Satisfiability of Downward XPath with Data Equality Tests

Diego Figueira
INRIA Saclay, LSV
ENS Cachan, CNRS, France

ABSTRACT

In this work we investigate the satisfiability problem for the logic $\mathsf{XPath}(\downarrow^*, \downarrow, =)$, that includes all downward axes as well as equality and inequality tests. We address this problem in the absence of DTDs and the sibling axis. We prove that this fragment is decidable, and we nail down its complexity, showing the problem to be ExpTime-complete. The result also holds when path expressions allow closure under the Kleene star operator. To obtain these results, we introduce a new automaton model over data trees that captures $\mathsf{XPath}(\downarrow^*, \downarrow, =)$ and has an ExpTime emptiness problem. Furthermore, we give the exact complexity of several downward-looking fragments.

Categories and Subject Descriptors

H.2.3 [**Languages**]: Query Languages; I.7.2 [**Document Preparation**]: Markup Languages

General Terms

Algorithms, Languages

Keywords

XML, XPath, unranked unordered tree, data-tree, infinite alphabet, data values, BIP automaton

1. INTRODUCTION

XPath is arguably the most widely used XML query language. It is implemented in XSLT and XQuery and it is used as a constituent part of several specification and update languages. XPath is fundamentally a general purpose language for addressing, searching, and matching pieces of an XML document. It is an open standard and constitutes a World Wide Web Consortium (W3C) Recommendation [4], implemented in most languages and XML packages.

The most important static analysis problem of a query language is that of optimization, which studies the problems of query containment and query equivalence. In logics closed under boolean operators, these problems reduce to *satisfiability* checking: does a given query express some property? I.e., is there a document where this query has a non-empty result? By answering this question we can decide at compile time whether the query contains a contradiction, and thus whether the computation of the query on the document can be avoided, or if one query can be safely replaced by another one. Moreover, this problem becomes crucial for many applications on security, type checking transformations, and consistency of XML specifications.

Core-XPath (introduced in [6]) is the fragment of XPath that captures all the navigational behavior of XPath. It has been well studied and its satisfiability problem is known to be decidable even in the presence of DTDs. We consider an extension of this language with the possibility to make equality and inequality tests between attributes of elements in the XML document. This logic is named Core-Data-XPath in [2], and as shown in [5], its satisfiability problem is undecidable. It is then reasonable to study the interaction between different navigational fragments of XPath with equality tests to be able to find decidable and computationally well-behaved fragments. In the present work, we focus on the downward-looking fragments of XPath, where navigation between elements can only be done in the downward direction.

Our main contribution is that the satisfiability problem for $\mathsf{XPath}(\downarrow^*, \downarrow, =)$ is decidable. This is the fragment with data equality and inequality tests, with the $\downarrow^*$ axis that can access descendant nodes at any depth and the $\downarrow$ axis to access child elements. We prove a stronger result, showing the decidability of the satisfiability of $\mathsf{regXPath}(\downarrow, =)$, which is the extension of $\mathsf{XPath}(\downarrow^*, \downarrow, =)$ with the Kleene star operator to take reflexive-transitive closures of arbitrary path expressions. Moreover, we nail down the precise complexity showing an ExpTime decision procedure (recall that $\mathsf{XPath}(\downarrow, \downarrow^*)$ is already ExpTime-hard [1]). In order to do this, we introduce a new class of automata that captures all the expressivity of $\mathsf{regXPath}(\downarrow, =)$. On the other hand, we prove that the fragment $\mathsf{XPath}(\downarrow^*, =)$ without the $\downarrow$ axis is ExpTime-hard, even for a restricted fragment of $\mathsf{XPath}(\downarrow^*, =)$ without unions of path expressions. This reduction can only be done by using data equality tests, as the corresponding fragment $\mathsf{XPath}(\downarrow^*)$ without unions is shown to be PSpace-complete. We thus prove that the satisfiability problem for $\mathsf{XPath}(\downarrow^*, =)$, $\mathsf{XPath}(\downarrow^*, \downarrow, =)$ and $\mathsf{regXPath}(\downarrow, =)$ are all ExpTime-complete. Additionally, we present a natural fragment of $\mathsf{XPath}(\downarrow^*, =)$ that is PSpace-complete. We complete the picture showing that satisfiability for $\mathsf{XPath}(\downarrow, =)$ is also PSpace-complete. Altogether, we establish the precise com-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODS'09, June 29–July 2, 2009, Providence, Rhode Island, USA.
Copyright 2009 ACM 978-1-60558-553-6 /09/06 ...$5.00.

plexity for all downward fragments of XPath with and without data tests (cf. Figure 4 in Section 6).

