Bounded depth data trees

Henrik Björklund[⋆1] Mikołaj Bojańczyk[2⋆⋆]

[1] University of Dortmund
[2] Warsaw University

Abstract. A data tree is a tree where each node has a label from a finite set, and a data value from a possibly infinite set. We consider data trees whose depth is bounded beforehand. By developing an appropriate automaton model, we show that under this assumption various formalisms, including a two variable first-order logic and a subset of XPath, have decidable emptiness problems.

1 Introduction

A data tree is a tree where each node has a label from a finite set, and a data value from a possibly infinite set. We consider trees where there is a fixed bound on the depth of nodes. For recognizing properties of such trees, we define an automaton model that traverses the trees in a depth-first manner. We show that the emptiness problem for the automata is decidable, by a reduction to reachability for priority multicounter automata, a powerful model for recognizing word languages [32]. The automaton model is used to show decidability of the satisfiability problem for a two-variable first-order logic, and also for a fragment of XPath. In the logic and XPath, we allow a rich vocabulary of navigational predicates, such as document order, thus extending the work from [6], where only successor axes were allowed.

The main application area for this paper is static analysis tasks for XML databases. We would like to develop tools that automatically answer questions such as: does property a of XML documents always imply property b?; or: is property a vacuously true?

A very successful approach to static analysis has been to use tree automata.An XML document is modeled as a tree, where the labels of the tree correspond to tag names in the document. Many formalisms for XML can be represented as tree automata, possibly extended with additional features, (see, e.g., [29, 23]). Using this representation, a large body of techniques for tree automata can be applied to solving static analysis tasks.

A drawback of the tree automaton approach is that it considers only the tag names, and ignores other content stored in the document. For instance, one cannot express key constraints such as: "every two nodes have different values stored in their `unique_key` attribute". Such constraints are clearly important for databases, and can be expressed in, say, XPath. One way of extending the tree automata approach beyond mere tag names is to consider data trees. In a data tree, each node has a label from a finite set, and a data value from a possibly infinite set. The data values are used to model the content of the document. Recently, there has been flurry of research on models with data, including data words [19, 31, 13, 14, 5, 22, 3], and data trees [8, 1, 4, 16, 6].

The typical tool for solving logics over trees without data is a finite-state automaton. When data is added, the appropriate automaton almost always involves counting:

⋆ Supported by the Deutsche Forschungsgemeinschaft Grant SCHW678/3-1 and the DAAD Grant D/05/02223.
⋆⋆ Supported by Polish goverment grant no. N206 008 32/0810.

ranging from automata with semilinear constraints on runs [6], through vector-addition systems [5], and on to faulty counter machines [14] and lossy channel machines [22]. Complexities are often high: non-primitive recursive, e.g. [14] and some results in [22], or as hard as reachability in vector-addition systems [5] (a decidable problem, but not known to be primitive recursive [26, 21]).

Due to the above, logics for objects with data are usually quite weak. For data trees, the present cutting edge is a fragment of first-order logic, where only two variables are used, and only the child and next-sibling axes are allowed for testing spatial relationship [6]. This logic has decidable emptiness, but most extensions are undecidable: adding a third variable, adding a second data value, adding order on data values. One question left open in [6] was whether the logic remains decidable if we add their transitive closures (i.e. descendant and following sibling)? The outlook is not optimistic, since the extended problem subsumes reachability for tree vector-addition systems, a difficult open problem [12].

In this paper, we try to deal with the extended axes. We show that if a fixed bound on the depth is imposed, the logic from [6] remains decidable after the descendant and following sibling axes (and even document order) are added to the signature. (The following sibling axis is more interesting than the descendant axis in bounded depth trees.) In terms of XPath, we extend the fragment from [6], by allowing all navigational axes in Core XPath ([17]) in path expressions, and considerably stronger predicate expressions, where the data values of two relative paths can be compared, as long as the paths belong to the same subtree.

Another motivation to consider bounded depth trees is that the lower bounds in [6] are somewhat artificial, using constructions alien to actual XML documents. Indeed, many XML data bases are wide, but not very deep (see, e.g.,[9]). Therefore, considering trees of arbitrary depth, which turns out to be a major technical difficulty, need not reflect problems in the real world. It is, however, sometimes crucial to compare elements on a horizontal axis (which nodes are later/earlier in the document), something that cannot be done by the logic in [6].

What do we gain by bounding the depth? The main idea is that a bounded depth tree actually bears more resemblance to a word than a tree. If a bounded depth tree is written down as a string (the way XML documents are stored in text files), a finite string automaton can recover the tree structure by using its finite control to keep track of the path leading to the current node. This simple observation is the essence of our approach. However, it is not immediately clear how the string automaton should deal with data values. We discover that the appropriate model is an extension of multicounter automata, where a limited form of zero test is allowed [32].

The paper is structured as follows. In Section 2 we define bounded depth data trees, and some notions for discussing them. Section 3 contains the main contributions of the paper. Here, we present our automaton model, describe its basic properties, and prove that the corresponding emptiness problem is decidable. Section 4 describes a fragment of first-order logic, which, thanks to the automaton model, can be shown to have a decidable satisfiability problem. Section 5 describes applications for XPath.

2 Definitions

To simplify technicalities, we do not actually consider data trees, but data forests. Informally, a data forest is an ordered sequence of data trees. Formally, a *data forest* is a partial function

$$t : \mathbb{N}^+ \to \Sigma \times \Delta$$

with nonempty finite domain. The set Σ is called the *alphabet* and is required to be finite, while the set Δ is called the *data domain*, and may be infinite (usually, we use the natural numbers for Δ). The *nodes* of the forest are elements of the domain of t. The first coordinate of $t(x)$ is called *the label* of the node x, while the second coordinate of $t(x)$ is called the *data value* of x. Furthermore, the set of nodes must be closed under parents and previous siblings:

– The *parent* of a node $a_1 \cdots a_n a_{n+1} \in \mathbb{N}^*$ is the node $a_1 \cdots a_n$.
– The *previous sibling* of a node $a_1 \cdots a_n \in \mathbb{N}^*$ is the node $a_1 \cdots a_{n-1}(a_n - 1)$. (A node with $a_n = 0$ has no previous sibling.)

