Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach

Christoph Koch

Laboratory for Foundations of Computer Science
University of Edinburgh
Edinburgh EH9 3JZ, UK
koch@dbai.tuwien.ac.at

Abstract

We propose a new, highly scalable and efficient technique for evaluating node-selecting queries on XML trees which is based on recent advances in the theory of tree automata.

Our query processing techniques require only two *linear passes* over the XML data on disk, and their main memory requirements are in principle *independent of the size of the data*. The overall running time is $O(m + n)$, where m only depends on the query and n is the size of the data. The query language supported is very expressive and captures exactly all node-selecting queries answerable with only a bounded amount of memory (thus, all queries that can be answered by *any* form of finite-state system on XML trees). Visiting each tree node only twice is optimal, and current automata-based approaches to answering path queries on XML streams, which work using *one* linear scan of the stream, are considerably less expressive.

These technical results – which give rise to expressive query engines that deal more efficiently with large amounts of data in secondary storage – are complemented with an experimental evaluation of our work.

1 Introduction

A main reason for the real-world success of databases is that the data management research community has always focussed on providing utility (including expressive query languages) while above all keeping required computational resources such as processing time and main memory small. Main memory consumption is of particular importance to scalability, since the available main memory is fixed on a given machine and secondary storage as a back-up is comparably very slow.

XML documents can be naturally viewed as trees, and for most practical queries, this model is sufficient. Trees have very desirable computational properties, which distinguish them from the setting of relational databases. The theory community has developed an impressive body of work on processing trees, some of the most remarkable automata-theoretical. (See [15] for a survey of some of this work and its application to the theory of semistructured data and XML.) *Tree automata* can do many important tasks – including query processing by selecting nodes – while only consuming a bounded amount of memory[1] fixed with the automaton. Tree automata thus seem to be of obvious practical relevance to query processing. Now, why are tree automata not used for query processing in practice? It can be argued that they are.

Recently, finite word automata, in conjunction with a stack maintaining states for the nodes that constitute the path from the root to the current node, became a standard way of matching simple path queries on streaming XML (see e.g. [12]). The use of automata is a natural consequence of the need to bound the amount of memory required to process queries, which is essential in the context of streams because the size of the stream must be assumed to be extremely large, or even infinite.

Closer observation suggests that such automata should be best viewed as sequential-processing *implementations* of *deterministic top-down tree automata*. (Indeed, formally, how if at all could a real word automaton work on a tree?) Surprisingly, this viewpoint does not seem to have been taken in the data management literature before, even though it provides a source of inspiration for more expressive query lan-

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

[1]A cautionary note is in order here: Automata on trees usually assume a highly parallel model of computation in which essentially no memory is required. A top-down automaton run on a binary tree on a sequential computer can be effected using only a stack of working memory bounded by the depth of the tree.

guages that can be dealt with cheaply by automata.

There are intrinsic limitations to the expressiveness of queries handled by any form of automaton that processes a tree in the streaming fashion, by one linear scan from the start to the end. Data stream processors can basically only read each data item once, and cannot go back to a previously-read data item later. They cannot look forward in the stream to decide whether the node at the current position should be part of the query result, and equally cannot select past nodes on the basis of new information collected while reading the stream. As a consequence, node-selecting query languages for streams are restricted to *very simple* path queries.

The task of *boolean queries* [12, 3] is to decide to either accept or reject an entire XML document on the grounds of its contents, rather than selecting nodes in it. When it comes to such queries, the situation is slightly better (e.g., documents can be selected if they match queries of certain XPath fragments with conditions). Still, it is known that deterministic top-down tree automata are strictly less expressive than deterministic bottom-up tree automata [4]. However, a bottom-up traversal of the tree arriving in a stream can only be effected using *substantial* amounts of main memory depending on the branching factor of the tree. (XML trees tend to be shallow but wide.)

Node-selecting queries are as important outside the context of streams as they are there. In general, greater expressive power is required than what the streaming query language fragments can provide. Unfortunately, current techniques trade in expressiveness against query evaluation cost very dearly. As pointed out in [10], current systems for evaluating XPath 1 queries require each node to be visited an *exponential* number of times at worst (exponential in the size of the query). However, it is possible to improve this to a polynomial time worst-case bound [10]. For certain fragments of XPath that are still strictly larger than those supported by the stream processors, linear and quadratic bounds in terms of query size are known [10, 11]. Thus, even the best currently known algorithms have to read each part of the tree a possibly very large number of times into main memory, depending on the query.

1.1 Aim

For query processors that have to deal with very large amounts of data (that can only be held in secondary storage) to be truly scalable, a query evaluation technique should

1. read each node of the XML tree only a *very* small number of times (constant if possible),
2. not need to jump back and forth on disk intensively[2], and
3. be able to cope with a fixed-size memory buffer for the data, or a stack bounded by the depth of the XML tree.

[2] Even though this is essential for good query evaluation performance, it does not seem to have been considered enough in previous work on XML query processing.

The most desirable setting would be one in which only a constant number of linear scans[3] of the data on disk is required. This pays off considerably even if a database system with a good buffer manager is available, because (paged) sequential reading makes optimal use of the disk throughput.

1.2 Contributions

We present a novel technique that – surprisingly – has exactly these properties, and which works for a large and elegant class of queries which we discuss in more detail in Section 1.3. It is based on the notion of *selecting tree automata* (STA) recently proposed[4] in [8]. STAs only require a single nondeterministic bottom-up traversal of the tree to compute the result of a node-selecting query. Thus, STAs provide distinctive new capabilities for query evaluation.

The node selection criterion of STAs is slightly involved, and requires us to make the computation deterministic in practice. In this paper, we achieve this by translating the nondeterministic query evaluation of an STA into a two-phase technique in which we first have a run of a *deterministic* bottom-up tree automaton followed by a deterministic top-down tree automaton (very similar to those used on streaming XML) that runs on the tree of state assignments on nodes made by the bottom-up automaton. Thus, *two* traversals of the tree are required in total. The first pass streams state information *to the disk*, which the second pass reads. Free (temporary) disk space of size of the order of the size of the data is required for query evaluation, but apart from the space required to hold the two automata, only a stack bounded by the depth of the XML tree is required in main memory.

We also propose a new storage model for (XML) tree databases that allows to effect both top-down and bottom-up traversals with automata by just one linear scan each (forward for top-down and backward for bottom-up traversal).

Two scans are optimal, and any method that only needs to access each node once *must be strictly less expressive*. As we argue in Section 1.3, the additional expressive power is indeed important.

Our approach fulfills the three desiderata of Section 1.1. Moreover, the running time is $O(m + n)$, where m is a function only depending on the size of the query and n is the size of the data. As soon as the two automata for the query have been computed, query evaluation time is linear in the size of the data, with a very small constant, and *independent of the query.* As already for the restricted case of previous work on automata on streams, the size of our automata can be large, and they are best computed lazily [12].

Our techniques for constructing the automata are based on the observation that the set of all states an STA can possibly be in at a given node (as mentioned, STAs are nondeterministic) can be represented as a single residual propositional logic program (i.e., Horn formula), and that such programs are usually very

[3] That is, the consecutive reading of the data from the start to the end, as it is a given in stream processing.

[4] Independently and earlier, Frank Neven proposed the equivalent notion of *nondeterministic query automata* in his unpublished PhD thesis [14].

small. The development of scalable query processing techniques based on this observation is the main technical contribution of this paper.

Our techniques have been implemented in a high-performance query engine called *Arb* (motivated by the latin word for "tree"). As our experiments demonstrate, Arb scales very well and leads to extremely competitive performance for expressive queries on tree data in secondary storage.

No other similarly expressive framework for evaluating node-selecting queries with these properties is presently known. Previous automata-theoretic work does not constitute an exception to this observation. In particular, the *query automata* of [16], the only previous automata-theoretic formalism with the expressive power of STAs, may require to move up and down the tree an unbounded number of times.