Related work

In [1] there is a study of the satisfiability problem for many XPath logics, mostly fragments without negation or without data equality tests. Also, the fragment $\mathsf{XPath}(\downarrow,=)$ is proved to be in NExpTime. We improve this result by providing an optimal PSpace upper bound. It is also known that $\mathsf{XPath}(\downarrow)$ is already PSpace-hard, and in this work we match the upper bound showing PSpace-completeness. $\mathsf{XPath}(\downarrow,\downarrow^*)$ is proved ExpTime-complete in [1], and in this work we show that this complexity is preserved in the presence of data values and even under closure with Kleene star. We also consider a fragment that is not mentioned in [1]: $\mathsf{XPath}(\downarrow^*,=)$, and show that $\mathsf{XPath}(\downarrow^*)$ is PSpace-complete while $\mathsf{XPath}(\downarrow^*,=)$ is ExpTime-complete. In this case, data tests do make a real difference in complexity.

First-order logic with two variables and data equality tests is investigated in [2]. Although in the absence of data values FO^2 is expressive-equivalent to Core-XPath (cf. [8]), FO^2 with data equality tests becomes incomparable with respect to all the data aware fragments treated here. [2] also shows the decidability of a fragment of $\mathsf{XPath}(\uparrow,\downarrow,\leftarrow,\rightarrow,=)$ with sibling and upward axes but restricted to local elements accessible by a 'one step' relation, and to data formulæ of the kind $\varepsilon = p$ (or $\neq$). However, most of the fragments we treat here disallow upward and sibling axes but allow the descendant $\downarrow^*$ axis and arbitrary $p = p'$ data test expressions.

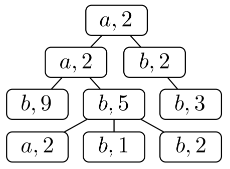
In [7] a fragment of $\mathsf{XPath}(\downarrow,\downarrow^*,\rightarrow,\rightarrow^*,=)$ is treated, denominated 'forward XPath'. In the cited work, the full set of downward and rightward axes are allowed, while the fragments treated here only allow the downward axis. As in [2], the language is restricted to data test formulæ of the form $\varepsilon = p$ contrary to the ones studied here, and hence no decidability results can be inferred. It is shown that its satisfiability problem is decidable, but with a non-primitive recursive algorithm, while in our work all the fragments considered are in ExpTime. The question of whether the forward fragment with arbitrary tests is decidable is still open.

2. STATEMENT OF THE PROBLEM AND MAIN RESULT

2.1 Data trees

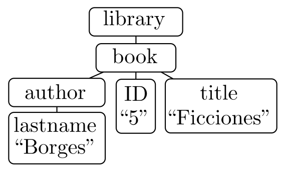
The structure of an XML document can be seen as an unranked tree with attributes and data values in its nodes. We work with an abstraction that we call *data tree*, that is, an unranked finite tree where every node contains a *symbol* from a finite alphabet Σ and a *data value* from some infinite domain Δ. Below we show an example of a data tree with $\Sigma = \{a, b\}$ and $\Delta = \mathbb{N}$.

Example 1.