Preceding siblings are defined by taking the transitive closure the previous sibling. The opposite of previous/preceding siblings are next/following siblings. A node has at most one previous/next sibling, but possibly many preceding/following siblings. A *root* in a forest is any node without a parent; there may be many roots. The *depth* of a node $a_1 \cdots a_n$ is the number n; in particular each root has depth 1. The opposite of parent is *child.* The transitive closure of the child relation is the *descendant* relation, similarly *ancestors* are defined for parents. A *leaf* is a node without children.

A *depth k data forest* is one where all leaves have depth k. We could also consider forests where leaves have depth at most k; however the more general type can be easily encoded in the special one by adding dummy nodes. When considering depth k data forests, we assume without loss of generality that the label set Σ is partitioned into k disjoint sets $\Sigma_1, \ldots, \Sigma_k$ such that nodes of depth i are only allowed to use labels from Σ_i. This assumption can be easily ensured by expanding the alphabet.

A *class* of a data forest is a maximal set of nodes with the same data value.

Let t be a forest. The *depth-first-search traversal* (DFS traversal) of t is a sequence $v_1, \ldots, v_n$ of nodes of t satisfying:

– Each non-leaf node appears twice, and each leaf appears once.
– If $i < n$ and v_i appears for the first time, i.e. $v_i \notin \{v_1, \ldots, v_{i-1}\}$, then v_{i+1} is the leftmost child of v_i, except if v_i is a leaf, in which case v_{i+1} is the next sibling of v_i, or, if v_i is a rightmost child, the parent of v_i.
– If $i < n$ and v_i is seen for the second time, i.e. $v_i \in \{v_1, \ldots, v_{i-1}\}$, then v_{i+1} is the next sibling of v_i, or the parent of v_i if v_i is a rightmost child.

There is only one DFS traversal, and it must begin with the leftmost root and end with the rightmost root. Later on, it will be convenient that non-leaf nodes are visited twice. If we remove repetitions from the DFS traversal (by deleting second occurrences), we get the *document ordering* on nodes of a forest.

3 Automata

This section contains the main contribution of the paper. In Section 3.1, we define an automaton model for bounded depth data forests. After showing some properties that can be recognized by our automata in Section 3.2, we show in Section 3.3 that the automata have decidable emptiness. The decidability proof is by reduction to reachability in an extended model of multicounter automata (Petri nets). Therefore, we have no primitive recursive upper bound for the complexity; lower bounds are also open.

The automaton model we define can be seen as an extension of the class memory automata for words from [3] to bounded depth forests. These are, in turn, a variant of the data automata from [5]. The basic idea is to use one class memory automaton per depth level in the forest.

3.1 Class memory automata for forests of bounded depth

A depth k forest class memory automaton (k-FCMA) is defined as follows. It has $k+1$ state spaces: $Q, Q_1, \cdots, Q_k$. Each has an initial and a final subset:

$$I, F \subseteq Q \quad I_1, F_1 \subseteq Q_1, \quad \cdots \quad , I_k, F_k \subseteq Q_k \ .$$

The idea is that the states Q_i will be used to examine data values of nodes at depth at least i, while the states in Q are used to examine properties that do not involve data.

The automaton runs on an input depth k forest by visiting its nodes in the DFS sequence (in particular, non-leaf nodes are visited twice). At every moment of its run, it keeps its current state $q \in Q$ – called the *global state* – as well as k *class memory functions* of the form

$$f_1 : \Delta \to Q_1 \qquad \cdots \qquad f_k : \Delta \to Q_k \ .$$

Therefore, a configuration of the automaton consists of: a node v of the forest t, the global state q and the class memory functions $f_1, \ldots, f_k$. (Thanks to the class memory functions, the automaton is a type of infinite-state system, which contributes to the high complexity of emptiness. Each configuration can be finitely represented, since the class memory functions have finite non-initial support.) At the beginning of the run, v is the leftmost root, q is set to be a designated *initial state* $q_I \in Q$, while all the class memory functions $f_1, \ldots, f_k$ assign initial states to all data values $d \in \Delta$. (If there are many initial states, this produces nondeterminism.)

A single step of the automaton works as follows. Assume that the automaton is in a node v of depth i with data value d. Depending on the global state, the values of $f_1(d), \ldots, f_i(d)$, and the label of v, the automaton picks a new global state and new values of $f_1(d), \ldots, f_i(d)$. It then advances to the next node in the DFS traversal. Therefore, the transition function is a set of rules from

$$\bigcup_{i=1,\ldots,k} \quad Q \times Q_1 \times \cdots \times Q_i \times \Sigma_i \times Q \times Q_1 \times \cdots \times Q_i$$

Note that since Σ is partitioned into sets $\Sigma_1, \ldots, \Sigma_k$, the label of a node determines its depth. In particular, the automaton knows if it is descending into a successor, moving to the right sibling, or ascending into the parent.

Furthermore, when the automaton has just read for the second time a rightmost sibling v at depth $i \in \{1, \ldots, k\}$ (or for the first time, if v is a leaf), it does some further processing on the class memory function f_i, which we call a *check-reset.* (The check-reset is done after the transition corresponding to the second visit in v has been applied.) First, the automaton checks if the class memory function f_i is *accepting*, i.e. all data values are assigned either initial or accepting states. If this is not the case, the run is aborted and cannot be continued. If this check succeeds, the class memory function f_i is reset, by assigning the initial state (nondeterministically, if there is more than one) to all data values.

The automaton accepts the data forest if, after completing the DFS traversal, it has an accepting global state (and the last-check reset has been successful). Note however, that before this happens, a large number of memory check-resets must be successfully carried out.

Example 1. Consider the following property of depth k forests: each data value occurs at most once. To recognize this property, the automaton only uses the states Q_1 (all other state spaces Q and $Q_2, \dots, Q_k$ contain one state q, which is both initial and final, and is never modified). There are two states in Q_1: an initial state *new* and a final state *old*. The transition function advances *new* to *old*, while *old* has no outgoing transitions. In other words, there is only one transition for each letter $a \in \Sigma$:

$$(q, new, q, \cdots, q, a, q, old, q \cdots, q).$$

3.2 Some properties of FCMA

In this section we present some properties of bounded depth data forests that can be recognized by FCMA. Apart from being useful later on, the results in this section are meant to give a feeling for what FCMA can do.