It was independently observed in [17] that *boolean attribute grammars on ranked trees* (a formalism that captures the unary MSO queries) can be evaluated in just two passes of the data. The main difference to this work lies in our automata construction based on residual logic programs which – as we show in our experiments – scales to practical queries. Moreover, we consider XML and thus *unranked* trees in this paper.

1.3 Expressiveness of Queries: Scope

The class of queries handled by our framework is the one of all unary (i.e., node-selecting) queries definable in monadic second-order logic (MSO) over trees. MSO is a very well-studied language with beautiful theoretical properties (see [18, 7] for surveys) and considered to be highly expressive, at least from a popular viewpoint (cf. [16, 9]). MSO on trees captures the class of all queries computable using a bounded amount of memory. We will not need to introduce it in detail; a simpler but equivalently expressive query language, TMNF, will be used in the technical sections of this paper. Since MSO is more widely known, we will continue to speak of the MSO-definable rather than the TMNF-definable queries.

We introduce MSO abstractly, and try to give an intuition of its expressive power. Let us consider some queries definable in MSO.

1. MSO subsumes the XPath fragments usually considered in the streaming XML context, and *much* larger ones that support all XPath axes (including the upward axes such as "ancestor" and the sideways axes such as "following") and *branching* through paths combined using "and", "or", and "not" in conditions[5].
2. Let the XML document's text be included in the tree as *one node for each character*. Then,

   Select all nodes labeled "gene" that have a child labeled "sequence" whose text contains a substring matching the regular expression ACCGT(GA(C|G)ATT)*.

   is expressible in MSO. Just as we can apply such regular expressions on sets of sibling (character) nodes, they can be applied to paths in the tree. Indeed, all regular path queries [2] (on trees) are expressible in MSO. This generalizes to the so-called *caterpillar expressions* [5, 9], regular expressions over the alphabet of all the input relations that describe the tree. These allow for "recursive" walking in the tree – up, down, and sideways – and the checking of local conditions along the way. Examples will be provided at a later point in the paper.
3. Counting modulo a constant, as employed in the example query below, is expressible in MSO:

   Select all nodes labeled "publication" whose subtrees[6] contain an *even* number of nodes labeled "page" and a unique node labeled "abstract" which contains at most *100* whitespace-separated words of text.

4. Moreover, the selection of nodes based on universal properties, such as conformance of their subtrees with a DTD, can also be expressed.

These are only a few examples of expressible queries. The features discussed above can of course be combined to obtain further MSO-definable queries.

One application of the flexibility gained from this high expressiveness is in parallel query processing: Tree automata (working on binary trees) naturally admit parallel processing, but XML trees are usually very wide, and need to be translated into (somewhat) balanced binary trees first. As shown in [8], MSO is expressive enough to make this transformation transparent. (All queries can still be answered in the modified tree model.) A case study related to this issue is part of our experiments.

1.4 Structure

The structure of the paper is as follows. Section 2 contains initial definitions related to binary trees, which we need for our notions of tree automata to work on, and the TMNF language. In Section 3, we introduce the main definitions relating to tree automata that are used throughout the paper, including STAs. Section 4 contains our main results, defining the two-pass query evaluation algorithm on the basis of tree automata. We present our model of storing binary trees on disk in Section 5 and our experiments in Section 6. We conclude with Section 7.

2 Preliminaries

2.1 XML Documents as Binary Trees

XML documents are commonly thought of as node-labeled ordered trees. In practice, XML documents contain information beyond the tree structure, such as text, attributes, and comments. It is not difficult to model these features as part of the tree (assuming

[5]This fragment has been called Core XPath in [10]. The best algorithm known so far for evaluating Core XPath queries takes time linear in the size of the data and linear in the size of the queries, where each position in the tree has to be "visited" a linear number of times. The results in this paper improve greatly on this.

[6]By a node's subtree, we mean the part of the tree "below" it, including itself as the root node.

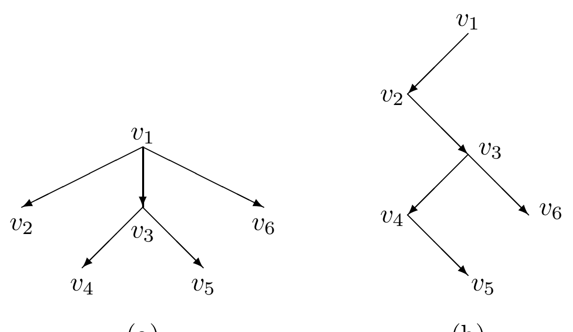

Figure 1: An unranked tree (a) and its corresponding binary tree version (b).

labels beyond the XML node *tags* to denote text characters etc.). As in the second MSO example above, we assume text strings to be represented as sibling nodes, one node for each character, ordered as in the text string.

XML trees are also *unranked*: each node may have an unlimited number of children. One straightforward way to interpret unranked trees as binary trees is to model the first child of a node in the unranked tree as the left child in the binary tree, and the right neighboring sibling in the unranked tree as the second child in the binary tree. (See Figure 1 for an example.) Often, no actual translation is necessary to obtain such a binary tree; XML data models such as DOM are naturally strikingly similar to this model.

Let σ be a set of labeling symbols on tree nodes, such as XML tags or ASCII characters. In the following, we model each binary tree $\mathbf{T}$ as a database consisting of unary relations $V^{\mathbf{T}}$, $Root^{\mathbf{T}}$, $HasFirstChild^{\mathbf{T}}$, $HasSecondChild^{\mathbf{T}}$ and $Label[l]^{\mathbf{T}}$ for each label $l \in \sigma$, and binary relations $FirstChild^{\mathbf{T}}$, and $SecondChild^{\mathbf{T}}$. As we will only deal with a single tree throughout the paper, we will subsequently omit the superscript $^{\mathbf{T}}$. Here, V is the set of nodes in $\mathbf{T}$, *Root* is the singleton set containing the root node, *HasFirstChild* and *HasSecondChild* denote the sets of nodes that have a first child and a second child, respectively, and $v \in Label[l]$ iff node v is labeled l. $\langle v, w\rangle \in FirstChild$ iff w is the first child of v and $\langle v, w\rangle \in SecondChild$ iff w is the second child of v.

2.2 Tree-Marking Normal Form (TMNF)

As mentioned, we do not directly use MSO as the query language of our framework, but a simpler formalism with the same expressive power, TMNF (*tree-marking normal form*) [9, 8]. TMNF is the computationally cheapest and structurally simplest language known that has the full expressive power of MSO over trees. TMNF has good computational properties, and is still powerful enough to formulate most practical queries very easily and concisely. TMNF programs can be evaluated in linear time in the size of the query and in linear time in the size of the data.

TMNF is to be understood as an internal formalism, to which queries in other languages such as XPath or special-purpose query languages from bio-informatics or linguistics can be efficiently translated and then evaluated as TMNF.

One way to view TMNF queries is as a *very* restricted form of datalog programs (with linear, rather than EXPTIME-complete query complexity as for datalog [1]). For this reason we will speak of TMNF *programs* rather than queries in this paper.

Below, we assume some familiarity with the datalog language (i.e., logic programming without function symbols), and refer to [1, 6] for detailed surveys.

A datalog program is called *monadic* if all its *IDB predicates* (that is, predicates that appear in rule heads somewhere in the program) are unary. A monadic datalog program of schema σ may use as *EDB predicates* (that is, input predicates) the relation names from Section 2.1, and a predicate $-U$ for each of the unary relation names U discussed there. $-U$ denotes the complement of the set of nodes that U stands for. Occasionally, when it is clear that we use the specific binary tree model discussed above, we refer to *SecondChild* as *NextSibling*, $-HasFirstChild$ as *Leaf*, and $-HasSecondChild$ as *LastSibling*.

