{open}, \mathtt{close}\} \times \Sigma \times \Gamma \times \mathtt{states}^2$ *of rules written*

$$\mathtt{open}\ a\ p_0 \to p_1\ \gamma_1 \qquad \text{or} \qquad \mathtt{close}\ a\ p_0\ \gamma_0\ \to p_1$$

*where* $p_0, p_1 \in \mathtt{states}$*,* $a \in \Sigma$*, and* $\gamma_0, \gamma_1 \in \Gamma$*.*

Whenever useful, we index the components of $A$ by upper index $A$ so that $A = (\Sigma^A, \Gamma^A, \mathtt{states}^A, \mathtt{init}^A, \mathtt{final}^A, \mathtt{rules}^A)$.

*Evaluators.* Unranked trees define an algebra with the adjunction operator $@ : T_\Sigma \times T_\Sigma \to T_\Sigma$ such that $a(t_1, \ldots, t_{k-1})@t_k = a(t_1, \ldots, t_k)$. This algebra is isomorphic to the term algebra over the signature $\Sigma \times \{@\}$ with a single binary function symbol.

The tree language accepted by STAs can be defined by set valued evaluation, that computes all results of all possible runs. An evaluator traverses unranked trees along their algebraic structure, i.e. their construction from constants in $\Sigma$ and tree extension $@$. We define the following sets for all labels $a \in \Sigma$, states $q \in \mathtt{states}$, and stack symbols $\gamma \in \Gamma$:

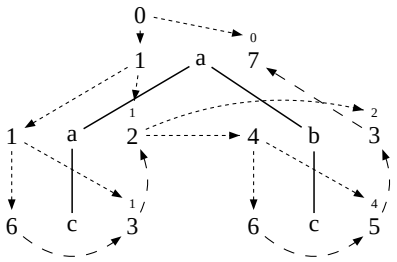
$$\mathtt{open}^A(a, p) = \{(p_1, \gamma_1) \mid \mathtt{open}\ a\ p \to p_1\ \gamma_1\ \ \mathtt{in}\ \mathtt{rules}^A\}$$
$$\mathtt{close}^A(a, (p, \gamma)) = \{p_1 \mid \mathtt{close}\ a\ p\ \gamma \to p_1\ \ \mathtt{in}\ \mathtt{rules}^A\}$$

The evaluator of $A$ is defined by two mutually recursive functions $\mathtt{eval}^A$ and $\mathtt{open}^A$, that are defined as follows for all $p \in \mathtt{states}$ and trees $t, t_1, t_2 \in T_\Sigma$:

$\mathtt{eval}^A(t, p) = \mathtt{close}^A(a, \mathtt{open}^A(t, p))$ where $a = \mathtt{label}^t(\epsilon)$
$\mathtt{open}^A(t_1@t_2, p) = \{(p_2, \gamma_1)\ \mid$

$\texttt{open}\ a\ 0 \rightarrow 1\ 0 \quad \texttt{close}\ a\ 3\ 0 \rightarrow 7$
$\texttt{open}\ a\ 1 \rightarrow 1\ 1 \quad \texttt{close}\ a\ 3\ 1 \rightarrow 2$
$\texttt{open}\ b\ 2 \rightarrow 4\ 2 \quad \texttt{close}\ b\ 5\ 2 \rightarrow 3$
$\texttt{open}\ c\ 1 \rightarrow 6\ 1 \quad \texttt{close}\ c\ 6\ 1 \rightarrow 3$
$\texttt{open}\ c\ 4 \rightarrow 6\ 4 \quad \texttt{close}\ c\ 6\ 4 \rightarrow 5$

(a) Example of STA




(b) Successful run of the STA


(c) $\texttt{open}\ c\ 1 \rightarrow 6\ 1$ (d) $\texttt{close}\ a\ 3\ 1 \rightarrow 2$

**Figure 2.** An STA, and an example of run

$$(p_1, \gamma_1) \in \texttt{open}^A(t_1, p),\ p_2 \in \texttt{eval}^A(t_2, p_1)\}$$

We freely lift $\texttt{eval}^A$ to set valued evaluation, so that for all $S \subseteq \texttt{states}^A$, $\texttt{eval}^A(t, S) = \{\texttt{eval}^A(t, p) \mid p \in S\}$.

The evaluation of a tree starts in mode `eval` with some state $p$. When applied to some tree $t_1@t_2$, this tree is opened, so that $t_1$ can be evaluated in `open` mode from state $p$. The result is a state $p_1$ and a stack symbol $\gamma_1$. Evaluation continues in mode `eval` with tree $t_2$ from state $p_1$ leading into some state $p_2$. Finally, $t_1@t_2$ is closed, while returning a state in $\texttt{close}^A(a, (p_2, \gamma_1))$. This is the time point, when the stack symbol $\gamma_1$ is reused. If implemented in a functional programming language, the memoization of stack symbol $\gamma_1$ is managed by the call stack of the evaluator. A tree $t \in T_\Sigma$ belongs to the language $L(A)$ recognized by $A$ iff the evaluator of $A$ satisfies $\texttt{eval}(t, \texttt{init}) \cap \texttt{final} \neq \emptyset$. An example STA is given in Figure 2(a).

*Runs.* A run of an STA $A$ on a tree $t$ is a function $r : \texttt{events}(t) \rightarrow \texttt{states}$ assigning states to events for which there exists a stacking function $s : \texttt{nodes}(t) \rightarrow \Gamma$ such that $r(\texttt{start}) \in \texttt{init}^A$ and for all $\pi \in \texttt{nodes}(t)$ with $a = \texttt{label}(t, \pi)$:

$$\texttt{open}\ a\ r(\texttt{pred}(\texttt{open}, \pi)) \rightarrow r(\texttt{open}, \pi)\ s(\pi) \in \texttt{rules}^A$$
$$\texttt{close}\ a\ r(\texttt{pred}(\texttt{close}, \pi))\ s(\pi) \rightarrow r(\texttt{close}, \pi) \in \texttt{rules}^A$$

The set of all possible runs of the STA $A$ on the tree $t$ is denoted $\texttt{runs}^A(t)$. A run $r$ is successful if $r(\texttt{close}, \epsilon) \in \texttt{final}^A$. Let $\texttt{runs\_succ}^A(t)$ be the set of such successful runs of $A$ on $t$.

A successful run on tree $a(a(c), b(c))$ is shown in Figure 2(b). At each node, we write on its left (resp. right) side the state reached after encountering the opening (resp. closing) event of this node. The stack symbol (pushed when opening and popped when closing) is placed on the upper right. Figure 2(c) (resp. 2(d)) represents the application of an open (resp. close) rule.