It is important to mention that this model has only one data value on each node, whilst an XML document element may typically have several (0 or more) *attributes*, each with a data value associated. We address this issue by coding each attribute element by a child as shown next.

```
<library>
   <book ID="5"
    title="Ficciones">
      <author
       lastname="Borges"/>
   </book>
</library>
```



In the above example the data values of the nodes tagged with non-attribute elements (library, book, author) may have any data value. For every fragment of XPath with the child axis ($\downarrow$) we can enforce that attributes are leaves and we can translate any XPath expression on XML to an equivalent one on data trees. For the fragments with the single descendant axis $\downarrow^*$, this is not true anymore, but a more careful analysis shows that all complexity results still hold in this case. Indeed, any $\mathsf{XPath}(\downarrow^*,=)$ formula on data trees can be seen as an $\mathsf{XPath}(\downarrow^*,=)$ formula on XMLs that makes use of only one fixed attribute, and we can thus transfer the lower bound (the upper bound follows from the more expressive fragment $\mathsf{XPath}(\downarrow^*,\downarrow,=)$). Summing up, all our forthcoming results also hold on arbitrary XML documents with multiple attributes per element. This implies, for example, that the results hold for the satisfiability problem for data trees that may have some nodes with no data values.

We use the standard representation of unranked trees by a nonempty, prefix-closed set T of elements from $\mathbb{N}^*$ such that whenever $x(i+1) \in T$ then $xi \in T$; together with a labeling function $\sigma : T \to \Sigma$ and a data value function $\delta : T \to \Delta$. A *data tree model* $\mathcal{T}$ is then a tuple $\langle T, \sigma, \delta\rangle$, and we call $\mathsf{Pos}(\mathcal{T}) = T$ the set of *positions* of $\mathcal{T}$. We denote by $\mathcal{T}|_x$ the subtree of $\mathcal{T}$ with root x, and $\delta(\mathcal{T}) = \{\delta(x) \mid x \in \mathsf{Pos}(\mathcal{T})\}$. In the Example 1 presented before, $T = \{\varepsilon, 1, 2, 11, 12, 121, 122, 123, 22\}$.

2.2 The logic XPath

We work with a simplification of XPath, stripped of its syntactic sugar. Actually, we consider fragments of XPath that correspond to the navigational part of XPath 1.0 with data equality and inequality. Let us give the formal definition of this logic. XPath is a two-sorted language, with *path* expressions $(\alpha, \beta, \dots)$ and *node* expressions $(\varphi, \psi, \dots)$. The fragment $\mathsf{XPath}(\mathcal{O},=)$, with $\mathcal{O} \subseteq \{\downarrow,\downarrow^*\}$ is defined by mutual recursion as follows:

$$\alpha ::= o \mid \alpha[\varphi] \mid [\varphi]\alpha \mid \alpha\beta \mid \alpha \cup \beta \qquad o \in \mathcal{O} \cup \{\varepsilon\}$$

$$\varphi ::= a \mid \neg\varphi \mid \varphi \wedge \psi \mid \langle\alpha\rangle \mid \alpha \circledast \beta \qquad \circledast \in \{=,\neq\}, a \in \Sigma$$

A *formula* of $\mathsf{XPath}(\mathcal{O},=)$ is either a node expression or a path expression of the logic. $\mathsf{XPath}(\mathcal{O})$ is the fragment $\mathsf{XPath}(\mathcal{O},=)$ without the node expressions of the form $\alpha \circledast \beta$.

There have been efforts to extend this navigational core of XPath in order to have the full expressivity of MSO, e.g. by adding a least fix-point operator (cf. [9, Sect. 4.2]), but these logics generally lack clarity and simplicity. However, a form of recursion can be added by means of the Kleene star, which allows to take the transitive closure of any path expression. Although in general this is not enough to already have MSO –as shown in [10]–, it does give an intuitive language with counting ability. By $\mathsf{regXPath}(\mathcal{O},=)$ we refer to the language where path expressions are extended

$$\alpha ::= o \mid \alpha[\varphi] \mid [\varphi]\alpha \mid \alpha\beta \mid \alpha \cup \beta \mid \alpha^* \qquad o \in \mathcal{O} \cup \{\varepsilon\}$$

by allowing the Kleene star on *any* path expression. In terms of expressivity, we see that $\mathsf{XPath}(\downarrow^*,=) \subset \mathsf{XPath}(\downarrow^*,\downarrow,=) \subset \mathsf{regXPath}(\downarrow,=) = \mathsf{regXPath}(\downarrow^*,\downarrow,=)$.