We only use monadic datalog programs with a restricted syntax described next. In a TMNF program, each rule is an instance of one of the four rule templates (with "types" 1 to 4)

$$\begin{aligned} P(x) &\leftarrow U(x). && (1)\\ P(x) &\leftarrow P_0(x_0) \wedge B(x_0, x). && (2)\\ P(x_0) &\leftarrow P_0(x) \wedge B(x_0, x). && (3)\\ P(x) &\leftarrow P_1(x) \wedge P_2(x). && (4) \end{aligned}$$

where P, P_0, P_1, P_2 are IDB predicates and U, B are EDB predicates.

A program $\mathcal{P}$ can be seen as a node-selecting query by distinguishing one IDB predicate, calling it the *query predicate*. The query defined by $\mathcal{P}$ maps a tree $\mathbf{T}$ to the set of all vertices v such that $\mathcal{P}$ derives over $\mathbf{T}$ that v is in the goal predicate. Clearly, by distinguishing multiple IDB predicates, TMNF can compute several node-selecting queries together in one program.

Proposition 2.1 ([9]) *TMNF captures the unary MSO queries over trees.*

Because of the simple structure of TMNF rules, we can (and will) use the following variable-free syntax, in which rules of type (1) are written as P `:-` U`;`, rules of type (2) as P `:-` P_0`.`B`;`, rules of type (3) as P `:-` P_0`.inv`B`;`, and rules of type (4) as P `:-` P_1`,` P_2`;` This is syntax accepted by the Arb system.

The following example program may seem to be far off from XPath processing, but gives a good intuition of query evaluation in TMNF.

Example 2.2 The program shown below assigns the predicate "Even" to precisely those nodes whose subtree contains an even number of leaves labeled "a". The remaining nodes obtain the predicate "Odd". Intuitively, this is done by traversing the tree bottom-up, starting at the leaves, which are annotated "Even" or "Odd" first.

```
Even     :- Leaf, -Label[a];
Odd      :- Leaf, Label[a];
```

As auxiliary predicates we have "SFREven" and "SFROdd", attributed to node v if the sum of occurrences of "a" at leaves in the subtrees of v itself and its right siblings is even, respectively odd. These predicates are computed on a set of siblings that all either have predicate "Even" or "Odd" from the right, starting at the rightmost, last sibling ("SFR" = "siblings from right").

```
SFREven :- Even, LastSibling;
SFROdd  :- Odd, LastSibling;
```

In order to be able to move leftward through the lists of siblings, we define additional auxiliary predicates FSEven and FSOdd ("following sibling has predicate Even/Odd") that are assigned to nodes whose immediate right neighbor has predicate SFREven/SFROdd.

```
FSEven  :- SFREven.invNextSibling;
FSOdd   :- SFROdd.invNextSibling;
SFREven :- FSEven, Even;
SFROdd  :- FSEven, Odd;
SFROdd  :- FSOdd, Even;
SFREven :- FSOdd, Odd;
```

Whenever we arrive at a leftmost sibling (deriving either the predicate SFREven or SFROdd for it), we can push the information up to the parent:

```
Even    :- SFREven.invFirstChild;
Odd     :- SFROdd.invFirstChild;
```

Again, it must be emphasized that TMNF is mainly an internal language. A linear-time translation from a large XPath fragment to TMNF is given in [8].

However, as a convenient shortcut that is also supported in the query language of the Arb system, in rules of the form `Q :- P.R;`, `R` can be an arbitrary regular expression over our unary and binary input relations and their inverses (a *caterpillar expression*). The meaning of such rules is obvious. For instance,

```
Q :- P.FirstChild.NextSibling*.Label[a];
```

assigns the predicate Q to all nodes that have label "a" and are children of nodes with predicate P. Programs containing caterpillar expressions can be translated into strict TMNF in linear time [9].

3 Tree Automata

In this section, we introduce tree automata (e.g., [4]), which are a natural and elegant model of computation on trees. Tree automata are not really more difficult than their cousins on words. However, there are top-down and bottom-up flavors to such automata. We discuss the case of binary trees only, but all results and techniques extend immediately to trees of higher (fixed) rank. Moreover, as pointed out in Section 2, binary trees offer all the generality we need.

Definition 3.1 A non-deterministic (bottom-up) tree automaton is a tuple $\mathcal{A} = (Q, \Sigma, F, \delta)$, where

- Q is a finite set of *states*,
- Σ is the *alphabet*, a finite set of labeling symbols,
- $F \subseteq Q$ is the set of *accepting states*, and
- $\delta : \big((Q \cup \{\bot\}) \times (Q \cup \{\bot\}) \times \Sigma)\big) \to 2^Q$ is the *transition function*.

The special symbol $\bot$ is used as a "pseudo-state" for non-existent children.

A *run* of a bottom-up tree automaton $\mathcal{A}$ on tree $\mathbf{T}$ is a mapping $\rho : V^{\mathbf{T}} \to Q$ s.t. for each node $v \in V^{\mathbf{T}}$,

- If v is a leaf, $\rho(v) \in \delta\big(\bot, \bot, \Sigma(v)\big)$.
- If v has a left child v_1 but no right child,
$$\rho(v) \in \delta\big(\rho(v_1), \bot, \Sigma(v)\big).$$
- If v has a right child v_2 but no left child,
$$\rho(v) \in \delta\big(\bot, \rho(v_2), \Sigma(v)\big).$$
- If v has both a left child v_1 and right child v_2,
$$\rho(v) \in \delta\big(\rho(v_1), \rho(v_2), \Sigma(v)\big).$$

where $\Sigma(v) \in \Sigma$ denotes the label of node v of tree $\mathbf{T}$.

The run ρ is called *accepting* if $\rho(root^{\mathbf{T}}) \in F$. The automaton $\mathcal{A}$ *accepts* $\mathbf{T}$ if there is an accepting run for $\mathcal{A}$ on $\mathbf{T}$.

Deterministic bottom-up tree automata only differ from this in that the transition function is of the form

$$\delta : \big((Q \cup \{\bot\}) \times (Q \cup \{\bot\}) \times \Sigma)\big) \to Q,$$

i.e. maps to one state rather than a set of states. Therefore, there is *exactly one* run on each tree.

So far, our automata can only decide *boolean queries*. To be able to define unary queries, we need to enhance tree automata by an additional mechanism for selecting nodes.

Definition 3.2 ([8]) A *selecting tree automaton (STA)* is a tuple $\mathcal{A} = (Q, \Sigma, F, \delta, S)$, where (Q, Σ, F, δ) is a nondeterministic bottom-up tree automaton and $S \subseteq Q$ is a set of *selecting states*. The unary query defined by an STA $\mathcal{A}$ maps every tree $\mathbf{T}$ to the set

$$\mathcal{A}(\mathbf{T}) = \big\{v \in V^{\mathbf{T}} \mid \rho(v) \in S \text{ for every accepting run } \rho \text{ of } \mathcal{A} \text{ on } \mathbf{T}\big\}.$$

In other words, we select the node $v \in V^{\mathbf{T}}$ if and only if every accepting run of $\mathcal{A}$ on $\mathbf{T}$ is in a selecting state at vertex v.

Even though – at least when judging from their concise definition – STAs are quite simple, they are surprisingly expressive: The node-selecting queries definable using STAs are precisely those definable in MSO.

Proposition 3.3 ([8]) *STAs capture the unary MSO queries on trees.*

We will only use a very weak form of *deterministic top-down tree automaton* $\mathcal{B} = (Q^{\mathcal{B}}, \Sigma^{\mathcal{B}}, s^{\mathcal{B}}, \delta_1^{\mathcal{B}}, \delta_2^{\mathcal{B}})$ with individual transition functions $\delta_k : Q^{\mathcal{B}} \times \Sigma^{\mathcal{B}} \to Q^{\mathcal{B}}$ for the two children ($k \in \{1, 2\}$) and without an acceptance condition (i.e., all states of $Q^{\mathcal{B}}$ are final states). $s \in Q^{\mathcal{B}}$ is the start state assigned to the root node, $\rho^{\mathcal{B}}(Root^{\mathbf{T}}) = s$. The sole purpose of such an automaton $\mathcal{B}$ is to annotate the tree nodes with states via its run $\rho^{\mathcal{B}} : V^{\mathbf{T}} \to Q^{\mathcal{B}}$, whose definition is obvious.