**Lemma 1.** *There exists a successful run $r$ of $A$ on $t$ if and only if $t \in L(A)$.*

## 4. An Algorithm for Earliest Query Answering

We elaborate an earliest query answering algorithm for queries defined by STAs and subject to regular schema restrictions. In Subsection 4.1, we show to compute optimal events by STAs.

The following three steps are original to this paper. The first problem is that the automata construction relies on a safety predicate, that is difficult to compute for nondeterministic automata. In Subsection 4.3 we show how to solve this problem after determinization. A further problem remains, which is that the size of the constructed automaton may be exponential in $n$. In Subsection 4.4 we show that we can perform automata transitions in polynomial time from every given state, nevertheless. In the last step (Subsection 4.5), we use these insights to construct a polynomial time earliest query answering algorithm.

### 4.1 Detecting sufficient events with an STA

We show how to compute optimal events for $n$-ary queries defined by STAs under regular schema assumptions, by constructing deterministic STAs.

**Proposition 2** (Computing optimal events by deterministic STAs)**.** *For every STA $A$ recognizing a canonical language over $\Sigma \times \mathbb{B}^n$ and schema defined by an STA $D$ over $\Sigma$ there exists a deterministic STA $C$ over $\Sigma \times \mathbb{B}^n$ and a selection set $Q \subseteq \texttt{states}(C)$ such that for all $t \in T_\Sigma$:*

$$\texttt{suff}^{L(D)}_{q(A)}(t) = \{(\sigma, e) \in \texttt{nodes}(t)^n \times \texttt{events}(t) \mid \exists r \in \texttt{runs}^C(\texttt{ch}(t, \sigma)).\ r(e) \in Q\}$$

This will finally be proved by Propositions 3 and 4.

***Schema handling.*** Let $q = q(A)$ for some STA $A$ over $\Sigma \times \mathbb{B}^n$ and $L = L(D)$ for some STA $D$ over $\Sigma$.

In a first step, we compute a deterministic automaton $B$ which adds information about $D$ to $A$. The final validation of the input tree $t \in L(D)$ will be done independently of the detection of optimal events. Automaton $B$ is supposed to accept trees from $\texttt{Can}_q$ plus those annotated trees whose projection to $\Sigma$ does not belong to the schema. More formally, for every annotated tree $t \in T_{\Sigma \times \mathbb{B}^n}$ let $\Pi_\Sigma(t) \in T_\Sigma$ be the relabeling of $t$ in which all bit vectors are removed. Automaton $B$ is supposed to recognize the following language:

$$\texttt{InputLang}(q, L) = Can_q \cup \{t \in T_{\Sigma \times \mathbb{B}^n} \mid \Pi_\Sigma(t) \notin L\}$$

In order to compute a deterministic STA $B$ recognizing this language, we first determinize $A$ and $D$ and complete them by adding sink states. The rules of $B$ are infered as follows:

$$\frac{\texttt{open}\ a_v\ p_0 \rightarrow p_1\ \gamma_1 \in \texttt{rules}^A \quad \texttt{open}\ a\ p'_0 \rightarrow p'_1\ \gamma'_1 \in \texttt{rules}^D}{\texttt{open}\ a_v\ (p_0, p'_0) \rightarrow (p_1, p'_1)\ (\gamma_1, \gamma'_1) \in \texttt{rules}^B}$$

$$\frac{\texttt{close}\ a_v\ p_0\ \gamma_0 \rightarrow p_1 \in \texttt{rules}^A \quad \texttt{close}\ a\ p'_0\ \gamma'_0 \rightarrow p'_1 \in \texttt{rules}^D}{\texttt{close}\ a_v\ (p_0, p'_0)\ (\gamma_0, \gamma'_0) \rightarrow (p_1, p'_1) \in \texttt{rules}^B}$$

We set the initial and final states to: $\texttt{init}^B = \texttt{init}^A \times \texttt{init}^D$ and $\texttt{final}^B = (\texttt{final}^A \times \texttt{states}^D) \cup (\texttt{states}^A \times (\texttt{states}^D - \texttt{final}^D))$. $A$ and $D$ being deterministic, $B$ is deterministic too.

***Construction of automaton*** $\mathit{earliest}(B)$**.** We transform $B$ into an automaton $C = \mathtt{earliest}(B)$ which recognizes the same language. Every state is enriched by a set of safe states, i.e. states such that any continuation leads to a final state. The states of $C$ are pairs of states of $B$ and subsets of safe states of $B$. Safe states are also added to stack symbols:

$$\mathtt{states}^C = \mathtt{states}^B \times 2^{\mathtt{states}^B}$$
$$\Gamma^C = \Gamma^B \times 2^{\mathtt{states}^B}$$

Automaton $C$ assigns a state $(p, S)$ to an event of the input tree, if

- $B$ assigns $p$ to the same event, and
- if $S$ is the set of safe states for entering into a hedge of children at the current node.

The safe states of the `start` event are the final states of $B$, i.e.:

$$\mathtt{init}^C = (\mathtt{init}^B, \mathtt{final}^B)$$

Final states of $C$ are final states of $B$ associated with the safe states of the root:

$$\mathtt{final}^C = \{(p, \mathtt{final}^B) \mid p \in \mathtt{final}^B\}$$

In order to define safe states for nodes below the root, we need some further definitions. Let the set of hedges annotated by 0-bit vectors be the set of sequence of trees annotated by 0-bit vectors: $H_{\Sigma\times\{0\}^n} = T^*_{\Sigma\times\{0\}^n}$. We extend the definition of the evaluator for STAs $E$ to hedges as follows:

$$\mathtt{eval}^E((), p) = \{p\}$$
$$\mathtt{eval}^E((t_1, \ldots, t_{k+1}), p) = \mathtt{eval}^E(t_{k+1}, \mathtt{eval}^E((t_1, \ldots, t_k), p))$$

Now we can define the set of safe states relatively to a set $S \subseteq \mathtt{states}^B$, a label $a \in \Sigma \times \mathbb{B}^n$, and a stack symbol $\gamma \in \Gamma^B$. This consists in all states of $B$ such that entering in every hedge of children and then closing the parent node leads to a state of $S$.

$$\mathtt{safe}^B(a, \gamma, S) = \{p \mid \forall h \in H_{\Sigma\times\{0\}^n}, \mathtt{close}^B(a, (\mathtt{eval}^B(h, p), \gamma)) \in S\}$$

This way, the safe states are computed at each opening event, and propagated through the children by the stack. The opening rules of $C$ are infered from the rules of $B$ by this way:

$$\frac{\mathtt{open}\ a\ p_0 \to p_1\ \gamma_1 \in \mathtt{rules}^B \quad S_1 = \mathtt{safe}^B(a, \gamma_1, S_0)}{\mathtt{open}\ a\ (p_0, S_0) \to (p_1, S_1)\ (\gamma_1, S_0) \in \mathtt{rules}^C}$$

Safe states are only computed on opening events. Processing a closing event consists only in propagating already computed safe states. The ones on the top of the stack are sent to the next event. The incoming set of safe states is not propagated. It can be used to know if we come from a safe state.

$$\frac{\mathtt{close}\ a\ p_0\ \gamma_0 \to p_1 \in \mathtt{rules}^B \qquad S_0, S_1 \subseteq \mathtt{states}^B}{\mathtt{close}\ a\ (p_0, S_0)\ (\gamma_0, S_1) \to (p_1, S_1) \in \mathtt{rules}^C}$$

These inference rules show that every run of $B$ can be translated to a run of $C$. This translation is exact: $B$ and $C$ recognize the same language.