Let $\mathcal{T} = \langle T, \sigma, \delta \rangle$. We define the semantics of XPath:

$$\begin{aligned}
[\![\downarrow]\!]^{\mathcal{T}} &= \{(x, xi) \mid xi \in T\} \\
[\![\alpha^*]\!]^{\mathcal{T}} &= \text{the reflexive transitive closure of } [\![\alpha]\!]^{\mathcal{T}} \\
[\![\varepsilon]\!]^{\mathcal{T}} &= \{(x, x) \mid x \in T\} \\
[\![\alpha\beta]\!]^{\mathcal{T}} &= \{(x, z) \mid \exists y.(x, y) \in [\![\alpha]\!]^{\mathcal{T}}, (y, z) \in [\![\beta]\!]^{\mathcal{T}}\} \\
[\![\alpha \cup \beta]\!]^{\mathcal{T}} &= [\![\alpha]\!]^{\mathcal{T}} \cup [\![\beta]\!]^{\mathcal{T}} \\
[\![\alpha[\varphi]]\!]^{\mathcal{T}} &= \{(x, y) \in [\![\alpha]\!]^{\mathcal{T}} \mid y \in [\![\varphi]\!]^{\mathcal{T}}\} \\
[\![[\varphi]\alpha]\!]^{\mathcal{T}} &= \{(x, y) \in [\![\alpha]\!]^{\mathcal{T}} \mid x \in [\![\varphi]\!]^{\mathcal{T}}\} \\
[\![a]\!]^{\mathcal{T}} &= \{x \in T \mid \sigma(x) = a\} \\
[\![\langle\alpha\rangle]\!]^{\mathcal{T}} &= \{x \in T \mid \exists y.(x, y) \in [\![\alpha]\!]^{\mathcal{T}}\} \\
[\![\neg\varphi]\!]^{\mathcal{T}} &= T \setminus [\![\varphi]\!]^{\mathcal{T}} \\
[\![\varphi \wedge \psi]\!]^{\mathcal{T}} &= [\![\varphi]\!]^{\mathcal{T}} \cap [\![\psi]\!]^{\mathcal{T}} \\
[\![\alpha = \beta]\!]^{\mathcal{T}} &= \{x \in T \mid \exists y, z.(x, y) \in [\![\alpha]\!]^{\mathcal{T}}, \\
&\qquad (x, z) \in [\![\beta]\!]^{\mathcal{T}}, \delta(y) = \delta(z)\} \\
[\![\alpha \neq \beta]\!]^{\mathcal{T}} &= \{x \in T \mid \exists y, z.(x, y) \in [\![\alpha]\!]^{\mathcal{T}}, \\
&\qquad (x, z) \in [\![\beta]\!]^{\mathcal{T}}, \delta(y) \neq \delta(z)\}
\end{aligned}$$

For instance, in the model of Example 1,

$$[\![\langle\downarrow^*[b \wedge \downarrow[b] \neq \downarrow[b]]\rangle]\!]^{\mathcal{T}} = \{\varepsilon, 1, 12\}.$$

We now state the problem we will address.

Definition 1. The *satisfiability problem* SAT-$\mathcal{L}$ consists in, given an $\mathcal{L}$-formula η, to decide whether there exists a data tree $\mathcal{T}$ such that $[\![\eta]\!]^{\mathcal{T}} \neq \emptyset$.

It turns out that –as we are working with downward-looking fragments of XPath– this is equivalent to asking if there is a model where the formula η is satisfied at its root. Moreover, for this problem we can restrict ourselves to the case where η is a node expression. We remind the reader that although we state the problem in terms of data trees, all our results hold on the class of all XML documents with multiple attributes.

2.3 Main contribution

Our main results are the following.

THEOREM 1. *SAT-regXPath*$(\downarrow, =)$ *is decidable, with complexity in* EXPTIME.

THEOREM 2. *SAT-XPath*$(\downarrow^*, =)$ *is hard for* EXPTIME.