4 Two-Phase Query Evaluation

The main issue about STAs is their nondeterministic nature. There are possibly many alternative runs, and the node-selection criterion is such that a node v is in the query result if and only if v is in a selecting state for all of the accepting runs.

By a single bottom-up run through tree $\mathbf{T}$, we can compute the set of states reachable in some run for every node of $\mathbf{T}$. This "powerset" construction, in which *sets of states* in a nondeterministic automaton become *states* in a deterministic automaton, comes close to making STAs deterministic, because in the standard translation from TMNF to STA proposed in [8], all states are final states ($F = Q$), and thus all possible runs are accepting. However, the reachable states do not yet represent the solution[7], because there may be states from which the root node cannot be reached.

To deal with this, we proceed in two phases. First, we compute the reachable states on each node in a bottom-up run, and then return in a top-down run to prune them. In the second phase, we also apply the selection criterion of the automaton and compute those predicates that occur in all remaining states.

The two phases can be easily modeled as deterministic tree automata. The bottom-up phase is done using a deterministic bottom-up tree automaton

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

with $Q^{\mathcal{A}} \subseteq 2^{2^{IDB(\mathcal{P})}}$ and $\Sigma^{\mathcal{A}} = 2^{\sigma}$, i.e. the alphabet is the set of subsets of the schema σ. The (unique) run $\rho^{\mathcal{A}}$ of $\mathcal{A}$ assigns to each node the set of reachable states of the corresponding STA.

The top-down phase consists of a run of a deterministic top-down tree automaton

$$\mathcal{B} = (Q^{\mathcal{B}}, \Sigma^{\mathcal{B}}, s^{\mathcal{B}}, \delta_1^{\mathcal{B}}, \delta_2^{\mathcal{B}}).$$

s.t. $\Sigma^{\mathcal{B}} = Q^{\mathcal{A}}$, $Q^{\mathcal{B}} = 2^{IDB(\mathcal{P})}$, and $s^{\mathcal{B}} = \bigcap \rho^{\mathcal{A}}(Root^{\mathbf{T}})$.

The state assignments $\rho^{\mathcal{B}}(v)$ made by the run $\rho^{\mathcal{B}} : V^{\mathbf{T}} \to 2^{IDB(\mathcal{P})}$ of top-down automaton $\mathcal{B}$ on each node v of the tree $\mathbf{T}$ labeled using $\rho^{\mathcal{A}} : V^{\mathbf{T}} \to Q^{\mathcal{A}}$ will be precisely the set of IDB predicates assigned to v in the evaluation result $\mathcal{P}(\mathbf{T})$ of TMNF program $\mathcal{P}$ on $\mathbf{T}$.

Theorem 4.1 $P \in \rho^{\mathcal{B}}(v) \Leftrightarrow P(v) \in \mathcal{P}(\mathbf{T})$.

In this section, our primary goal is to provide good algorithms for computing the transition functions of these two automata; we will assign more readable names to them as follows:

$$\begin{aligned} \delta^{\mathcal{A}} &= \mathit{ComputeReachableStates} \\ \delta_k^{\mathcal{B}} &= \mathit{ComputeTruePreds}_k \quad (k \in \{1,2\}) \end{aligned}$$

These are the two[8] main procedures that we will develop throughout this section.

[7]Meaning, for node v, the predicates occurring in state $\rho(v)$ for all runs ρ of the STA.

[8]*ComputeTruePreds*$_k$ can of course be defined as a single procedure that takes k as an argument.

The main claim to the practical relevance of this approach follows from the surprising fact that *sets of reachable states* – rather than single states themselves – can be represented very concisely as propositional logic programs (a.k.a. propositional Horn formulae).

4.1 Propositional Logic Programs

For lack of space, we have to refer to any logics textbook for definitions on propositional logic programs.

Let $\mathcal{P}$ be a (propositional logic) program. By LTUR($\mathcal{P}$), we denote the *residual program* obtained as follows.

1. We compute the set M of all propositional predicates derivable from $\mathcal{P}$, i.e. which follow from facts (rules with empty body) in $\mathcal{P}$ using rules in $\mathcal{P}$.
2. We drop all rules of $\mathcal{P}$ whose heads are true (i.e., in M) or which contain an EDB predicate in the body that is not in M.
3. We remove all body predicates of remaining rules that are true (i.e., in M).
4. We insert each IDB predicate $p \in M$ as a new fact $p \leftarrow$.

LTUR($\mathcal{P}$) can be computed in time O($|\mathcal{P}|$) using Minoux' linear-time unit resolution algorithm [13][9].

Definition 4.2 Let $\mathcal{P}$ be a TMNF program with $IDB(\mathcal{P}) = \{X_1, \ldots, X_\ell\}$ and where σ consists of the unary EDB predicates in $\mathcal{P}$. By *PropLocal*($\mathcal{P}$), we denote the propositional program in the propositional predicates $\sigma \cup \{X_i, X_i^1, X_i^2 \mid 1 \leq i \leq \ell\}$[10] with the following propositional Horn clauses (rules):

1. If X_i `:-` R`;` is a rule of $\mathcal{P}$ then $X_i \leftarrow R$ is a clause of *PropLocal*($\mathcal{P}$).
2. If X_i `:-` X_j `,` X_k`;` is a rule of $\mathcal{P}$ then $X_i \leftarrow X_j \wedge X_k$ is a clause of *PropLocal*($\mathcal{P}$).
3. If X_i `:-` X_j`.invFirstChild;` is a rule of $\mathcal{P}$ then $X_i \leftarrow X_j^1$ is a clause of *PropLocal*($\mathcal{P}$).
4. If X_i `:-` X_j`.invSecondChild;` is a rule of $\mathcal{P}$ then $X_i \leftarrow X_j^2$ is a clause of *PropLocal*($\mathcal{P}$).
5. If X_i `:-` X_j`.FirstChild;` is a rule of $\mathcal{P}$ then $X_i^1 \leftarrow X_j$ is a clause of *PropLocal*($\mathcal{P}$).
6. If X_i `:-` X_j`.SecondChild;` is a rule of $\mathcal{P}$ then $X_i^2 \leftarrow X_j$ is a clause of *PropLocal*($\mathcal{P}$).

Given a TMNF program $\mathcal{P}$, local_rules denotes the rules in *PropLocal*($\mathcal{P}$) obtained using the bullet points (1) and (2) of Definition 4.2, left_rules denotes those produced using (3) and (5), right_rules those obtained from (4) and (6), downward_rules$_1$ those using (5), and downward_rules$_2$ those using (6). By "left-child" and "right-child" predicates, we denote those with "1" and "2" as superscript, respectively, and by "local" predicates those without a superscript.

[9]Only (1) is covered in [13], but (2)-(4) is straightforward.

[10]Relation names are considered as propositional predicates.

Example 4.3 We start a running example used throughout this section. For the TMNF program $\mathcal{P}$

$$\begin{array}{rcl} P_1 & \texttt{:-} & \texttt{Root;} \\ P_2 & \texttt{:-} & P_1\texttt{.FirstChild;} \\ P_3 & \texttt{:-} & P_2\texttt{.FirstChild;} \\ P_4 & \texttt{:-} & P_3\texttt{, Leaf;} \\ P_5 & \texttt{:-} & P_4\texttt{.invFirstChild;} \\ Q & \texttt{:-} & P_5\texttt{.invFirstChild;} \end{array}$$

we have

$$\begin{aligned} \mathit{PropLocal}(\mathcal{P}) &= \{P_1 \leftarrow \mathit{Root};\ \ P_2^1 \leftarrow P_1; \\ &\quad P_3^1 \leftarrow P_2;\ \ P_4 \leftarrow P_3 \wedge \mathit{Leaf}; \\ &\quad P_5 \leftarrow P_4^1;\ \ Q \leftarrow P_5^1\}. \\ \text{local\_rules} &= \{P_1 \leftarrow \mathit{Root};\ \ P_4 \leftarrow P_3 \wedge \mathit{Leaf};\}, \\ \text{left\_rules} &= \{P_2^1 \leftarrow P_1;\ \ P_3^1 \leftarrow P_2; \\ &\quad P_5 \leftarrow P_4^1;\ \ Q \leftarrow P_5^1\}, \\ \text{right\_rules} &= \emptyset, \\ \text{downward\_rules}_1 &= \{P_2^1 \leftarrow P_1;\ \ P_3^1 \leftarrow P_2;\}, \\ \text{downward\_rules}_2 &= \emptyset. \end{aligned}$$

In what follows, we will exclusively deal with propositional programs. That is, whenever we speak of a program, we will always refer to a propositional one.