Fact. FCMA are closed under union and intersection.

When the depth of a data forest is limited to 1, the forest is a *data word*, as considered in [5]. Furthermore, *data automata*, the automaton model introduced in [5] to recognize properties of data words, coincides with the restriction of FCMA to depth 1. Lemma 1 below can be used to transfer results about data words to data forests.

Lemma 1. *Let $\mathcal{A}$ be a data automaton. The following properties of data forests are recognized by FCMA:*

- *For every node v, the children of v, when listed from left to right, form a data word accepted by $\mathcal{A}$.*
- *For every node v, the descendants of v, when listed in document order, form a data word accepted by $\mathcal{A}$.*

Proof. Since FCMA are closed under intersection, it is sufficient to produce an FCMA that checks the above properties only for nodes v at a given depth i.

We only present the automaton for the second property, the first one is done in a similar way. The FCMA uses the states Q_i to simulate $\mathcal{A}$ on the data word formed by the descendants; the other states are not used. Note that an FCMA knows when it visits a node for the second time (the previous node was at greater depth); therefore the document order can be reproduced from the DFS traversal given to an FCMA. □

Sometimes it is convenient to see how the data value of a node is related to the data values of its neighborhood. The *profile* of a node is information about which nodes among its ancestors, previous and next siblings have the same data value. Once the depth k of forests is fixed, there are finitely many possible profiles. The following lemma shows that these can be tested by an automaton:

Lemma 2. *For each possible profile p, there is an FCMA that recognizes the language: "a node has label a if and only if it has profile p".*

Proof. Thanks to closure under intersection, it is sufficient to present separate automata for the following languages:

– "A node has label a if and only if it has an ancestor at depth i with the same data value." Here i is a fixed depth in $\{1, \ldots, k\}$. The automaton only uses the states Q_i, which can have two values: "ancestor" and "not ancestor". The invariant is going to be: when the automaton is in a node v of depth greater than i, the class memory function f_i assigns "ancestor" to the unique data value in the ancestor of v at depth i. All other data values are assigned "not ancestor". Using this information, the language from the statement of the lemma can easily be recognized. Furthermore, it is easy to define the automaton so that the invariant is satisfied. When a node at depth i with data value d is entered for the first time, $f_i(d)$ is updated to "ancestor". When the node is visited for the second time, $f_i(d)$ is updated to "not ancestor". Note that it is important that each inner node is visited twice.
– "A node has label a if and only if its previous sibling (next sibling) has the same data value." In [3], the authors show that there is a data automaton that recognizes the following property of data words: "a node has label a if and only if its previous sibling has the same data value." This gives the desired result, thanks to Lemma 1. □

We will also need to use FCMAs to recognize languages of the form: "for every class, a given property holds". Here we present a general result of this type. Note that it is not clear what we mean when saying that a class satisfies some property, since it is not clear how the nodes of a class should be organized once they are taken out of the data forest. Here we use one such definition, which we call a *take-out.* Let t be a forest and V a set of nodes in this forest. The nodes of the take-out are nodes of V, along with their ancestors. The labels in the take-out are inherited from t, except we add a special marker to distinguish if a node is from V, or just an ancestor of a node from V. The take-out is a forest without data, where leaves may have different depths.

Lemma 3. *Let L be a regular forest language (without data). An FCMA can test if the take-out of every class belongs to L.*

In the proof of this lemma, it will be convenient to use a non-standard automaton model for bounded depth forests without data values, called a *leaf automaton.* Such an automaton works over forests where all leaves are at depth at most k, where k is fixed for the automaton. (Unlike for depth k data forests, we allow leaves at varying depths.) The automaton only visits leaves of the forest, from left to right, i.e. in document order. When moving from a leaf v to the next leaf w, the automaton checks the following information:

– The label of v;
– The depth of v, and the labels of its ancestors;
– The depth of the deepest common ancestor of v and w.

(Note that in a forest, two nodes may not have a common ancestor; in which case the answer to the last question is output as 0.) To read the information in the rightmost leaf, we append a dummy leaf at the end of the forest. The automaton accepts if in this dummy it has a designated accepting state. It is not difficult to see that leaf automata are strong enough to capture all regular forest languages (over forests of bounded depth).

Proof (of Lemma 3). Let k be the depth of the forest. We assume that the regular language L is given by a leaf automaton $\mathcal{A}$. The FCMA will simulate the automaton $\mathcal{A}$. To simplify presentation, we divide the work into six passes. Using nondeterminism and closure under intersection, these can easily be combined into a single pass of an FCMA.

1. In the first pass, we mark each node with the labels of all its ancestors. This is easy, since an FCMA can remember in its global state the labels on the path leading to the current node. We will be using this feature (checking labels of ancestors) in the subsequent passes as well.
2. In this pass, we mark each node with information about which ancestors have the same data value. This can be done thanks to Lemma 2.
3. In this pass, we place a special marker on class leaves; here a *class leaf* is a node that has no descendants in its class. With the information from the previous pass, class leaves can be identified without looking at data values: a node at depth i is class leaf if and only if none of its descendants has an ancestor with the same data value at depth i.
4. In this pass, we mark a node v if it satisfies the following property (*):

   (*) Some following sibling w of v has a descendant (or possibly w itself) in the class of v.

   Note that in the above, w need not be the next sibling of v, it may be further to the right. The automaton for this pass is slightly more involved, and we relegate its description to Lemma 4 below.
5. In this pass, we mark a class leaf v with the depth $i \in \{-1, 0, \dots, k-1\}$ of the greatest common ancestor for v and the next class leaf $w \sim v$ to the right of v. (The meaning of -1 is that v is rightmost in its class and thus no w exists.) This marking can be inferred from the information gathered in the previous pass. If all ancestors of v satisfy property (*), then the answer is $i = -1$. Otherwise, the answer is $i = j - 1$, where j is the smallest depth at which an ancestor of v with property (*) can be found.
6. In the last pass, we use the information gathered above to simulate $\mathcal{A}$ on the class take-outs.