Consequently, for any logic $\mathcal{L} \in \{$XPath$(\downarrow^*, =)$, XPath$(\downarrow^*, \downarrow, =)$, regXPath$(\downarrow, =)\}$, SAT-$\mathcal{L}$ is EXPTIME-complete.

The strategy of the proof can be outlined as follows.

1. We introduce a model of automata that captures all the expressivity of regXPath$(\downarrow, =)$. Automata of this new class are called *Bottom-up Interleaved Path automata* (BIP). The automaton relies on an interaction between the runs of two kinds of automata, which corresponds to the two sorts of formulæ of XPath.
2. We show that the translation from regXPath$(\downarrow, =)$ to BIP automata can be done in PTIME.
3. The main result is that the emptiness problem for the class of BIP automata is in EXPTIME. We show this by a reduction to the non-emptiness problem of classical bottom-up tree automata over trees with bounded branching width. Given a BIP automaton M, we construct a bottom-up tree automaton $\mathcal{A}$ whose states describe the behavior of M for certain data values. We show it is sufficient to consider both the branching width and the number of data values to be polynomially bounded by the BIP automaton M. We show that M is non-empty iff $\mathcal{A}$ is non-empty.
4. Finally, we prove that XPath$(\downarrow^*, =)$ is EXPTIME-hard by a reduction from the two-player corridor tiling game. Here, the challenge is to be able to move from one square of the game board to the next one, without counting with the '$\downarrow$' operator in the language.

3. A NEW CLASS OF AUTOMATA

We introduce a new automaton that we call *Bottom-up Interleaved Path automata* (BIP for short) that in its transition function uses another automaton, the *Pathfinder automaton* that runs over the already executed BIP run. This interleaving mechanism of running one automaton as a condition of the transition function of the other one corresponds exactly to the two sorts of formulæ of XPath. Thanks to this duality, we obtain an automaton that captures the whole expressivity of regXPath$(\downarrow, =)$. It is worth noting that although the BIP automaton does not contain 'registers' *per se*, it can do an unbounded number of equality and inequality tests between any pair of nodes of the tree. However strong this automaton may appear to be, we prove that the emptiness problem is only in EXPTIME.

3.1 Definitions

A *Pathfinder automaton* is a bottom-up non deterministic automaton that basically can only recognize a *path* from some node to the root and retrieve *one* data value from it. The automaton retrieves a data value d with state k if there is a run that starts in a node with data value d and ends at the root with state k. Its definition is very weak as it can only *retrieve* a data value, but it cannot test it against any other. It runs over a data tree over the alphabet $\Sigma = 2^Q$ (i.e., where the labeling function $\sigma : T \to 2^Q$ tags each node with a subset of Q), where Q is a finite set of symbols that –as we will see shortly– consists of the states of another automaton (the BIP). It is defined as the tuple $\mathcal{P} = \langle K, k_I, Q, \nu \rangle$ where K is a finite set of states, $k_I \in K$ is a distinguished initial state, and ν is the transition function. At each transition, the automaton can either (1) check the presence of some element of Q in the label of the node (we call this a 'non-moving' transition), or (2) move up in the tree (we call this a 'moving' transition).

$$\nu : (Q \cup \{\mathsf{up}\}) \times K \to 2^K$$

For example, the non-moving transition $\nu(q_1, k_1) = \{k_2, k_3\}$ indicates that if we are in state k_1 at some node labeled with a set $S \subseteq Q$ such that $q_1 \in S$, then we can label it with one of the states among k_2, k_3. On the other hand, if $\nu(\mathsf{up}, k_1) = \{k_4\}$ and if we are in state k_1, then the *father* of this node can be labeled with state k_4.

Observe that, although this automaton runs on models labeled with *subsets* of Q, the transition function takes only *one* state of Q at a time, its intended meaning being that it applies to any set that contains the specified state. We do so in order to obtain a polynomial time translation from XPath$(\downarrow^*, \downarrow, =)$ to a pathfinder automaton and to prove the precise upper-bound of EXPTIME. Otherwise, we would have a translation on models with an exponential number of states. This will become clear in the following.