We need the following auxiliary definitions. PredsAsRules(S) translates a set of predicates $S = \{X_1, \ldots, X_n\}$ into a set of rules $\{X_1 \leftarrow;\ \ldots\ X_n \leftarrow\}$. TruePreds($\mathcal{P}$) computes the set of predicates already known to be true in $\mathcal{P}$, i.e. those predicates X for which $X \leftarrow;$ is a rule in $\mathcal{P}$.

Let $k \in \{1, 2\}$. PushDown$_k$($\mathcal{P}$) assumes that each predicate appearing in $\mathcal{P}$ is local and adds the superscript ^{k} to each of them. Given a set of predicates S, Preds$_k$(S) outputs those with superscript ^{k}. Given a set of predicates S that all have superscript ^{k}, PushUpFrom$_k$ removes these superscripts.

Given a program $\mathcal{P}$, ContractProgram($\mathcal{P}$) is computed as follows. We unfold two rules r_1 and r_2 if head(r_2) $\in$ body(r_1) and head(r_2) has a superscript (^{1} or ^{2}). (Unfolding means to replace head(r_2) in body(r_1) by body(r_2), creating a new rule). This is done until no new rules can be computed. Then, all rules containing a predicate with superscript ^{1} or ^{2} are removed. The rules that remain are all local.

Even in the theoretical worst case, there are "only" exponentially many different rules, because a rule body is a *subset* of a fixed set of predicates. Our concise description conceals that better implementations of the ContractProgram procedure are possible. In the important case that the bodies of all rules contain at most one IDB predicate, the computation required is very similar to the transitive closure of a binary relation from the result of which some tuples are removed.

Example 4.4 The propositional program

$$\begin{array}{lll} \{P_0 \leftarrow P_1 \wedge P_2; & P_1 \leftarrow P_3^1; & P_2 \leftarrow P_4^1; \\ \quad P_3^1 \leftarrow P_5^1; & P_4^1 \leftarrow P_5^1 \wedge P_6^1; & P_5^1 \leftarrow P_7; \\ P_6^1 \leftarrow P_7 \wedge P_8; & P_8 \leftarrow P_9^2 \wedge P_{10}^2; & P_9^2 \leftarrow P_{11};\} \end{array}$$

```
procedure ComputeReachableStates(
    Program* $P^1_{res}$, Program* $P^2_{res}$, SetOfPreds labels)
returns Program
begin
    Program $P$ := local_rules ∪ PredsAsRules(labels);

    if ($P^1_{res} \neq \bot$) then
        $P$ := $P$ ∪ left_rules ∪ PushDown$_1$($P^1_{res}$);

    if ($P^2_{res} \neq \bot$) then
        $P$ := $P$ ∪ right_rules ∪ PushDown$_2$($P^2_{res}$);

    $P$ := LTUR($P$);

    if (($P^1_{res} \neq \bot$) or ($P^2_{res} \neq \bot$)) then
        $P$ := ContractProgram($P$);

    return $P$;
end.
```

Figure 2: The ComputeReachableStates algorithm.

contracts to $\{P_0 \leftarrow P_1 \wedge P_2;\ P_1 \leftarrow P_7;\ P_2 \leftarrow P_7 \wedge P_8;\}$.

4.2 The Bottom-up Phase

Figure 2 presents the ComputeReachableStates procedure, which computes the transitions of our deterministic bottom-up automaton $\mathcal{A}$. Sets of reachable states for a node n are represented as (propositional) *residual programs* that contain all the restrictions on subsets of $\mathit{IDB}(\mathcal{P})$ inferable using the information in the subtree below n.[11]

Example 4.5 Consider the three-node tree obtained from the XML document

$$\langle a\rangle\ \ \langle a\rangle\ \ \langle a/\rangle\ \ \langle/a\rangle\ \ \langle/a\rangle$$

Let v_0 be the root node, v_1 its child, and v_2 the child of v_1, the single leaf. We have

$$\begin{aligned} \Sigma^{\mathcal{A}}(v_0) &= \{\mathit{Root}, \mathit{HasFirstChild}, -\mathit{HasSecondChild}, a\}, \\ \Sigma^{\mathcal{A}}(v_1) &= \{\mathit{HasFirstChild}, -\mathit{HasSecondChild}, a\}, \\ \Sigma^{\mathcal{A}}(v_2) &= \{-\mathit{HasFirstChild}, -\mathit{HasSecondChild}, a\}. \end{aligned}$$

We re-use the program $\mathit{PropLocal}(\mathcal{P})$ of Example 4.3. Through the procedure ComputeReachableStates of Figure 2, we obtain the residual programs

$$\begin{array}{rcl} \rho^{\mathcal{A}}(v_2) & = & \{P_4 \leftarrow P_3\}, \\ \rho^{\mathcal{A}}(v_1) & = & \{P_5 \leftarrow P_2\}, \\ \rho^{\mathcal{A}}(v_0) & = & \{P_1 \leftarrow;\ \ Q \leftarrow\}. \end{array}$$

Note that, for instance, the state $\rho^{\mathcal{A}}(v_2)$ of our deterministic bottom-up tree automaton $\mathcal{A}$ encodes $2^{|\mathit{IDB}(\mathcal{P})|-2} * (2^2 - 1) = 48$ STA states (all possible truth assignments to the six IDB predicates are possible, except those where P_4 is false and P_3 is true at the same time).

[11]Note that the residual programs returned by LTUR are by its definition guaranteed not to contain EDB predicates.

```
procedure ComputeTruePreds(
    SetOfPreds parent_preds,
    Program $P_{res}$, $k \in \{1, 2\}$)
returns SetOfPreds
begin
    Program $P$ := downward_rules$_k$ $\cup$
                   PredsAsRules(parent_preds) $\cup$
                   PushDown$_k$($P_{res}$);

    SetOfPreds $S$ := TruePreds(LTUR($P$));

    return PushUpFrom$_k$(Preds$_k$($S$));
end.
```

Figure 3: The ComputeTruePreds algorithm.

To illustrate the procedure of Figure 2, we compute $\rho^{\mathcal{A}}(v_1)$ using the steps described there as

$$\text{ComputeReachableStates}(\{P_4 \leftarrow P_3\}, \bot, \{\mathit{HasFirstChild}, -\mathit{HasSecondChild}, a\}).$$

Initially, program P is set to

$$\text{local\_rules} \cup \text{PredsAsRules(labels)} = \{P_1 \leftarrow \mathit{Root};\ \ P_4 \leftarrow P_3 \wedge \mathit{Leaf};\} \cup \{\mathit{HasFirstChild} \leftarrow;\ \ -\mathit{HasSecondChild} \leftarrow;\ \ a \leftarrow\}.$$

Since there is a first child with residual program $P^1_{res} = \{P_4 \leftarrow P_3\}$, we add

$$\text{left\_rules} \cup \text{PushDownLeft}(\{P_4 \leftarrow P_3\}) = \{P^1_2 \leftarrow P_1;\ \ P^1_3 \leftarrow P_2;\ \ P_5 \leftarrow P^1_4;\ \ Q \leftarrow P^1_5\} \cup \{P^1_4 \leftarrow P^1_3\}$$

to P. Next, we compute LTUR(P), which results in

$$\{P^1_2 \leftarrow P_1;\ P^1_3 \leftarrow P_2;\ P_5 \leftarrow P^1_4;\ Q \leftarrow P^1_5;\ P^1_4 \leftarrow P^1_3;\ \mathit{HasFirstChild} \leftarrow;\ \mathit{HasSecondChild} \leftarrow;\ a \leftarrow\}$$

After removing the EDB predicate rules, in ContractProgram, the only unfolding of this program that we can make to obtain a rule over exclusively local predicates is of $P_5 \leftarrow P^1_4$; $P^1_4 \leftarrow P^1_3$; $P^1_3 \leftarrow P_2$ into $P_5 \leftarrow P_2$. Thus, $\rho^{\mathcal{A}}(v_1) = \{P_5 \leftarrow P_2\}$.