□

Lemma 4. *An FCMA can verify that a node has label a if and only if it satisfies property (*) from the previous lemma.*

Proof. In this lemma, it will be convenient that an FCMA visits each node twice. The automaton we define does not use global states Q, only the states $Q_1, \dots, Q_k$ for the class memory functions $f_1, \dots, f_k$. Each state space Q_i is the same, and contains three states:

p	An initial state.
q	From now until the class-reset, no nodes in this class are allowed.
r	From now until the class-reset, a node from this class must be seen.

The accepting states are $\{p, q\}$. The transitions of the automaton are defined as follows. Let v be a node at depth i, with data value d.

1. When the node v is visited for the first time, the label is not read. The automaton checks if any of the class memories $f_1(d), \dots, f_i(d)$ contains the state q. If yes, the automaton fails and the run is terminated. Otherwise, all the class memories $f_1(d), \dots, f_i(d)$ are set to the initial (and accepting) state p.
2. When the node v is visited for the second time (i.e. after the automaton has visited the subtree of v), and v is marked by a, the automaton sets the class memory function $f_i(d)$ to the non-accepting state r. Therefore, some node in the same class must occur (and the state r will be changed to q, as above) before the check-reset happens; otherwise the check-reset will fail.

3. When the node v is visited for the second time (i.e. after the automaton has visited the subtree of v), and v is not marked by a, the automaton sets the class memory function $f_i(d)$ to the accepting state q. However, we will not be able to read a node in the same class until the check-reset, as described in the first item above.

It should be clear that this automaton describes the language in question. □

3.3 Decidable emptiness for the automata

In this section, we will show that emptiness is decidable for k-FCMA. The proof is by reduction to emptiness of priority multicounter automata. Note that universality is undecidable even for 1-FCMA, as it is already undecidable for data automata over words.

Priority multicounter automata A priority multicounter automaton is an automaton over words (without data) that has a number of counters, which can be incremented, decremented and tested for zero. (Multicounter automata with zero tests correspond to Petri nets with inhibitor arcs.) To keep the model decidable, the zero tests are restricted. This is where the priorities come in.

More formally, a *priority multicounter automaton* has a set C of counters, a state space Q and an input alphabet Σ. Furthermore, the counters come with a distinguished chain of subsets: $C_1 \subseteq \cdots \subseteq C_m \subseteq C$.

The automaton reads a word $w \in \Sigma^*$ from left to right, possibly using ϵ-transitions. At each point in its run, the automaton has a current state $q \in Q$ and a non-negative counter assignment $c \in \mathbb{N}^C$. At the beginning, a designated initial state is used, and all the counters are empty.

In a transition, the automaton reads a letter – possibly ϵ – from the word. Depending on this letter the automaton changes its state, and performes a *counter operations*, that is, it increases a counter, decrements a counter, or checks that all counters in C_i, for some i, are empty.

The above operations can fail: if a decrement is done on an empty counter; or if a zero test fails. When the counter operation fails, the transition fails and the run is aborted. The automaton accepts if at the end of the word it has reached a designated accepting state. The following difficult result has been shown in [32]:

Theorem 1. *Emptiness is decidable for priority multicounter automata.*

Note that priority multicounter automata are an extension of multicounter automata (where the zero tests are not allowed). In particular, no primitive recursive emptiness algorithm is known.

Reduction to priority multicounter automata We now show that emptiness for FCMA can be reduced to emptiness of priority multicounter automata. In particular, thanks to Theorem 1, emptiness is decidable for FCMA.

Let t be a depth k forest, and let $v_1, \ldots, v_n$ be its DFS traversal. Let $\mathrm{trav}(t)$ be the word over Σ containing the labels of $v_1, \ldots, v_n$. Since $\mathrm{trav}(t)$ does not use the data values, it is irrelevant if t is a data forest or a non-data forest.

Theorem 2. *Emptiness is decidable for k-FCMA, for all $k \in \mathbb{N}$. Furthermore, for each k-FCMA $\mathcal{A}$, the set $\{\mathrm{trav}(t) : t$ is accepted by $\mathcal{A}\}$ is accepted by an (effectively obtained) priority multicounter automaton.*

By Theorem 1, the first clause of the theorem follows from the second one. This section is therefore devoted to simulating a k-FCMA with a priority multicounter automaton.

We fix a k-FCMA $\mathcal{A}$. We assume that in every transition

$$(q, q_1, \ldots, q_i, a, r, r_1, \ldots, r_i)\ ,$$

none of the states $r_1, \ldots, r_i$ are initial; and if some q_j is initial, then so are $q_{j+1}, \ldots, q_i$. Any k-FCMA can be effectively transformed into one satisfying the above assumptions.

The priority multicounter automaton that recognizes the traversals is defined as follows. It is a conjunction of two automata. The first one checks that the depths indicated by the labels are consistent with a DFS traversal, i.e. the input word belongs to $\{\text{trav}(t) : t \text{ is a depth } k \text{ forest}\}$. Since the latter is a regular word language, we do not even need to use counters.

The real work is done by the second automaton, which we call $\mathcal{B}$. To simplify presentation, we use a slightly extended notion of transition. We will later comment on how the extended notion can be realized by a standard priority multicounter automaton. The control states of $\mathcal{B}$ are the global states Q of $\mathcal{A}$. It has a counter for each of the states in $Q_1, \ldots, Q_k$ used in the class memory functions (we assume these state spaces are disjoint).

When the simulating automaton $\mathcal{B}$ is in state $q \in Q$, and the input letter is $a \in \Sigma_i$ (with $i = 0, \ldots, k$) the automaton performs the following actions:

1. As preprocessing for the transition, $\mathcal{B}$ may nondeterministically choose to increment any counter corresponding to an initial state.
2. In the next step, $\mathcal{B}$ nondeterministically picks a transition

   $$(q, q_1, \ldots, q_i, a, r, r_1, \ldots, r_i)$$

   of the simulated k-FCMA $\mathcal{A}$. It decrements counters $q_1, \ldots, q_i$, and then increments the counters $r_1, \ldots, r_i$.
3. In the third step, $\mathcal{B}$ sets its finite control to the the state r from the transition chosen in step 2.
4. The last step corresponds to the check-reset and is carried out if the next label is going to be from Σ_{i-1} (this corresponds to a rightmost successor node appearing for the second time in the DFS, or for the first time, if the node is a leaf). In this case, the automaton $\mathcal{B}$ tests that all counters in

   $$Q_i \setminus (F_i \cup I_i) \tag{1}$$

   are empty, and then empties all the counters in Q_i.