***Correctness.*** Automaton $C = \mathtt{earliest}(B)$ can be used to recognize $\mathtt{suff}_q^{L,\alpha}$ for both $\alpha \in \{\mathtt{open}, \mathtt{close}\}$. To prove this correctness property, we need the two properties of Lemma2.

The first lemma indicates that a safe run exists in $C$ iff the sufficiency property holds. By safe run, we mean a run $r$ of $C$ verifying $r(e) = (p, S)$ at event $e$ with $p \in S$. More precisely, for every input tree $t \in T_{\Sigma\times\mathbb{B}^n}$:

- $\mathtt{safe\_run}(C, t, \mathtt{init})$ never holds
- for $e \in \mathtt{events}(t) - \{\mathtt{init}\}$, $\mathtt{safe\_run}(C, t, e)$ holds iff there is a run $r \in \mathtt{runs}^C(t)$ such that $\exists S \subseteq \mathtt{states}^C, \exists p \in S$ with $r(e) = (p, S)$.

The next lemma shows that if a safe state is reched, then any continuation is recognized by $B$, and that once a safe state is reached during a run, the following states remain safe.

**Lemma 2.** *For every deterministic STA $B$ recognizing the language $\mathit{InputLang}(q, L)$, if $C = \mathit{earliest}(B)$ then for every tree $t \in L$, the following properties hold:*

1. *for every event $e \in \mathit{events}(t) - \{\mathit{init}\}$,*

$$\mathit{safe\_run}(C, t, e) \Leftrightarrow \forall t' \in L,\ t^{\leq e} \in \mathit{prefix}(t') \Rightarrow t' \in L(B)$$

2. *for every event $e \in \mathit{events}(t)$,*

$$\mathit{safe\_run}(C, t, e) \Rightarrow \forall e' \in \mathit{events}(t),\ e \prec^t e' \Rightarrow \mathit{safe\_run}(C, t, e')$$

Using these properties, we can prove that the safe states correctly detect the sufficient events:

**Proposition 3** (Early detection)**.** *For every MSO-definable $n$-ary query $q$ and regular schema language $L$, if $B$ is a deterministic STA recognizing $\mathit{InputLang}(q, L)$, then the STA $C = \mathit{earliest}(B)$ satisfies the following property for all $t \in L$, $\sigma \in \mathit{nodes}(t)^n$, and $e \in \mathit{events}(t)$:*

$$(\sigma, e) \in \mathit{suff}_q^L(t) \Leftrightarrow \begin{cases} \exists r \in \mathit{runs}^C(\mathit{ch}(t, \sigma)), \\ \exists S \subseteq \mathit{states}^B,\ \exists p \in S,\ r(e) = (p, S) \end{cases}$$

This proposition proves the recognizability of the relations $\mathtt{suff}_q^{L,\alpha}$. A key property for streaming holds in this construction. The automaton $C$ is able to detect optimal nodes immediately when they happen, as indicated by the following proposition.

**Proposition 4.** *For every deterministic STA $B$, $\mathit{earliest}(B)$ is deterministic.*

### 4.2 Failure states

Failure states are states of $B$ from which every valid continuation of the input tree leads to no selection. This is complementary to the notion of safe states: once in a failure state, the candidate tuple can be discarded because we are sure that whatever follows the current event, the candidate tuple $\sigma$ will not be selected, or the input tree will not be valid.

A major difference between safe and failure states is that failure states are used on complete and incomplete candidate tuples, whereas safe states are only used on complete candidate tuples (from the definition of optimality). By complete candidate tuple, we mean a tuple for which selection occurred on every components, i.e. every component contains a node of the input tree. For this reason, we cannot use the same technique as before on the complementary of the query.

As a consequence, the computation of failure states is very similar to the computation of safe states, in particular the propagation of the set of failure states in the automaton is the same. What differs is the initialization and the update steps.

The initial set of failure states is the set $F_0$ of states that recognizes non selected tuples or invalid trees:

$$F_0 = ((\mathtt{states}^A - \mathtt{final}^A) \times \mathtt{states}^D) \cup (\mathtt{states}^A \times (\mathtt{states}^D - \mathtt{final}^D))$$

Detecting a failure states means that any continuation of the annotated tree (including non-canonical ones) will lead to no selection or to an invalid tree. We do not have to restrict continuations to canonical ones because non-canonical continuations always lead to a non-final state of $A$.

We define the set $\texttt{fail}^B$, which is an adaptation of the definition of $\texttt{safe}^B$ to failure states. The only difference appears in the set of hedge, $H_{\Sigma\times\mathbb{B}^n}$ replacing $H_{\Sigma\times\{0\}^n}$:

$$\texttt{fail}^B(a,\gamma,F) = \{p \mid \forall h \in H_{\Sigma\times\mathbb{B}^n}, \texttt{close}^B(a,(\texttt{eval}^B(h,p),\gamma)) \in F\}$$

Let $\oslash$ indicate components of a candidate tuple for which no selection already occurred. For any tuple $\sigma \in (\texttt{nodes}(t) \cup \{\oslash\})^n$ and any $e \in \texttt{events}(t)$, $\texttt{completions}(\sigma,t,e)$ represents the set of completions of $\sigma$ with nodes of $t$ after $e$: $\texttt{completions}((\pi_1,\ldots,\pi_n),t,e) = \{(\pi'_1,\ldots,\pi'_n) \in \texttt{nodes}(t)^n \mid \forall i \in [1..n], \pi_i \neq \pi'_i \Rightarrow \pi_i = \oslash \wedge e \prec^t (\texttt{open},\pi'_i)\}$. We extend canonical trees on incomplete tuples $\sigma$: the labels of nodes of $\texttt{ch}(t,\sigma)$ are set to the value 0 for components marked by $\oslash$.

To define the set of events from which a failure state is reached for a given candidate tuple, we introduce the relation $\widehat{\texttt{suff}}_q^L$. This corresponds to sufficient events, but for failure states instead of safe states.