4.3 The Top-down Phase

As the ComputeReachableStates procedure on a given node v uses all the information available below v, when reaching the root, all the information in the tree has been used to obtain the set of reachable STA states for v, represented as a residual program P. Thus, TruePreds(P) constitutes the solution of the TMNF program restricted to the root node (the minimum fixpoint semantics that we usually attribute to logic programs – including those in TMNF – corresponds nicely to the selection criterion of STAs, which is to select those predicates that are in the state assignments of *all* runs on the node).

Using the procedure ComputeTruePreds of Figure 3, we can now compute the minimum fixpoint resp. true predicates on the nodes below by a top-down run.

Algorithm 4.6 (Two-phase query evaluation)
Input: A binary node-labeled tree $\mathbf{T}$ and a TMNF program $\mathcal{P}$.
Output: $\mathcal{P}(\mathbf{T})$ as the true predicates for each node.

1. Compute[12] the run $\rho^{\mathcal{A}}$ of bottom-up automaton $\mathcal{A}$, whose transition function can be obtained lazily using the ComputeReachableStates procedure, on $\mathbf{T}$ starting at the leaves whose (nonexistent) children have residual program $\bot$.

2. On reaching the root, extract the true predicates from its residual program as TruePreds($\rho^{\mathcal{A}}(\mathit{Root})$) (this amounts to computing the set of predicates true in all states).

3. Beginning with the true predicates of the root node as the start state $s^{\mathcal{B}}$, compute the run $\rho^{\mathcal{B}}$ of the top-down tree automaton $\mathcal{B}$ on $\mathbf{T}$, which assigns the set of true predicates to each node.

   The transitions of $\delta^{\mathcal{B}}_k$ ($k \in \{1, 2\}$) are computed lazily using ComputeTruePreds(S, P_{res}, k).

The correctness proof for this algorithm is not difficult, but is beyond the scope of this paper.

Example 4.7 We continue where we stopped in Example 4.5. In the top-down traversal using the ComputeTruePreds procedure of Figure 3, we obtain as precise state descriptions

1. $\{P_1, Q\}$ for v_0, because this is the set of predicates true in all reachable states and all the information in the tree constraining the states at the root node has been deployed,

2. $\{P_2, P_5\}$ for v_1, because on descending from v_0, ComputeTruePreds($\{P_1, Q\}$, $\{P_5 \leftarrow P_2\}$, 1) has

   $$P = \{P^1_2 \leftarrow P_1; P^1_3 \leftarrow P_2; P_1 \leftarrow; Q \leftarrow; P^1_5 \leftarrow P^1_2\},$$

   which simplifies to

   $$\text{LTUR}(P) = \{P_1 \leftarrow; Q \leftarrow; P^1_2 \leftarrow; P^1_5 \leftarrow; P^1_3 \leftarrow P_2\},$$

   thus $S = \{P_1, Q, P^1_2, P^1_5\}$, Preds$_1(S) = \{P^1_2, P^1_5\}$, PushUpFrom$_1(\{P^1_2, P^1_5\}) = \{P_2, P_5\}$, and

3. $\{P_3, P_4\}$ for v_2, analogously.

5 The Arb Storage Model

An essential requirement posed by our algorithms is that it must be possible to process the data tree (that is, the tree consisting of both element and character nodes.) both top-down and bottom-up, and to access it as a binary tree as described in Section 2. Now, such traversals can be effected with a linear scan of an XML document only if an auxiliary stack is available that can grow to sizes that depend on the maximum

[12]Since the run of $\mathcal{A}$ may be very large and $\mathcal{B}$ needs to process it, we write it to the disk. In our implementation, we write the pointer to the internal data structure of the residual program $\rho^{\mathcal{A}}(v)$ for each node v, in the order we visit the nodes. Our temporary file thus consumes four bytes per node.

number of children a node may have (which may be huge), apart from the depth of the tree. In order to be scalable, this has to be avoided.

Another reason for an internal storage model is that the data must be accessed twice for each run of the query processor. The storage model we propose below allows to do both top-down and bottom-up traversals faster than parsing runs on the XML document and takes main memory proportional to the depth of the XML tree at most for these operations.

The model for storing binary trees on disk used by Arb is as follows. Each node v is stored as a fixed-size field of k bytes on disk in which the two highest bits denote whether v has a first and/or a second child and the remaining $8 * k - 2$ bits are used to hold an integer denoting the label of v. The nodes are stored in the database in pre-order, where n recursively precedes all of the nodes in its subtree, and the nodes in the subtree of the first child (if it exists) precede all of the nodes in the subtree of the second child (if it exists). This order can be produced by a simple top-down traversal and is analogous to *document order* in XML (cf. [19]). For example, the tree of Figure 1 (b) is represented as

l_1	l_2	l_3	l_4	l_5	l_6
1 0	0 1	1 1	0 1	0 0	0 0

where the l_i denote the labels of the nodes v_i.

In our implementation, by default, $k = 2$, and the tree can therefore contain $2^{14} = 16,384$ different labels. The indexes 0 to 255 are reserved for text characters[13]. For each label that is not a text character, we store its name in a separate (".lab") file. The name of a label with index $i \geq 256$ is the $(i-255)$th (whitespace-separated) entry in the .lab file.

We create such .arb databases in two passes. In a first pass, we make a SAX parsing run through the XML document to count the total number n of nodes and write the SAX events to a file. Then we create a new file – the .arb database – and start writing it backwards, beginning at an offset of $k * n$ bytes, while reading our SAX events file backward. In this single backward pass, we can transform the document into a binary tree (as described in Section 2) and only require a stack of memory proportional to the depth of the XML tree (rather than the depth of the binary tree-version of, which may be extremely right-deep) to do it. We also write the .lab file with the label names (mostly tag names).

Proposition 5.1 *Let* $\mathbf{T}$ *be an XML tree and* $ARB(\mathbf{T})$ *be its .arb database. Then, its binary-tree version* $B(\mathbf{T})$ *can be traversed*

- *top-down by reading* $ARB(\mathbf{T})$ *in one linear scan from left to right with only a stack of size* $O(depth(\mathbf{T}))$ *and*
- *bottom-up by reading* $ARB(\mathbf{T})$ *in one linear scan from right to left with a stack of the same size.*

[13]As mentioned above, in our model, text characters are stored as part of the tree, rather than separately.

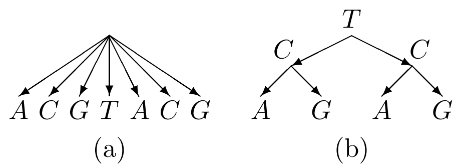

Figure 4: Two trees representing sequence ACGTACG: "flat" (a) and "infix" (b).

A precise description of the required techniques for database creation and the two forms of tree traversal are beyond the scope of this paper, but are not difficult to obtain given the description provided.

6 Experiments

In this section, we present some of our experiments with the Arb system, a C++ implementation of our techniques that strictly follows the algorithms described. All experiments were carried out on a Dell Inspiron 8100 laptop with 256 MB of RAM and a 1 GHz Pentium III processor running Linux.