A *run* ρ of a pathfinder $\mathcal{P} = \langle K, k_I, Q, \nu\rangle$ on a data tree $\mathcal{T} = \langle T, \sigma, \delta\rangle$ is a non-empty list of states with positions $\rho \in (K \times \mathsf{Pos}(\mathcal{T}))^+$. We denote by $\rho(i)$ the ith element of the run, starting from 0. The run must be such that $\rho(0) = (k, p)$ with $k = k_I$, and $\rho(N) = (k', p)$ with $p = \varepsilon$ for $N = |\rho| - 1$. For any two positions $\rho(i) = (k', x')$, $\rho(i+1) = (k, x)$ either (1) $x = x'$ and a 'non-moving' transition applies between k and k' for x, or (2) $xn = x'$ for some $n \in \mathbb{N}$ and a 'moving' transition applies between k and k' for xn. We define that a 'non moving' transition *applies* between k and k' for x iff $k \in \nu(k', q)$ for some $q \in \sigma(x)$, and that a 'moving' transition applies iff $k \in \nu(k', \mathsf{up})$.

Runs of pathfinder automata are noted by the symbol ρ. The *output* of a run ρ, denoted by $o(\rho)$, is defined as the pair (k, d), where $\rho(N) = (k, \varepsilon)$, and $d = \delta(p)$ with $\rho(0) = (k_I, p)$, $N = |\rho| - 1$. We also define the *non-moving closure* of a pathfinder $\mathcal{P}$ w.r.t. a state k and label $S \subseteq Q$ (noted $cl(k, S)$) as the set of states k' that can be reached by 'non-moving' transitions on a node labeled S starting with the state k. Formally, $cl(k, S) := \bigcup_{n\geq 0}(f_S^n(\{k\}))$, with $f_S(K) := \{k_1 \mid k_1 \in \nu(q, k_2), q \in S, k_2 \in K\}$. Observe that this set can be built in time polynomial in the set S and the automaton $\mathcal{P}$. We do not give accepting conditions because this automaton is used by the BIP automaton as we shall see.

Example 2. Consider $\mathcal{P} = \langle\{k_I, k_1, k_{\downarrow 1}, k_2, k_{\downarrow 2}, k_3\}, k_I, \{q_1, q_2, q_f\}, \nu\rangle$ that recognizes $(q_1 q_2)^+$, that is, where $\nu(k_I, q_2) = \{k_2\}$, $\nu(k_2, \mathsf{up}) = \{k_{\downarrow 2}\}$, $\nu(k_{\downarrow 2}, q_1) = \{k_1\}$, $\nu(k_1, \mathsf{up}) = \{k_{\downarrow 1}\}$, $\nu(k_{\downarrow 1}, q_2) = \{k_2\}$, $\nu(k_I, q_1) = \{k_3\}$, $\nu(k_3, \mathsf{up}) = \{k_3\}$. Any run of $\mathcal{P}$ that ends in k_1 retrieves a data value that can be accessed by a 'path' like $Q_1^1 Q_2^2 \ldots Q_1^{t-1} Q_2^t$ where for any i, $q_1 \in Q_1^i$ and $q_2 \in Q_2^i$. Any run that ends in k_3 retrieves a data value from a node that is labeled by q_1.

As we show next, the runs of the pathfinder are the basic means for the BIP automaton to test for data (in)equalities.

A *bottom-up Interleaved Path automaton* (BIP) M is a tuple $\langle\Sigma, Q, \mu, F, \mathcal{P}\rangle$, where Σ is a finite set of symbols, Q is a finite set of states, $F \subseteq Q$ is the set of final states, $\mathcal{P} = \langle K, k_I, 2^Q, \nu\rangle$ is a Pathfinder automaton, and $\mu : Q \to \mathsf{Form}_M$ is the transition function, where Form_M is defined:

$$\varphi ::= a \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\varphi \mid \exists(k_1, k_2)^{\circledast}$$

where $\varphi, \psi \in \mathsf{Form}_M, a \in \Sigma, \circledast \in \{=, \neq\}, k_1, k_2 \in K$.