**Definition 4.** *For every $n$-ary query $q$ in trees of $T_\Sigma$, schema $L$ and tree $t \in L$, we define the relation $\widehat{\textit{suff}}_q^L(t)$ between candidate $n$-tuples $\sigma \in (\textit{nodes}(t) \cup \{\oslash\})^n$ and events $e \in \textit{events}(e) - \{\textit{init}\}$:*

$$(\sigma,e) \in \widehat{\textit{suff}}_q^L(t) \Leftrightarrow \begin{cases} \forall t' \in L,\ t^{\trianglelefteq e} \in \textit{prefix}(t') \Rightarrow \\ \forall \sigma' \in \textit{completions}(\sigma,t',e),\ \sigma' \notin q(t') \end{cases}$$

In the previous section, we detailed the construction of the STA $\texttt{earliest}(B)$ that detects sufficient events given a tree and a candidate tuple. Similarly, we define the STA $\texttt{earliest}_{\texttt{fail}}(B)$ that detects events from which a failure state is reached in $B$, given a tree and a candidate tuple. The STA $\texttt{earliest}_{\texttt{fail}}(B)$ is obtained via the same steps as for $\texttt{earliest}_{\texttt{fail}}(B)$, except that we replace the final states of $B$ by $F_0$, and we use $\texttt{fail}^B$ instead of $\texttt{safe}^B$.

**Proposition 5** (Early detection of failure states)**.** *For every MSO-definable $n$-ary query $q$ and regular schema language $L$, if $B$ is a deterministic STA recognizing $\textit{InputLang}(q,L)$, then the STA $C = \textit{earliest}_{\textit{fail}}(B)$ satisfies the following property for all $t \in L$, $\sigma \in (\textit{nodes}(t) \cup \{\oslash\})^n$, and $e \in \textit{events}(t)$:*

$$(\sigma,e) \in \widehat{\textit{suff}}_q^L(t) \Leftrightarrow \begin{cases} \exists r \in \textit{runs}^C(\textit{ch}(t,\sigma)), \\ \exists F \subseteq \textit{states}^B,\ \exists p \in F,\ r(e) = (p,F) \end{cases}$$

*Proof.* This proposition is the adaptation of Proposition 3 to failure states. It can be proved by adapting proofs of Lemma 2 and Proposition 3, using $F_0$ for final states of $B$, and taking care of incomplete candidates. □

### 4.3 Precomputations

This section describes which computations are done in our algorithm before parsing the stream.

A first step of the algorithm is to compute $B$, the automaton recognizing $\texttt{InputLang}(q,L)$, as explained in the paragraph "schema" of Section 4.1.

Moreover, we can precompute a relation that will be used for the computation of safe (and failure) states at each opening event, as explained below.

***Accessibility for safe states.*** The rules of $\texttt{earliest}(B)$ indicate that our algorithm has to compute the safe states for the current label, stack symbol, and previous safe states at each opening event. To compute these set of states, we first need to find the states of $B$ that are accessible through a hedge of $H_{\Sigma\times\{0\}^n}$, from each state of $B$. This is done via the predicate $\texttt{accH}_0$:

$$\frac{p \in \texttt{states}^B}{\texttt{accH}_0(p,p).}$$

$$\frac{p_1,p_2,p_3 \in \texttt{states}^B}{\texttt{accH}_0(p_1,p_2) \texttt{ :- } \texttt{accH}_0(p_1,p_3),\texttt{accH}_0(p_3,p_2).}$$

$$\frac{a \in \Sigma \times \{0\}^n \quad (p_3,\gamma) \in \texttt{open}^B(a,p_1) \quad p_2 \in \texttt{close}^B(a,(p_4,\gamma))}{\texttt{accH}_0(p_1,p_2) \texttt{ :- } \texttt{accH}_0(p_3,p_4).}$$

This predicate computes exactly the states accessible through a hedge of $H_{\Sigma\times\{0\}^n}$.

**Proposition 6.** *For any $(p_1,p_2) \in (\textit{states}^B)^2$, we have:* $\textit{accH}_0(p_1,p_2) \Leftrightarrow (\exists h \in H_{\Sigma\times\{0\}^n},\ \textit{eval}^B(h,p_1) = p_2)$

These rules constitute a ground Datalog program that computes the $\texttt{accH}_0$ relation. As this can be resolved by standard saturation techniques, we do not go into further details.

***Accessibility for failure states.*** As explained in section, we use the accessibility through hedges of $H_{\Sigma\times\{0\}^n}$ instead of hedges of $H_{\Sigma\times\mathbb{B}^n}$.

As a consequence we introduce the predicate $\texttt{accH}$, expressing accessibility through hedges of $H_{\Sigma\times\mathbb{B}^n}$ in $B$:

$$\frac{p \in \texttt{states}^B}{\texttt{accH}(p,p).}$$

$$\frac{p_1,p_2,p_3 \in \texttt{states}^B}{\texttt{accH}(p_1,p_2) \texttt{ :- } \texttt{accH}(p_1,p_3),\texttt{accH}(p_3,p_2).}$$

$$\frac{(p_3,\gamma) \in \texttt{open}^B(a,p_1) \quad p_2 \in \texttt{close}^B(a,(p_4,\gamma))}{\texttt{accH}(p_1,p_2) \texttt{ :- } \texttt{accH}(p_3,p_4).}$$

**Proposition 7.** *For any $(p_1,p_2) \in (\textit{states}^B)^2$, we have:* $\textit{accH}(p_1,p_2) \Leftrightarrow (\exists h \in H_{\Sigma\times\mathbb{B}^n},\ \textit{eval}^B(h,p_1) = p_2)$

***Complexity of this precomputation.*** Proposition 8 estimates the complexity of the precomputation.

**Proposition 8.** *For any deterministic input STA $A$ and $D$ recognizing respectively the canonical language of the query and the schema, the precomputation step of our algorithm is done in time $O(|A|^3.|D|^3 + |\Sigma|^2.|\Gamma|^2.|A|^2.|D|^2)$.*

*Proof.* The completion of $A$ and $D$ is respectively done in time $O(|\Sigma|.|\Gamma|.|\texttt{states}^A|)$ and $O(|\Sigma|.|\Gamma|.|\texttt{states}^D|)$. These bounds are also bounds on the number of rules of the corresponding completed automata. To avoid confusion, we denote these completed automata by $A_c$ and $D_c$. $B$ is similar to a product automaton between $A_c$ and $D_c$. Its number of rules and the time to compute it are in $O(|\texttt{rules}^{A_c}|.|\texttt{rules}^{D_c}|)$. The number of rules of the ground Datalog programs defining $\texttt{accH}_0$ and $\texttt{accH}$ are in $O(|\texttt{states}^B|^3 + |\texttt{rules}^B|.|\texttt{states}^B|)$. The computation of $\texttt{accH}_0$ and $\texttt{accH}$ consist in a saturation process on these rules. This one can be done in time linear in the number of rules. As a consequence, computing $\texttt{accH}_0$ and $\texttt{accH}$ is done in $O(|\texttt{states}^B|^3 + |\texttt{rules}^B|.|\texttt{states}^B|)$. □

***Example.*** In the following, we consider the example given in Section 1.3. The query $q_0$ selects nodes that have no next sibling, and the schema $L_0$ corresponds to the DTD $\{a \to a^*b,\ b \to \epsilon\}$. We show how the algorithm would behave on this input.