We call such a sequence of actions a *macrotransition.* A macrotransition can be carried out by a multicounter automaton with zero checks, by using ϵ-transitions and additional control states. Perhaps the most delicate point is the last step in the macrotransition. First of all, the automaton needs to know the next label. Here, we can nondeterministically guess the next label in advance; this nondeterministic guess is then validated in the next step. (The degenerate case of $j = 0$ is handled by using ϵ-transitions.)

At first glance, the automaton is not a priority multicounter automaton, since the zero checks in (1) are done for disjoint counters. But this can easily be fixed, by imposing a chain discipline on the zero checks. Indeed, when the automaton is doing the zero check in (1), we know that in the previous moves it has emptied the counters $Q_{i+1}, \dots, Q_k$. Therefore, it could equivalently zero check the counters

$$Q_i \setminus (F_i \cup I_i) \quad \cup \quad Q_{i+1} \quad \cup \quad \cdots \quad \cup \quad Q_k \; .$$

Furthermore, the emptying of the counters in Q_i, which is done after (1), can be simulated by a sequence of nondeterministic decrements on Q_i and then a zero check on $Q_i \cup \cdots \cup Q_k$. The automaton $\mathcal{B}$ accepts if it reaches an accepting global state after processing all the nodes. It is fairly clear that if $\mathcal{A}$ accepts t, then $\mathcal{B}$ accepts $\mathrm{trav}(t)$. Theorem 2 then follows once we show the converse:

Lemma 5. *If t is a depth k forest whose DFS traversal is accepted by $\mathcal{B}$, then t can be labeled with data values so that the resulting data forest is accepted by $\mathcal{A}$.*

Proof. Consider an accepting run of $\mathcal{B}$, with macrotransitions $m_1, \dots, m_n$. Let $v_1, \dots, v_n$ be the DFS traversal of the forest t. These nodes correspond to the macrotransitions $m_1, \dots, m_n$. Recall that each macrotransition corresponds (in step 2) to a transition of the automaton $\mathcal{A}$. Let then $\delta_1, \dots, \delta_n$ be the sequence of transitions of $\mathcal{A}$ that corresponds to $m_1, \dots, m_n$.

We will assign data values to nodes of the forest t, so that the result s is accepted by $\mathcal{A}$, using the run $\delta_1, \dots, \delta_n$. This is done progressively for $v_1, \dots, v_n$, so that at each intermediate step $j = 0, \dots, n$ the following invariant is satisfied.

Assume that data values have been assigned to nodes $v_1, \dots, v_j$. The sequence $\delta_1, \dots, \delta_j$ is a partial run of $\mathcal{A}$ on t (that has read nodes $v_1, \dots, v_j$) such that:

> For each class memory function $f_i \in \{f_1, \dots, f_k\}$, and each non-initial state $q \in Q_i$, the counter q contains the number of data values d with $f_i(d) = q$.

This invariant can be easily shown by induction on j. □

4 A two-variable logic for bounded depth data forests

In this section, we define a first-order logic that can express properties of data forests. Formulas of this logic can be effectively compiled into FCMA; in particular this logic has decidable satisfiability thanks to Theorem 2. Variables quantify over nodes. Only two variables, x and y, are allowed. Furthermore, data values can only be compared for equality, via a predicate $x \sim y$. On the other hand, we allow a large body of navigational predicates.

For a fixed depth k, we define the logic FO^2_k as the two-variable fragment of FO, with the following predicates (some parameterized by $i = 1, \dots, k$):

$d_i(x)$	x has depth i
$a(x)$	x has label a (here a is a label from Σ)
$x \downarrow_i y$	y is a descendant of x and $\mathit{depth}(y) - \mathit{depth}(x) = i$
$x \downarrow_+ y$	y is a descendant of x
$x + 1 = y$	x is the left sibling of y
$x \prec y$	x comes before y in the document ordering (the ordering produced by a pre-order traversal)
$x < y$	x and y are siblings, and x is to the left of y
$t_i(x, y)$	$x \preceq y$ and the nodes x, y share the same depth i ancestor but not the same depth $i + 1$ ancestor

$t_0(x, y)$	x, y do not have a common ancestor
$x \sim y$	x and y have the same data value
$x \oplus y$	y is the *class successor* of x, that is, $x \sim y$, x comes before y in the document ordering, and there is no z between x and y (in the document ordering) which has the same data value.

The semantic of the logic is defined as usual. For instance, the following is a long way of saying that all nodes have the same data value:

$$\forall x \forall y (x \downarrow_+ y \Rightarrow x \sim y) \wedge (x + 1 = y \Rightarrow x \sim y) \ .$$

The predicates d_i, $\downarrow_+, \downarrow_i$ and $<$ are syntactic sugar, and can be removed from the signature without loss of expressivity. For instance, $d_i(x)$ is the same as $t_i(x, x)$. In similar ways, $\downarrow_i$, $\downarrow_+$ can be defined in terms of t_i, and $<$ can be defined in terms of t_i and $\prec$. Since we only have two variables, $x + 1 = y$ cannot be defined in terms of $<$, and $x \oplus y$ cannot be defined in terms of $\prec$ and $\sim$.

In the following example, we show that thanks to the bounded depth assumption, two-variable formulas can express properties that seemingly require three variables.