6.1 Tree Databases

We first introduce the three XML databases used in our experiments.

1. Penn Treebank, the linguistic database.
2. Swissprot, the protein database.
3. ACGT, a bogus DNA database consisting of a randomly generated sequence of $2^{25} - 1 = 33,554,431$ symbols from the alphabet $\{A, C, G, T\}$. Two XML versions of it were created,
   - one with a root node with one child for each symbol of the sequence, in the order of the sequence from left to right (called ACGT-flat in the following) and
   - one in which a complete binary infix tree (of depth 24) was generated, below a separate root node (called ACGT-infix below).

   To communicate the idea, we show two such trees for sequence ACGTACG (i.e., of length $2^3 - 1$) in Figure 4. It is clear that almost complete infix trees can be created for sequences of arbitrary length.

Figure 5 provides information on the creation of the .arb databases. Columns (1) and (2) show the numbers of element and character nodes inserted, respectively. Note that our source XML documents contain no other kinds of nodes (w.r.t. the DOM classification), such as attribute or comment nodes. The number of tags is shown in column (3). This number does not include the ASCII characters used to label the character nodes. Column (4) shows the overall time required for creating a .arb database from the XML file. Columns (5) and (6) show the sizes of the files created, and (7) the size required to hold the temporary .evt for holding the sax events, to be processed backwards to compute the .arb file. Note that (5) is always $((1) + (2)) * 2$, as each node takes precisely two bytes, and (7) is twice as much. (We use two bytes for each event and two events – a "begin" and an "end" event for each node.)

	elem nodes	char nodes	tags	time	.arb file size	.lab file size	.evt file size
unit	#	#	#	seconds	bytes	bytes	bytes
column ref.	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Treebank	2,447,728	29,337,845	251	49.01	63,571,146	1,716	127,142,292
ACGT-infix	33,554,432	0	4	134.06	67,108,864	14	134,217,728
ACGT-flat	33,554,432	0	4	76.77	67,108,864	14	134,217,728
SWISSPROT	10,903,569	296,563,873	48	559.82	614,934,884	302	1,229,869,768

Figure 5: Statistics on .arb database creation.

6.2 Benchmark Queries

The experimental activities reported on here consist of three basic threads:

1. Top-down regular path queries.

   We used randomly generated regular path queries on Treebank.

   All regular expressions, in this experiment and those below (i.e., the second and third) were always of the form $w_1.w_2^*.w_3$, where the w_i were sequences of symbols over the alphabet $\{NP, VP, PP, S\}$[14] of length at least one. By the size of such a regular expression, we mean $|w_1| + |w_2| + |w_3|$. An example of a regular expression of length five is $S.VP.(NP.PP)^*.NP$. Such queries were written as (single-rule) programs in our extended syntax as

   ```
   QUERY :- V.Label[S].R.Label[VP].
            (R.Label[NP].R.Label[PP])*.
            R.Label[NP];
   ```

   where R is short for `FirstChild.NextSibling*`.

   The point of this experiment was to assess the performance of our approach on queries intuitively deriving predicates downwards. As we start with a bottom-up automaton run, we have to deal with much incomplete information.

2. Bottom-up regular path queries.

   As above, we randomly generated regular expressions over the alphabet $\{A, C, G, T\}$ and with R = `invNextSibling`. These regular expressions were matched in the ACGT-flat database.

   The main goal of this experiment is to provide a yardstick for the next.

3. (Sideways) Caterpillar queries.

   We matched the same regular expressions over alphabet $\{A, C, G, T\}$, now with

   ```
   R = (FirstChild.SecondChild*.
              -hasSecondChild |
       -hasFirstChild.invFirstChild*.
                          invSecondChild)
   ```

   in ACGT-infix. It can be verified that this caterpillar expression R "walks" our infix tree to always find the symbol immediately previous to the one it starts from in the sequence.

   The queries needed to do this make almost full use of the expressiveness provided in our formalism. However, the queries do not have any branches.

[14]NP = noun phrase; VP = verb phrase, PP = prepositional phrase; S = sentence.

The second and third classes of queries model the same regular expressions on strings first represented in sequence (ACGT-flat, where character nodes are linked in a very long *NextSibling* list, and thus an extremely right-deep tree) and second as a balanced binary tree (ACGT-infix). This has an important application. Since the automata-based model of computation on trees is intrinsically parallel (computations in distinct subtrees are completely independent), it can also be easily implemented on real parallel computers. A requirement for parallelization, however, is that the binary trees are balanced. This is not the case in ACGT-flat, but it is in ACGT-infix. However, the need to translate regular expressions on words to our balanced trees provides an interesting application of the expressive power of our query language.

Our techniques provide an algorithm for parallel regular expression matching which (assuming the regular expression fixed and the text string of length n) runs in parallel time $O(\log n)$. It can be shown that this time bound can be achieved with $O(n)$ total work. Using the terminology of parallel processing, this is thus an optimal parallel algorithm.

Another binary tree model in which in practice also XML (element node) trees tend to be well-balanced is provided in [8]. A translation of queries to work transparently on this model (by replacing relations representing the tree structure by appropriate caterpillar expressions that simulate the given queries in the new model) is also provided there. That tree model is supported in Arb as well, and can be activated as an option. Indeed, the database ACGT-infix.arb was *created* from an XML document using this binary tree model.

6.3 Discussion

The information in Figure 6 reads as follows. Each row represents averages for 25 randomly generated regular path queries of length indicated in column (1). The numbers of IDB predicates and numbers of rules in the internal TMNF programs are shown in columns (2) and (3) respectively. The times taken for the bottom-up phase are presented in column (4), and the numbers of transitions computed lazily for the bottom-up tree automaton are given in (5). Columns (6) and (7) show the analogous running times and transition counts for the top-down automaton. The overall times taken from start to termination of Arb are given in (8). (9) shows how many nodes were assigned the query predicate, and thus selected by the query. (10) shows the average maximum amount of main memory taken by Arb during its running time.

As mentioned, all numbers are averages over 25 runs. Note that the same 25 regular expressions were always used to create the ACGT-infix and ACGT-flat

Query			Phase 1 (BU)		Phase 2 (TD)		Global characteristics		
size #	$\lvert IDB \rvert$ #	$\lvert\mathcal{P}\rvert$ #	time sec.	transitions #	time sec.	transitions #	time sec.	selected #	mem kbytes
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
Treebank path queries.									
5	14	21	11.31	15.3	7.89	40.3	19.20	8,136.5	1,653.5
6	17	25	11.27	16.2	7.86	47.0	19.13	7,599.8	1,655.5
7	20	29	11.29	16.8	7.86	54.6	19.15	2,625.1	1,657.5
8	23	33	11.31	18.0	7.85	54.6	19.17	7,241.8	1,659.5
9	26	37	12.08	17.4	10.43	61.2	22.53	616.8	1,660.8
10	29	41	11.42	18.4	10.82	67.0	22.26	508.7	1,662.9
11	32	45	11.36	18.1	10.89	63.8	22.27	437.8	1,666.6
12	35	49	11.36	18.6	10.79	64.2	22.17	2.2	1,668.0
13	38	53	11.38	18.6	10.79	62.1	22.19	347.6	1,671.8
14	41	57	11.35	18.4	10.86	64.8	22.23	2.1	1,672.6
15	44	61	11.38	19.0	10.83	62.2	22.23	7.2	1,682.6
ACGT-infix queries.									
5	26	41	28.81	56,003.8	13.40	16,673.6	42.24	340,058.0	4,710.7
6	32	50	35.95	88,080.5	14.89	28,392.0	50.85	132,336.0	6,630.9
7	38	59	46.38	122,192.0	21.18	41,385.0	67.58	350,205.0	8,746.6
8	44	68	53.66	137,956.0	24.24	47,955.8	77.92	249,552.0	9,872.3
9	50	77	65.46	171,152.0	24.96	53,707.0	90.44	96,007.6	11,829.8
10	56	86	84.68	221,063.0	29.05	71,632.8	113.74	19,737.7	14,984.6
11	62	95	99.06	248,124.0	27.56	66,273.2	126.64	23,215.7	16,211.2
12	68	104	104.21	244,967.0	28.30	66,801.8	132.53	88,528.8	16,280.0
13	74	113	134.08	290,691.0	31.06	72,878.5	165.16	22,374.0	18,831.4
14	80	122	155.84	324,762.0	31.84	77,746.0	187.70	169,539.0	20,886.2
15	86	131	173.48	346,809.0	33.40	84,285.4	206.89	6,137.9	22,577.9
ACGT-flat queries.									
5	10	13	13.99	63.3	9.51	40.2	23.53	340,058.0	1,643.4
6	12	15	14.15	83.6	9.30	52.1	23.48	132,336.0	1,648.8
7	14	17	12.48	104.4	8.85	63.2	21.36	350,205.0	1,652.5
8	16	19	11.69	118.9	8.77	73.8	20.50	249,552.0	1,654.1
9	18	21	11.73	138.6	8.73	84.4	20.54	96,007.6	1,660.2
10	20	23	11.67	154.3	8.77	94.4	20.47	19,737.7	1,662.6
11	22	25	11.66	168.2	8.58	101.0	20.27	23,215.7	1,667.8
12	24	27	11.64	179.8	8.68	108.4	20.35	88,528.8	1,671.2
13	26	29	19.29	187.5	14.59	113.9	33.96	22,374.0	1,673.1
14	28	31	19.58	193.8	14.53	118.7	34.14	169,539.0	1,677.3
15	30	33	19.51	205.9	14.54	127.0	34.09	6,137.9	1,681.3