Let $A$ be the STA on $T_{\Sigma\times\mathbb{B}}$ such that $\texttt{states}^A = \{0,1,2\}$, where $\texttt{init}^A = \{0\}$, $\texttt{final}^A = \{1,2\}$, $\Gamma^A = \{0\}$ and the rules of $A$ are the following ones, for every $x \in \Sigma$:

$\texttt{open}\ x_0\ 0 \to 0\ 0$ $\quad$ $\texttt{close}\ x_0\ 0\ 0 \to 0$ $\quad$ $\texttt{close}\ x_0\ 1\ 0 \to 2$
$\texttt{open}\ x_1\ 0 \to 0\ 0$ $\quad$ $\texttt{close}\ x_1\ 0\ 0 \to 1$ $\quad$ $\texttt{close}\ x_0\ 2\ 0 \to 2$
$\texttt{open}\ x_0\ 2 \to 2\ 0$

| accH | 00 | 01 | 02 | 10 | 11 | 12 | 20 | 21 | 22 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 1 | 1 | 1 | *1* | *1* | *1* | *1* | *1* | *1* | *1* | *1* | *1* |
| 01 | 0 | 1 | 1 | 0 | 0 | *1* | 0 | 0 | *1* | 0 | 0 | *1* |
| 02 | 0 | 0 | 1 | 0 | 0 | *1* | 0 | 0 | *1* | 0 | 0 | *1* |
| 10 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 11 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 12 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 20 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | *1* | *1* | *1* |
| 21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | *1* |
| 22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | *1* |
| 30 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 31 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 32 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**Figure 3.** The relation accH corresponding to $q_0$ and $L_0$

Let $D$ be the STA on $T_\Sigma$ with $\texttt{states}^D = \{0, 1\}$, 0 being the initial state and the only final state, $\Gamma^D = \{0\}$ and the rules of $D$ being:

$$\texttt{open}\, a\, 0 \to 0\, 0 \quad \texttt{close}\, a\, 1\, 0 \to 0$$
$$\texttt{open}\, b\, 0 \to 0\, 0 \quad \texttt{close}\, b\, 0\, 0 \to 1$$

We have $q_0 = q(A)$ and $L_0 = L(D)$. We start by completing $A$ with the sink state 3 and $D$ with the sink state 2. By applying the inference rules, we obtain the STA $B$, recognizing $\texttt{InputLang}(q_0, L_0)$. States of $B$ are pairs $(p_0, p_1) \in \texttt{states}^A \times \texttt{states}^D$. The initial state is $(0, 0)$, the final states are $(1, 0)$, $(1, 1)$, $(1, 2)$, $(2, 0)$, $(2, 1)$, $(2, 2)$ and $(0, 1)$, $(0, 2)$, $(3, 1)$, $(3, 2)$. The stack alphabet contains only the element $(0, 0)$. We denote this one by $\gamma$, to distinguish it from the state $(0, 0)$. The rules are infered from the rules of $A$ and $D$ according to the two inference rules detailed previously.

Then we compute the relations $\texttt{accH}_0$ and accH. Figure 3 is an array of Booleans representing the relation accH. States $(p_0, p_1)$ are written $p_0p_1$ for sake of conciseness. The relation $\texttt{accH}_0$ is obtained from this array by replacing values in italics by 0.

We write $\texttt{accH}_0(p) = \{p' \mid \texttt{accH}_0(p, p')\}$ and $\texttt{accH}(p) = \{p' \mid \texttt{accH}(p, p')\}$. Thus in our example $\texttt{accH}_0((1, 0)) = \{(1, 0), (3, 0), (3, 1), (3, 2)\}$.

### 4.4 Processing an event

We describe here how a candidate tuple $\sigma$ is updated when an event is received. We first explain what is exactly a candidate. Then we show the code corresponding to this update. This one needs to compute some safe states, so we show how this can be done by reusing precomputed data. Finally, we analyze what can be lazily evaluated, and what is the complexity of this update.

***Candidates.*** The algorithm will operate on an input tree $t \in T_\Sigma$, and has to compute the runs on all possible $n$-ary canonical labelings of $t$. As a consequence, it has to deal with a set of candidates, each candidate corresponding to the run of $\texttt{earliest}(B)$ on one canonical labeling of the input tree, i.e. one candidate tuple of nodes. Thus, at each step of the processing, a candidate contains a candidate tuple $\sigma$, the current state $(p, S)$ of $\texttt{earliest}(B)$ for this run, a set of "failure states" (denoted by $F$) and the stack content $s$.

Finally, a 5-tuple $(\sigma, p, S, F, s)$ is a candidate at the event $e$ of the input tree $t \in T_\Sigma$ iff (1) there is a run $r$ of $\texttt{earliest}(B)$ on $\texttt{ch}(t, \sigma)$ until $e$, verifying $r(e) = (p, S)$ and using the stack $s$, (2) no optimal event for $q$ is reached (i.e. $p \notin S$), and (3) if $\sigma$ is complete, then no failure state is reached (i.e. $p \notin F$).

In other words, the set of candidates contains all the tuples leading to this configuration, for which selection can possibly occur or not, depending on the end of the stream.

***Updating a candidate.*** Let us introduce a few notations for sake of clarity. For $v = (b_1, \ldots, b_n) \in \mathbb{B}^n$, $\sigma = (\pi_1, \ldots, \pi_n) \in (\texttt{nodes}(t) \cup \{\oslash\})^n$ and $\pi \in \texttt{nodes}(t)$:

- $\texttt{addSelection}(v, \pi, \sigma)$ is the tuple $(\pi'_1, \ldots, \pi'_n)$ such that $\forall i \in [1..n]$, $\pi'_i \neq \pi_i \Leftrightarrow (b_i \wedge (\pi_i = \oslash) \wedge (\pi'_i = \pi))$. It is used to add the node $\pi$ among already selected nodes of $\sigma$, at the components indicated by $v$.
- $\texttt{projection}(\sigma, \pi)$ is the tuple $(b'_1, \ldots, b'_n) \in \mathbb{B}^n$ such that $\forall i \in [1..n]$, $b'_i \Leftrightarrow \pi_i = \pi$. It indicates which components select $\pi$ in $\sigma$.
- $\texttt{compatible}(v, \sigma)$ is true iff $\forall i \in [1..n]$, $b_i \Rightarrow \pi_i = \oslash$. This indicates if the labeling $v$ does not select a node that has already been selected in $\sigma$.