Example 2. We write a formula $\varphi(x)$, which holds in a node x that has two distinct descendants $y \sim z$. The natural formula would be

$$\varphi(x) \quad = \quad \exists y \exists z \ (y \neq z \ \wedge \ y \sim z \ \wedge \ x \downarrow_+ y \ \wedge \ x \downarrow_+ z) \ .$$

The problem is that this formula uses three variables. We will show that for depth k forests, φ can be written with only two variables. The idea is to do a disjunction over the finitely many possible depths i of the node x:

$$\varphi(x) = \bigvee_i d_i(x) \wedge \exists y (x \downarrow_+ y \wedge \exists x (x \neq y \wedge x \sim y \wedge \bigvee_{j \geq i} t_j(x, y))) \ .$$

In the above formula, the second existential quantifier $\exists x$ actually corresponds to the node z. We do not need to verify if the new node x is a descendant of the "real" node x in the free variable; this is a consequence of $t_i(x, y)$.

Theorem 3. *Every language definable in FO^2_k can be recognized by a k-FCMA.*

Proof. To minimize repetition from [5] and [6], we try to give an informal overview of the translation, and only give details where something new is happening.

Any formula in FO^2_k can be effectively rewritten into Scott normal form with additional unary second-order quantification at the beginning. That is, every FO^2_k formula φ is equivalent to one of the form

$$\exists R_1 \ldots \exists R_m (\forall x \forall y \mathcal{X} \wedge \bigwedge_i \forall x \exists y \mathcal{X}_i),$$

where $R_1, \ldots, R_m$ are new unary predicates, and the formulas $\mathcal{X}, \mathcal{X}_i$ are all quantifier free. The transformation into Scott normal form is quite simple. The general idea is that the predicates $R_1, \ldots, R_m$ correspond to subformulas of the original formula φ, while the conjuncts of the first-order part verify that the marking is consistent.

In our translation, we will consider formulas in Scott normal form. Since FCMA are nondeterministic, they can easily simulate the existential quantification $\exists R_1, \ldots, R_m$.

Since FCMA are closed under intersection, we attack each conjunct separately. Therefore, it remains to show that a k-FCMA can recognize any property defined by a formula of either type below:

$$\forall x \forall y \mathcal{X} \qquad \forall x \exists y \mathcal{X} \qquad \mathcal{X} \text{ is quantifier-free .} \tag{2}$$

We will show that there are essentially finitely many formulas of the above type; and each of these types can be recognized by an automaton.

In the sequel, we will assume that the signature does not contain the relations $\downarrow_i$; but it does contain $\prec$, $\downarrow_+$ and d_i. (Although the latter two are syntactic sugar, they will be convenient in the proofs.) Two nodes, x and y, of a data forest can only be related in a finite number of ways by the signature of FO^2_k. Each of the two nodes can have a *unary type* (a conjunction of unary predicates and their negations). The relationship between the nodes can be described by a *data sensitive binary type* (a conjunction of the predicates $\sim$ and $\oplus$ and their negations) and a *data blind binary type* (a conjunction of the other binary predicates and their negations). We will use α, β to denote unary types, Γ for data sensitive binary types, and Λ for data blind binary types.

There are four data blind binary predicates, one of which is symmetrical. Thus in principle we have 2^7 data blind binary types.[3] Most of these types are inconsistent, e.g., $x \downarrow_+ y \ \wedge \ x + 1 = y$, so the actual number of possible combinations is much smaller. There are four data sensitive binary types: either $x \not\sim y$, or $x \sim y$; in the latter case at most one of $x \oplus y$, $y \oplus x$ holds.

The important thing is that we have a finite number of *binary types* (a conjunction of one data sensitive and one data blind binary type). We will classify the formulas from (2) by looking at the possible binary type relating x with y. We consider the cases of $\forall x \forall y$ and $\forall x \exists y$ in two separate sections.

Formulas of the form $\forall x \forall y \mathcal{X}$. Any such formula can be rewritten as a finite conjunction of formulas of the form

$$\forall x \forall y (\Gamma(x,y) \wedge \Lambda(x,y) \rightarrow \phi(x,y)),$$

where ϕ is quantifier free and uses only unary predicates. We further rewrite this formula as

$$\begin{aligned} \forall x \forall y \ (\ & \Gamma(x,y) \wedge \Lambda_0(x,y) \wedge t_0(x,y) \rightarrow \phi_0(x,y) \\ & \vdots \\ & \wedge \Gamma(x,y) \wedge \Lambda_k(x,y) \wedge t_k(x,y) \rightarrow \phi_k(x,y)). \end{aligned}$$

We move the quantifiers into the conjunction and are left with formulas

$$\forall x \forall y (\Gamma(x,y) \wedge \Lambda_i(x,y) \wedge t_i(x,y) \rightarrow \phi_i(x,y)) \ ,$$

where Λ_i does not use any t_j-predicates.

If Γ contains $x \sim y$, $x \oplus y$, or $y \oplus x$ as a conjunct, such formulas express regular properties of the take-out of each class. The only problem is that two siblings in the take-out might have lost intermediate siblings, so that the truth value of $x + 1 = y$ has changed. This is no real problem since, by Lemma 2, we can assume that each node is marked with information about whether its next sibling belongs to the same class. Thus, by Lemma 3, there are FCMAs that can check the formulas.

[3] To be precise, we actually have more binary predicates, since we have t_i for each i, but since the t_i predicates are mutually exclusive, this is not important.

We are left with the case where Γ is $x \not\sim y$. We separately consider each possible combination of depths for the nodes x, y. That is, we need to present an automaton for the following type of formula:

$$\varphi = \forall x \forall y (x \not\sim y \wedge t_i(x,y) \wedge d_j(x) \wedge d_l(y) \rightarrow \psi(x,y)) \ .$$

If both j, l are $i+1$, then the above formula speaks about sets of siblings. Each such property can be tested by a data automaton (see [5]). Thus, by Lemma 1, there is an FCMA for checking the property.

In the remaining cases, neither $y = x+1$ nor $y = x+1$ can hold, so we may assume that $\psi(x,y)$ only uses unary relations. We transform ψ into CNF. By using standard logic laws, and closure of FCMA under conjunction, we only need consider the case when ψ has only one CNF conjunct; i.e. ψ is a disjunction of unary relations or their negations, some for x, others for y:

$$\alpha_1(x) \vee \cdots \vee \alpha_n(x) \vee \beta_1(y) \vee \cdots \vee \beta_m(y) \ .$$

We write $A(x)$ for $\alpha_1(x) \vee \cdots \vee \alpha_n(x)$; likewise for $B(y)$. Two cases need be considered: when one of the depths j, l is i; and otherwise.