Intuitively, $\exists(k_1, k_2)^{\circledast}$ tests for the values retrieved by pathfinder runs (quantified existentially).

A *run* of the BIP automaton over a data tree $\mathcal{T}$ is a labeling function $\lambda : \mathsf{Pos}(\mathcal{T}) \to 2^Q$. In general if $\mathcal{T} = \langle T, \sigma, \delta\rangle$ we define as $\lambda(\mathcal{T})$ the data tree $\langle T, \lambda, \delta\rangle$. Remember that $\mathcal{T}|_n$ is the subtree that has n as a root, and let $\lambda|_n(i) = \lambda(ni)$. A run λ must fulfill, for every position n, that $q \in \lambda(n)$ iff $\mathcal{T}|_n, \lambda|_n \models \mu(q)$, where $\models$ is defined as follows

- $\mathcal{T}, \lambda \models a$ iff $\sigma(\varepsilon) = a$, that is, the root's symbol is a,
- $\mathcal{T}, \lambda \models \neg\varphi$ iff $\mathcal{T}, \lambda \not\models \varphi$, and all boolean connectors are defined in the standard way, and
- $\mathcal{T}, \lambda \models \exists(k_1, k_2)^{\circledast}$ with $\circledast \in \{=, \neq\}$ iff there exist two runs ρ_1, ρ_2 of $\mathcal{P}$ over the run-labeled tree $\lambda(\mathcal{T})$ such that $o(\rho_1) = (k_1, d)$, $o(\rho_2) = (k_2, d')$ and $d \circledast d'$.

The run is *accepting* if $\lambda(\varepsilon) \cap F \neq \emptyset$.

Example 3. Consider the BIP automaton $\langle\Sigma, \{q_1, q_2, q_f\}, \mu, \{q_f\}, \mathcal{P}\rangle$, where $\Sigma = \{a, b\}$, and $\mathcal{P}$ is the one defined in Example 2. We can define μ to accept the trees that contain two elements accessible by a $(ab)^+$ path from the root, with different data values: $\mu(q_f) = \exists(k_{\downarrow 1}, k_{\downarrow 1})^{\neq} \wedge \neg\exists(k_I, k_3)^{\neq}$, $\mu(q_1) = a$, $\mu(q_2) = b$. Note that it corresponds to the XPath formula $(\downarrow [a] \downarrow [b])^+ \neq (\downarrow [a] \downarrow [b])^+ \wedge \neg\varepsilon \neq \downarrow^* [a]$, and that it accepts the tree of Example 1.

3.2 From regXPath to BIP automata

THEOREM 3. *Given a node expression $\eta \in \mathsf{regXPath}(\downarrow, =)$, there is a* BIP *automaton M such that for any model $\mathcal{T}$, $\varepsilon \in [\![\eta]\!]^{\mathcal{T}}$ iff M accepts $\mathcal{T}$. Moreover, this automaton can be obtained in* PTIME.

PROOF. Let η be a formula of $\mathsf{regXPath}(\downarrow, =)$. We build the BIP automaton $M = \langle\Sigma, Q, \mu, F, \mathcal{P}\rangle$ where $\Sigma = \{a \mid a$ a label in $\eta\} \cup \{a_\perp\}$, $Q = \{q_\psi \mid \psi$ is a node expression in $\mathsf{sub}(\eta)\} \cup \{q_\top\}$ where $\mathsf{sub}(\eta)$ is the set of subformulæ of η, and $F = \{q_\eta\}$. Intuitively, for every state q_ψ where ψ is a node expression, μ associates it to a formula that is –exactly as ψ– a boolean combination of symbol and data equality tests. For formulæ whose principal operator is a boolean connector, the transition is straightforward. E.g., $\mu(q_{\psi_1 \wedge \psi_2}) = \mu(q_{\psi_1}) \wedge \mu(q_{\psi_2})$. If ψ is of the form $\alpha \circledast \alpha'$, then $\mu(q_\psi) = \exists(k_\alpha, k_{\alpha'})^{\circledast}$, and if $\psi = \langle\alpha\rangle$, $\mu(q_\psi) = \exists(k_\alpha, k_\alpha)^{=}$.