Figures 4 and 5 describe how such a candidate is processed, respectively for an opening and a closing event. This corresponds precisely to the inference rules that generate rules of $\texttt{earliest}(B)$. At each event, we look for every rule of $B$ that can be applied to the current state. If the event is an opening one, we compute the safe states as explained below.

**Figure 4.** ⟨⟨ process opening event on candidate $(\sigma, p, S, F, s)$ ⟩⟩

```
for open a_v p → p' γ ∈ rules^B with compatible(v,σ)
  let
    S' = safe(a_v, γ, S)
    F' = fail(a_v, γ, F)
    σ' = addSelection(v, π, σ)
  in
  if p' ∈ S' then
    stream_out.write(σ')
  else
    if p' ∉ F' then
      newCandidates.add((σ', p', S', F', s.push(γ, S, F)))
```

**Figure 5.** ⟨⟨ process closing event on candidate $(\sigma, p, S, F, s)$ ⟩⟩

```
let
  (γ, S', F') = s.pop()
  v = projection(σ,π)
in
for close a_v p γ → p' ∈ rules^B
  if p' ∈ S' then
    stream_out.write(σ)
  else
    if p' ∉ F' then
      newCandidates.add((σ, p', S', F', s))
```

***From accessiblity to safe and failure states.*** The precomputation of $\texttt{accH}_0$ was the first step for computing the set of safe states. The second one is done at each opening event, and consists in finding the states that lead to a safe state after closing the current node. We denote this set safeBeforeClose. For $a \in \Sigma \times \{0\}^n$, $\gamma \in \Gamma$ and $S \subseteq \texttt{states}^B$,

$$\texttt{safeBeforeClose}^B(a, \gamma, S) = \{p \mid \texttt{close}^B(a, (p, \gamma)) \in S\}$$

From the definitions of $\texttt{accH}_0$, accH and safeBeforeClose, we immediately obtain the set of safe and failure states.

**Proposition 9.** *For all $a \in \Sigma \times \mathbb{B}^n$, $\gamma \in \Gamma$ and $S \subseteq \texttt{states}^B$, if $B$ is deterministic and complete then the following properties hold:*

$$\mathit{safe}^B(a, \gamma, S) = \{p \mid \mathit{accH}_0(p) \subseteq \mathit{safeBeforeClose}^B(a, \gamma, S)\}$$

$$\mathit{fail}^B(a, \gamma, F) = \{p \mid \mathit{accH}(p) \subseteq \mathit{safeBeforeClose}^B(a, \gamma, F)\}$$

**Figure 6.** ⟨⟨ define safe(B) ⟩⟩

```
/*
    a_v: a label in Σ × B^n
    γ:  a stack symbol
    S:  a set of states of B
    This function returns the set safe^B(a_v, γ, S)
*/
fun safe(a_v, γ, S)
in
  safeBeforeClose = ∅
  for p ∈ states^B
    if close^B(a_v, (p, γ)) ∈ S then
      safeBeforeClose.add(p)
  safeAccess = accH_0
  for (p, p') ∈ safeAccess
    if p' ∉ safeBeforeClose then
      safeAccess.remove((p, p'))
  safeStates = ∅
  for (p, p') ∈ safeAccess
    safeStates.add(p)
  return safeStates
end
```

**Figure 7.** ⟨⟨ define fail(B) ⟩⟩

```
/*
    a_v: a label in Σ × B^n
    γ:  a stack symbol
    F:  a set of states of B
    This function returns the set fail^B(a_v, γ, F)
*/
fun fail(a_v, γ, F)
in
  safeBeforeClose = ∅
  for p ∈ states^B
    if close^B(a_v, (p, γ)) ∈ F then
      safeBeforeClose.add(p)
  failAccess = accH
  for (p, p') ∈ failAccess
    if p' ∉ safeBeforeClose then
      failAccess.remove((p, p'))
  failureStates = ∅
  for (p, p') ∈ failAccess
    failureStates.add(p)
  return failureStates
end
```

Figures 6 and 7 present the code corresponding to these processings.

We can notice that we do not have to test the condition $\texttt{accH}_0(p) \subseteq \texttt{safeBeforeClose}^B(a, \gamma, S)$ for every $p \in \texttt{states}^B$. We restrict this search to states $p \in \texttt{safeBeforeClose}^B(a, \gamma, S)$, as $\texttt{accH}_0$ is reflexive. The same remark holds for accH.

***Laziness.*** We have seen how to compute $\texttt{safe}^B(a, \gamma, S)$ for a given $a$, $\gamma$ and $S$. The algorithm works on an input tree $t \in T_\Sigma$. It will have to consider all the possible canonical labelings of this tree.

However, all the rules of $\texttt{earliest}(B)$ will not be necessarily used for evaluating these labeled trees. So we do not have to compute the whole set of rules of $\texttt{earliest}(B)$. For each labeled tree we apply the run of $B$ and enrich it with safe and failure states (computed at opening events, popped at closing events).

Nevertheless, $\texttt{accH}_0(p)$ has to be computed, for all $p \in \texttt{states}^B$.

***Complexity.*** To bound the number of simultaneous candidates, we extend the definition of concurrency introduced in (4), in order to deal with schemas and $n$-ary queries (and thus incomplete candidates). Moreover, we consider completion from the opening event of a node, instead of the closing one in (4). This way we can express an upper bound on the number of candidates.

Consider the tree $t \in T_\Sigma$, an event $e \in \texttt{events}(t) - \{\texttt{start}\}$ and the candidate tuple $\sigma = (\pi_1, \ldots, \pi_n) \in (\texttt{nodes}(t) \cup \{\emptyset\})^n$. We define the set of events where a candidate tuple can be candidate, i.e. events following $(\texttt{open}, \pi)$ where $\pi$ is the greatest node of $\sigma$: $\texttt{candidateEvents}(t, \sigma) = \{e \in \texttt{events}(t) \mid \forall i \in [1..n],\ (\texttt{open}, \pi_i) \preceq^t e\}$.