Consider first the case when $j = i$; the case for $l = i$ being symmetric. The formula φ says if x is a node at depth i that does not satisfy A, then all of its descendants $y \not\sim x$ at depth j must satisfy B. This formula can be checked by an FCMA in the following way. For each node x at depth i, the automaton checks if x satisfies A. If this is not the case, it uses the states in Q_{i+1} to check that all descendants y at depth l that belong to another class satisfy B (whether or not a descendant y is in the same class as x can be read from the profile of y, see Lemma 2).

Finally, we consider the case when neither j, l are i. We assume without loss of generality that $i = 0$; the other cases follow thanks to the following:

Observation. Let $\mathcal{A}$ be a k-FCMA. An $(i+k)$-FCMA can check the following property: for every node x at depth i, the forest formed by the children of x is accepted by $\mathcal{A}$.

We put the requirement on $d_j(x)$ into A by adding a disjunct $\neg d_j(x)$; likewise for $d_l(y)$ and B. The formula φ now boils down to checking:

> Let $t_1, \ldots, t_m$ be the trees in the data forest. If for some $j = 1, \ldots, m$, the tree t_j contains a node x that violates A, then all nodes $y \not\sim x$ in the trees $t_{i+1}, \ldots, t_m$ satisfy B.

We say a node is *prime* if it has the same data value as the first node in document order that violates A. One can easily check that φ is equivalent to the following property:

> Let $j = 1, \ldots, m$. If the tree t_j contains a prime node that violates A, then all non-prime nodes in the trees $t_{j+1}, \ldots, t_j$ satisfy B. If t_j contains a non-prime node that violates A, then all nodes in the trees $t_{j+1}, \ldots, t_m$ satisfy B.

The above property can be checked by an FCMA. The general idea is that the automaton uses the global state as well as the class memory function f_1 to keep track of which nodes are prime, and which are not.

Formulas of the type $\forall x \exists y \mathcal{X}$. Following [5], we rewrite them as a disjunction

$$\forall x \exists y \bigvee_i (\alpha_i(x) \rightarrow \beta_i(y) \wedge \Gamma_i(x,y) \wedge \Lambda_i(x,y)) \ .$$

To get rid of the disjunction, we introduce a new unary predicate R_i for each disjunct, and rewrite $\forall x \exists y \mathcal{X}$ as

$$\exists R_1 \exists R_2 \cdots \bigwedge_i \forall x \exists y (\alpha_i(x) \wedge R_i(x) \rightarrow \beta_i(y) \wedge \Gamma_i(x,y) \wedge \Lambda_i(x,y)).$$

Since R_i is unary, it can be incorporated into the unary type of x, and we thus only need to consider formulas of the form

$$\forall x \exists y (\alpha(x) \rightarrow \beta(y) \wedge \Gamma(x,y) \wedge \Lambda(x,y)).$$

Again, if $\Gamma(x,y)$ contains $x \sim y$, $x \oplus y$, or $y \oplus x$, the formula is a regular property of the take-out of each class, and can be checked by an FCMA by Lemma 3. Thus we only worry about the case where $\Gamma(x,y)$ is $x \not\sim y$. We proceed similarly as in the $\forall x \forall y$ case. We only show how to do the following special case, to highlight the differences with $\forall x \forall y$:

$$\varphi = \forall x \exists y (\alpha(x) \rightarrow \beta(y) \wedge x \not\sim y \wedge t_0(x,y)).$$

The formula φ now boils down to checking:

> Let $t_1, \ldots, t_m$ be the trees in the data forest. If for some $j = 1, \ldots, m$, the tree t_j contains a node x that satisfies α, then one of the trees $t_{j+1}, \ldots, t_m$ must contain a node $y \not\sim x$ that satisfies β.

As in the $\forall x \forall y$ case, we distinguish one class. We say a node is *ultimate* if it has the same data value as the last node in document order that satisfies β. One can easily check that φ is equivalent to the following property:

> Let $j = 1, \ldots, m$. If the tree t_j contains an ultimate node that satisfies α, then one of the trees $t_{j+1}, \ldots, t_m$ contains a non-ultimate node that satisfies β. If t_j contains a non-ultimate node that satisfies α, then one of the trees $t_{j+1}, \ldots, t_m$ contains an ultimate node that satisfies β.

$\square$

Corollary 1. *Satisfiability is decidable for FO^2_k.*

5 XPath

We now apply our results to show decidability of some static analysis tasks for XML. Our approach closely mirrors that in [6]. To avoid repetition, we only explain which expressive power can be *added* to the fragment *LocalDataXPath* from [6], while preserving decidability over bounded depth trees. Since we have the predicates $\downarrow_+$ and $<$ in our logic, we can, unlike [6], capture the XPath axes `descendant` (`ancestor`) and `following` (`preceding`). Also, we can allow attribute comparisons in which *both* sides of the (in-)equality are *relative*, as long as they stay *within the subtree* rooted at the node to which they are relative.

As in [6], decidability is shown by encoding XPath into two-variable logic. We first give an example that illustrates how the bounded depth can be used to encode XPath expressions that could not be handled in [6]. This example is similar to Example 2 in the Section 4.