Figure 6: Benchmark results. Each row represents the average of 25 random queries of the same size.

queries for each query size, thus the average numbers of nodes selected (9) are – correctly – the same.

Note that in the times reported in Figure 6, the top-down phase was fully executed and all true predicates were computed for each node. The occurrences of the query predicate were counted and reported in column (9), but no XML output was produced *in the experiments reported*. The reason for this is that the output depends very much on the context in which the query engine is used. For example, returning the subtrees of all nodes as XML may lead to a tree that is quadratically larger that the input document, which is infeasible given the sizes of our databases.

As the default behavior of Arb, the entire XML document is returned with selected nodes marked up in the usual XML fashion. This output can be produced in the second (top-down traversal) phase of query processing, and all information necessary for this is computed within the times shown in in Figure 6.

As expected, the lazy computation of our automata caused the processing of both phases to consist of a warm-up phase in which most transitions were computed, followed by a phase in which the query engine had a simple task and was mainly waiting for the disk to read and/or write files.

Our benchmarks show that the tree automata-based approach to processing node-selecting queries on very large trees is extremely competitive. We only need to scan the database linearly twice and write and reread temporary data in the same linear fashion. The main memory requirements are very low. Apart from a stack bounded by the maximum depth of the XML tree, we only need to store the state descriptions and transition functions for the two automata[15], which are both computed lazily.

There are a few pleasant surprises. It was to be expected that the queries on ACGT-flat would perform well, as they are bottom-up, so intuitively, the initial bottom-up phase has to do little guesswork. In the top-down queries on treebank, however, an STA

[15]In total, we use four hash tables to store and quickly access the states and transitions of the two automata. ComputeReachableStates and ComputeTruePreds are only invoked when transitions cannot be found in the hash tables.

would have to do much guess-work. Here it shows how practically important our representations of sets of reachable STA states as single residual programs are: only very few such programs are required, and the automata computed for top-down queries (on Tree-bank) in our benchmarks end up being very small.

As expected, ACGT-infix queries are substantially more complicated. Nevertheless, we can deal with them surprisingly well. It is particularly remarkable that even when hundreds of thousands of transitions have to be computed, main memory consumption is very low. Our residual programs indeed tend to be amazingly small.

7 Conclusions and Future Work

Our experiments demonstrate the immediate practical usefulness of our approach of using tree automata for the evaluation of expressive node-selecting queries on trees in secondary storage, but more experiments are in place and under way.

Beyond this, which was the goal motivated in the introduction, the approach has a number of interesting properties that need to be further studied and possibly exploited in the future.

- Tree automata-based query processing lends itself to parallel query processing. As pointed out, this application is in need of the expressive power of our framework, as trees have to be restructured (balanced) for parallel processing.
- Multiple query evaluation. TMNF programs can evaluate several queries (each one defined by one IDB predicate) in one program. It will be interesting to study how well Arb handles multiple queries.

A further goal is the integration of Arb with index structures. Already now, precomputed information can be made use of through predicates available to our automata as part of the labeling information (each node may have *any set* of input predicates as label). In the future, we plan to work on ways of detecting, given a query, which parts of the data tree can be jumped over and do not have to be processed by our automata.

Acknowledgments

The author is indebted to Wang-Chiew Tan for her XML-ized version of Swissprot. The work was sponsored by Erwin Schrödinger grant J2169 of the Austrian Research Fund (FWF).

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] S. Abiteboul and V. Vianu. "Regular Path Queries with Constraints". In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'97)*, Tucson, AZ USA, 1997.

[3] M. Altinel and M. Franklin. "Efficient Filtering of XML Documents for Selective Dissemination of Information". In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00)*, pages 53–64, Cairo, Egypt, 2000.

[4] A. Brüggemann-Klein, M. Murata, and D. Wood. "Regular Tree and Regular Hedge Languages over Non-ranked Alphabets: Version 1, April 3, 2001". Technical Report HKUST-TCSC-2001-05, Hong Kong University of Science and Technology, Hong Kong SAR, China, 2001.

[5] A. Brüggemann-Klein and D. Wood. "Caterpillars: A Context Specification Technique". *Markup Languages*, **2**(1):81–106, 2000.

[6] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990.

[7] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer-Verlag, 1999. Second edition.

[8] M. Frick, M. Grohe, and C. Koch. "Query Evaluation on Compressed Trees". In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS)*, Ottawa, Canada, June 2003.

[9] G. Gottlob and C. Koch. "Monadic Datalog and the Expressive Power of Web Information Extraction Languages", Nov. 2002. Journal version of PODS'02 paper, submitted. Available as CoRR report arXiv:cs.DB/0211020.

[10] G. Gottlob, C. Koch, and R. Pichler. "Efficient Algorithms for Processing XPath Queries". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, 2002.

[11] G. Gottlob, C. Koch, and R. Pichler. "XPath Query Evaluation: Improving Time and Space Efficiency". In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE'03)*, Bangalore, India, Mar. 2003.

[12] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. "Processing XML Streams with Deterministic Automata". In *Proc. of the 9th International Conference on Database Theory (ICDT'03)*, 2003.

[13] M. Minoux. "LTUR: A Simplified Linear-Time Unit Resolution Algorithm for Horn Formulae and Computer Implementation". *Information Processing Letters*, **29**(1):1–12, 1988.

[14] F. Neven. *"Design and Analysis of Query Languages for Structured Documents – A Formal and Logical Approach".* PhD thesis, Limburgs Universitair Centrum, 1999.

[15] F. Neven. "Automata Theory for XML Researchers". *SIGMOD Record*, **31**(3), Sept. 2002.

[16] F. Neven and T. Schwentick. "Query Automata on Finite Trees". *Theoretical Computer Science*, **275**:633–674, 2002.

[17] F. Neven and J. van den Bussche. "Expressiveness of Structured Document Query Languages Based on Attribute Grammars". *J. ACM*, **49**(1):56–100, Jan. 2002.

[18] W. Thomas. "Automata on Infinite Objects". In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2, chapter 4, pages 133–192. Elsevier Science Publishers B.V., 1990.

[19] World Wide Web Consortium. XML Path Language (XPath) Recommendation. http://www.w3c.org/TR/xpath/, Nov. 1999.