**Definition 5.** *Consider a query $q$ and a schema $L$. A candidate tuple $\sigma \in (\texttt{nodes}(t) \cup \{\emptyset\})^n$ is said* alive at event $e \in \texttt{candidateEvents}(t, \sigma)$ *if:*

1. *there is a tree $t_1 \in L$ verifying $t^{\leq e} \in \texttt{prefix}(t_1)$ and a completion $\sigma_1 \in \texttt{completions}(\sigma, t_1, e)$ such that $\sigma_1 \in q(t_1)$ and*
2. *there is a tree $t_2 \in T_\Sigma$ verifying $t^{\leq e} \in \texttt{prefix}(t_2)$ and a completion $\sigma_2 \in \texttt{completions}(\sigma, t_2, e)$ such that $\sigma_2 \notin q(t_2)$.*

*The* concurrency *of the tree $t$ at event $e$ for the query $q$ and the schema $L$ is the number of candidate tuples that are alive at event $e$. The* concurrency *of the tree $t$ for the query $q$ and the schema $L$, denoted by $CONCUR(t, q, L)$, is the maximal concurrency, for all events of $t$.*

Note that $CONCUR(t, q, L)$ may be exponential in the size of $t$. In (14), Meuss, Schulz and Bry propose an intelligent data structure in order to store $n$-ary candidate tuples. We plan to investigate whether such a structure can be adapted to our algorithm.

The following proposition estimates the time needed to process each event.

**Proposition 10.** *For every deterministic input STA $A$ and $D$ recognizing respectively the canonical language of the query $q$ and the schema $L$, and for every input tree $t$, processing an event of $t$ in* `processOpeningEvent` *or* `processClosingEvent` *is done in time $O(c.|\Sigma|^2.|\Gamma|^2.|A|^3.|D|^3)$, where $c = CONCUR(t, q, L)$.*

*Proof.* Processing an opening event requires more computations than processing a closing event, as it needs to determine safe and failure events. The function for opening events computes the safe and failure states for any current candidate tuple and for any opening rule of $B$ that is compatible with already selected nodes of this tuple. We can bound this number of current candidate tuples by $CONCUR(t, q, L)$: the current state of a candidate being neither a safe nor a failure state, there exists an accepting valid completion and a failing completion. The computation of safe (and failure) states is done in time $O(|\texttt{states}^B|^2)$ (see Figures 6 and 7). So the complexity for processing one event is in $O(c.|\texttt{rules}^B|.|\texttt{states}^B|^2)$. The link between the size of $B$ and the sizes of $A$ and $D$ has been studied in the proof of Proposition 8. □

***Example.*** Suppose that we want to compute the safe states at the root for the labeling $a_0$ on our example. This corresponds to computing $\texttt{safe}^B(a_0, \gamma, \texttt{final}^B)$. First, we obtain from the "close" rules of $B$:

$$\texttt{safeBeforeClose}^B(a_0, \gamma, \texttt{final}^B) = \{(0,0),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2),(3,0),(3,2)\}$$

We denote this set $BC_1$. From the previous section, we can look at which states $p$ verify $\texttt{accH}_0(p) \subseteq BC_1$. These states are the safe states:

$$\texttt{safe}^B(a_0, \gamma, \texttt{final}^B) = \{(0,2),(1,1),(1,2),(2,0),(2,1),(2,2),(3,2)\}$$

### 4.5 Implementation

Our implementation relies on parsing an XML document. The resulting stream produces opening and closing events that correspond exactly to $\texttt{events}(t)$ where $t$ is the tree representing this XML document. This stream can originate from an XML file, a database output, a network socket, or whatever.

For sake of clarity, we suppose that the two variables `stream_in` and `stream_out` representing respectively the input stream of events and the output stream of selected tuples are defined outside the scope of the main function, and are accessible everywhere in the code.

Figure 8 describes this interface and the core elements of the algorithm.

**Figure 8.** ⟨⟨ define earliest ⟩⟩

```
/*
    This function outputs every selected tuple σ of an
        input tree for a given query at their optimal
        event, if the input tree is supposed to be
        valid.
    The streams stream_in and stream_out are defined
        outside this function, and accessible
        everywhere in its code.
    A: a deterministic STA recognizing the canonical
        language of a MSO-definable n-ary query q
    D: a deterministic STA recognizing the schema
        language
*/
fun earliest(A,D)
  ⟨⟨ define safe(B) ⟩⟩
  ⟨⟨ define fail(B) ⟩⟩
  ⟨⟨ define processOpeningEvent(B) ⟩⟩
  ⟨⟨ define processClosingEvent(B) ⟩⟩
  ⟨⟨ precomputation(A,D) ⟩⟩  // builds B, accH_0 and accH
  ⟨⟨ initialize candidates(B) ⟩⟩
  forall e in stream_in.events()
    match e with
      case (open, π): processOpeningEvent(e)
      case (close, π): processClosingEvent(e)
end
```

The tag ⟨⟨ precomputation ⟩⟩ corresponds to the precomputation described in Section 4.3.

For each opening event, the function `processOpeningEvent` is called. This one, as depicted in Figure 9, tries to continue the run for each candidate by testing all possible new selections, tests optimality, and outputs every reached optimal event. Figure 10 presents the symmetric behaviour for dealing with closing events.

***Data structure*** The algorithm operates on an input tree $t \in T_\Sigma$, and has to compute the runs on all possible $n$-ary canonical labelings of $t$. As a consequence, it has to deal with a set of candidate tuples, for which no safe state and no failure state have been reached so far, but the following of the stream could lead to such a safe or failure states. The candidates are stored in the set `candidates`.

Figure 11 describes the initialization of this structure. Initially, no selection is done, we start from the initial state, safe states are final states and failure states are states from $F_0$, as introduced previously. Hence, `candidates` has only one element.

***Example.*** Figure 12 illustrates partially our algorithm on the example previously introduced: the query selecting nodes that have no next sibling, and a schema recognizing trees where the root is labeled by $a$, and every list of siblings is of the form $a^*b$. It shows how one candidate tuple is managed, and which safe and failure states are computed. We can notice that no failure states are reached, a first safe state is reached when opening the second node labeled by $b$, and then every following event remains safe.