Example 3. Consider the XPath expression

$$\texttt{child} :: a/\texttt{child} :: b/@B_1 = \texttt{child} :: c/\texttt{next} - \texttt{sibling} :: d/@B_2.$$

It is not allowed in LocalDataXPath, since both sides of the equality are relative paths. For bounded depth trees we can, however, define the corresponding first-order predicate as

$$\begin{aligned}\textstyle\bigvee_i (d_i(x)\wedge\ & \exists y\ x \downarrow_1 y \wedge a(y)\wedge \\ & \exists x\ y \downarrow_1 x \wedge b(x)\wedge \\ & \exists y\ x \downarrow_1 y \wedge B_1(y)\wedge \\ & \exists x\ x \sim y \wedge t_i(x,y) \wedge B_2(x)\wedge \\ & \exists y\ y \downarrow_1 x \wedge d(y)\wedge \\ & \exists x\ x+1 = y \wedge c(x) \wedge d_{i+1}(x))\end{aligned}$$

We now define *BDXPath* (Bounded Depth XPath). It is defined the same way as LocalDataXPath from [6]; except that BDXPath can use all the navigational axes in Core XPath [17]:

$$\begin{aligned}\texttt{Axis} := {} & \texttt{ancestor} \mid \texttt{ancestor} - \texttt{or} - \texttt{self} \mid \texttt{attribute} \mid \texttt{child} \mid \texttt{descendant} \mid \\ & \texttt{descendant} - \texttt{or} - \texttt{self} \mid \texttt{following} \mid \texttt{following} - \texttt{sibling} \mid \\ & \texttt{parent} \mid \texttt{preceding} \mid \texttt{preceding} - \texttt{sibling} \mid \texttt{self}\ .\end{aligned}$$

(The non-standard `elsewhere` axis from LocalDataXPath can be simulated by using `preceding` and `following`.)

Also, in predicate expressions, we allow comparisons of attribute values with $=$ and $\neq$ as long as one of the following holds.

1. At least one side of the (in-)equality is an *absolute* location path (i.e., one starting at the root, or document node); or
2. The comparison is relative, but *safe* (as defined in [6]); or
3. Both sides have location expressions that start with `child` or `descendant` and do not use `parent` or `ancestor`.

Using the same proof, but aided by our more powerful two-variable logic, we can upgrade the main XPath result from [6]:

Theorem 4. *Over trees of bounded depth, Satisfiability and Containment for (unary or binary) BDXPath is decidable. This holds even relative to a schema consisting of a regular tree language and unary key and inclusion constraints.*

Acknowledgements. We thank Wim Martens and Thomas Schwentick for valuable discussions.

References

1. N. Alon, T. Milo, F. Neven, D. Suciu and V. Vianu. XML with Data Values: Typechecking Revisited. In *JCSS*, 66(4): 688-727 (2003).
2. M. Arenas, W. Fan and L. Libkin. Consistency of XML specifications. In *Inconsistency Tolerance*, LNCS 3300, 2005, pp. 15-41.
3. H. Björklund, T. Schwentick. On notions of regularity for data languages Manuscript, 2006, available at http://lrb.cs.uni-dortmund.de/~bjork/papers/regular-data.pdf

4. M. Benedikt, W. Fan, and F. Geerts. XPath Satisfiability in the Presence of DTDs. In *PODS'05*, 2005.
5. M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on words with data. In *LICS'06*, pp. 7-16, 2006.
6. M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-Variable Logic on Data Trees and XML Reasoning. In *PODS'06*, 2006.
7. P. Bouyer, A. Petit and D. Thérien. An algebraic approach to data languages and timed languages. *Inf. Comput.*, 182(2): 137-162 (2003).
8. P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, W. C. Tan. Reasoning about keys for XML. In *Inf. Syst.*, 28(8): 1037-1063 (2003).
9. B. Choi. What are real DTDs like. In *WebDB'02*, pp. 43-48, 2002.
10. J. Cristau, C. Löding, W. Thomas. Deterministic Automata on Unranked Trees. In *Fundamentals of Computation Theory (FCT'05)*, LNCS 3623, 2005, pp. 68-79.
11. C. David. Mots et données infinis. Master thesis, Université Paris 7, LIAFA, 2004.
12. P. de Groote, B. Guillaume, and S. Salvati. Vector Addition Tree Automata. In *LICS'04*, pp. 64-73, 2004.
13. S. Demri, R. Lazic, D. Nowak. On the Freeze Quantifier in Constraint LTL: Decidability and Complexity. In *TIME'05*, 2005.
14. S. Demri and R. Lazic. LTL with the Freeze Quantifier and Register Automata. In *LICS'06*, pp. 17-26, 2006.
15. K. Etessami, M.Y. Vardi, and Th. Wilke. First-Order Logic with Two Variables and Unary Temporal Logic. *Inf. Comput.*, 179(2): 279-295 (2002).
16. F. Geerts and W. Fan. Satisfiability of XPath Queries with Sibling Axes. In *DBPL'05*, 2005.
17. G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *VLDB*, 2002.
18. E. Grädel and M. Otto. On Logics with Two Variables. *TCS*, 224:73-113 (1999).
19. M. Kaminski and N. Francez. Finite memory automata. *TCS*, 134:329-363 (1994).
20. E. Kieroński and M. Otto. Small Substructures and Decidability Issues for First-Order Logic with Two Variables. In *LICS'05*, 2005.
21. S.R. Kosaraju. Decidability of reachability in vector addition systems. In *STOC'82*, pp. 267-281, 1982.
22. R. Lazić. Safely Freezing LTL. Foundations of Software Technology and Theoretical Computer Science (FSTTCS), 2006.
23. W. Martens. *Static analysis of XML transformation and schema languages.* PhD Thesis, Hasselt University, 2006.
24. W. Martens, J. Niehren. Minimizing Tree Automata for Unranked Trees. In *10th International Symposium on Database Programming Languages*, LNCS 3774, 2005.
25. M. Marx. First order paths in ordered trees. In *ICDT'05*, 2005.
26. E. Mayr. An algorithm for the general Petri net reachability problem. In *STOC'81*, pp. 238-246, 1981.
27. M. Mortimer. On languages with two variables. *Zeitschr. f. math. Logik u. Grundlagen d. Math.*, 21: 135-140 (1975).
28. K. Neeraj Verma, H. Seidl, T. Schwentick. On the Complexity of Equational Horn Clauses. In *CADE'05*, 2005.
29. F. Neven. Automata, Logic, and XML. In *CSL'02*, pp. 2-26, 2002.
30. F. Neven and T. Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In *ICDT'03*, 2003.
31. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 15(3): 403-435 (2004).
32. K. Reinhardt. Counting as Method, Model and Task in Theoretical Computer Science. Habilitation-thesis, 2005
33. XML Path Language (XPath), W3C Recommendation 16 November 1999. Available at http://www.w3.org/TR/xpath.