**Figure 9.** ⟨⟨ define processOpeningEvent(B) ⟩⟩

```
/*
    This function processes an opening event, and
        writes on the output stream every selected n-
        tuple for which this event is optimal
    e: an opening event from the input stream
*/
fun processOpeningEvent(e)
  let
    a = label(e)
    π = node(e)
  in
    newCandidates ← ∅
    for (σ, p, S, F, s) ∈ candidates
      ⟨⟨ process opening event on the candidate (σ,p,S,F,s) ⟩⟩
    candidates ← newCandidates
end
```

**Figure 10.** ⟨⟨ define processClosingEvent(B) ⟩⟩

```
/*
    This function is similar to processOpeningEvent,
        but for closing events
    e: a closing event from the input stream
*/
fun processClosingEvent(e)
  let
    a = label(e)
    π = node(e)
  in
    newCandidates ← ∅
    for (σ, p, S, F, s) ∈ candidates
      ⟨⟨ process closing event on the candidate (σ,p,S,F,s) ⟩⟩
    candidates ← newCandidates
end
```

**Figure 11.** ⟨⟨ initialize candidates(B) ⟩⟩

```
candidates = {((∅, ..., ∅), init^B, final^B, F_0, emptyStack)}
```

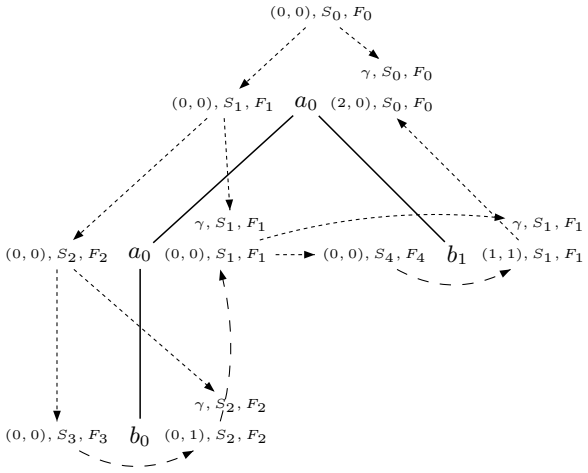
### 4.6 Complexity of the optimality problem

First, let us notice that our construction does not work directly if the input automaton is non-deterministic. This is mainly due to the fact that we cannot decompose the computation of safe (and failure) states to accessibility through a hedge and closing the current node. Hence, Proposition 9 does not hold for the non-deterministic case.

A way to avoid this is to consider accessibilty relation through sets of states. However, this would have the same complexity as a determinization step. As a consequence, if the entry is non-deterministic, we determinize it before running the algorithm.

**Theorem 1.** *The optimality problem (as described in Section 2) is DEXPTIME-complete. If the input automata are deterministic, this problem is in PTIME.*

*Proof.* First, let us notice that the optimality problem deals with only one candidate. We modify our algorithm in order to take a tuple as input and considering only candidates compatible with




(a) Run of our algorithm on an input tree for one candidate tuple

$S_0$={(0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 1), (3, 2)}
$S_1$=$S_2$={(0, 2), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 2)}
$S_3$={(0, 1), (0, 2), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 1), (3, 2)}
$S_4$={(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)}

$F_0$={(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2)}
$F_1$={(0, 1), (0, 2), (1, 0), (1, 2), (2, 2), (3, 0), (3, 1), (3, 2)}
$F_2$={(0, 2), (1, 0), (1, 2), (2, 2), (3, 0), (3, 1), (3, 2)}
$F_3$={(0, 1), (0, 2), (1, 1), (1, 2), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2)}
$F_4$={(0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2)}

(b) Sets involved in this run

**Figure 12.** A canonical tree and the corresponding run of our algorithm

this tuple. Considering the computation done by this algorithm until one event, the complexity in the deterministic case is a direct consequence of Proposition 8 and Proposition 10. So the problem is in PTIME in the deterministic case.

In the non-deterministic case, we just add a determinization step when computing $B$. The cost for this determinization is in DEXPTIME. As a consequence, we can solve the complexity problem in DEXPTIME. Hardness has been proved in Proposition 1 for bottom-up tree automata. Given a bottom-up tree automaton, its translation to an equivalent STA is in PTIME in the size of the input automaton. So the optimality problem is DEXPTIME-complete. □

### Conclusion

We have presented an algorithm for earliest query answering with polynomial time combined complexity, which applies to $n$-ary queries defined by deterministic STAs. We have shown that earliest query answering becomes DEXPTIME-complete in the nondeterministic case.

Future work includes a more precise study of space complexity. We hope to obtain a characterizations of queries that are suitable for streaming.

## References


[1] Rajeev Alur. Marrying words and trees. In *26th ACM Symposium on Principles of Database Systems*. ACM-Press, 2007.

[2] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *36th ACM Symposium on Theory of Computing*, pages 202–211. ACM-Press, 2004.

[3] Marcelo Arenas, Pablo Barcelo, and Leonid Libkin. Combining temporal logics for querying XML documents. In *International Conference on Database Theory*, pages 359-373, 2007.

[4] Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. Buffering in query evaluation over XML streams. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 216–227. 2005

[5] Michael Benedikt and Alan Jeffrey. Efficient and expressive tree filters. In *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science. 2007.

[6] Alexandru Berlea. Online evaluation of regular tree queries. *Nordic Journal of Computing*, 13(4):1–26, 2006.

[7] Alexandru Berlea. On-the-fly tuple selection for XQuery. In *Proceedings of the International Workshop on XQuery Implementation, Experience and Perspectives*, June 2007.

[8] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://tata.gforge.inria.fr. Edition from October 2007.

[9] Emmanuel Filiot, Joachim Niehren, Jean-Marc Talbot, and Sophie Tison. Polynomial time fragments of XPath with variables. In *26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 205–214. ACM-Press, 2007.

[10] Olivier Gauwin, Joachim Niehren, and Yves Roos. Streaming tree automata. Submitted, 2008.

[11] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[12] Ashish Kumar Gupta and Dan Suciu. Stream processing of XPath queries with predicates. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 419–430. ACM-Press, 2003.

[13] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Visibly pushdown automata for streaming XML. In *16th international conference on World Wide Web*, pages 1053–1062. ACM-Press, 2007.

[14] Holger Meuss, Klaus U. Schulz, and François Bry. Towards aggregated answers for semistructured data. In *Database Theory - ICDT 2001*, volume 1973 of *Lecture Notes in Computer Science*, pages 346–360. 2001.

[15] Andreas Neumann and Helmut Seidl. Locating matches of tree patterns in forests. In *Foundations of Software Technology and Theoretical Computer Science*, pages 134–145, 1998.

[16] Joachim Niehren, Laurent Planque, Jean-Marc Talbot, and Sophie Tison. N-ary queries by tree automata. In *10th International Symposium on Database Programming Languages*, volume 3774 of *Lecture Notes in Computer Science*, pages 217–231. 2005.

[17] Dan Olteanu. Spex: Streamed and progressive evaluation of XPath. *IEEE Trans. Knowl. Data Eng.* 19(7):934–949, 2007.

[18] Michael Schmidt, Stefanie Scherzinger, and Christoph Koch. Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In *23rd IEEE International Conference on Data Engineering (ICDE 07)*, 2007.

[19] Luc Segoufin and Victor Vianu. Validating streaming XML documents. In *Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 53–64, 2002.

[20] H. Seidl. Equivalence of finite-valued tree transducers is decidable. *Mathematical System Theory*, 27:285–346, 1994.

[21] H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.

[22] J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2:57–82, 1968.