On the other hand, any path expression $\alpha \in \mathsf{sub}(\eta)$ can be seen as a regular expression over the alphabet $\Sigma_\eta = \{e \mid e$ is a node expression of $\eta\} \cup \{\downarrow\}$. Then, for any path α we can make the standard PTIME translation of α^r into a NFA over Σ, where α^r stands for the *reverse* of α (it is enough to exactly reverse the symbols, as path expressions are closed under reversal). This is necessary because although XPath path expressions name the path from the root to the leaves, the pathfinder automaton reads the branch from the leaves to the root. We thus obtain a NFA $A_\alpha = \langle K_\alpha, \Sigma_\eta, k_\alpha^0, F_\alpha, \delta_\alpha\rangle$ for each path expression $\alpha \in \mathsf{sub}(\eta)$ where we name the states $K_\alpha = \{k_\alpha^0, k_\alpha^1, \ldots\}$. We then define $\mathcal{P} = \langle K, k_I, Q, \nu\rangle$ the pathfinder where $K = \{k_I\} \cup \bigcup_\alpha(\{k_\alpha\} \cup K_\alpha)$, and

$$\nu(k_I, q_\varphi) = \{k_\alpha^0 \mid k_\alpha^0 \in K\},$$
$$\nu(k_\alpha^i, q_\varphi) = \delta_\alpha(\varphi, k_\alpha^i) \cup \{k_\alpha \mid \delta_\alpha(\varphi, k_\alpha^i) \cap F_\alpha \neq \emptyset\},$$
$$\nu(k_\alpha^i, \mathsf{up}) = \delta_\alpha(\downarrow, k_\alpha^i) \cup \{k_\alpha \mid \delta_\alpha(\varphi, k_\alpha^i) \cap F_\alpha \neq \emptyset\}.$$

We can check that all the runs that end in k_α –for α some path subformula of η– retrieve a data value of a path accessed via α, and conversely that it can retrieve all of them.

It is not surprising that the translation is so direct, as the BIP automaton mimicks closely XPath semantics. □

BIP automata are more expressive than $\mathsf{regXPath}(\downarrow, =)$, but if we restrict the definition of BIP to have a bounded number of mutual recursions between the BIP and the pathfinder, we precisely characterize $\mathsf{regXPath}(\downarrow, =)$: for each BIP there is an equivalent $\mathsf{regXPath}(\downarrow, =)$ formula, and vice-versa. In a sense, the restriction disallows the existence of two states q, k (the former of BIP, the latter of pathfinder) in mutual recursion, where k is named in $\mu(q)$ and q in $\nu(k, \cdot)$.

4. MAIN RESULT

We devote this section to prove that emptiness of BIP automata is in EXPTIME and that the satisfiability problem for $\mathsf{XPath}(\downarrow^*, =)$ is EXPTIME-hard. In this way we obtain that the satisfiability problem for $\mathsf{XPath}(\downarrow^*, =)$, $\mathsf{XPath}(\downarrow^*, \downarrow, =)$ and $\mathsf{regXPath}(\downarrow, =)$ are all ExpTime-complete.

4.1 Upper bound

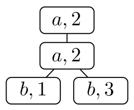
Theorem 4. *Emptiness of* bip *automata is in* ExpTime.

Proof. The proof consists in a reduction from the bip emptiness problem into the emptiness of a classical bottom-up non deterministic tree automaton over trees with bounded branching width. Moreover, the tree automaton has at most an exponential number of states, and therefore its emptiness problem can be solved in ExpTime. A state of this automaton (that we call *extended state* to avoid confusion with states of bip automata) contains the *description* of the behavior of the bip automaton w.r.t. some data values. More precisely, each data value d is described by the set of states of the pathfinder by which d can be reached (cf. Ex. 4 below).

We guarantee that for each extended state reachable by the tree automaton, there is a witnessing data tree that consists in any tree that reaches this state at the root, together with an assignment for data values. For the extended states that correspond to the leaves, the witnessing data tree is the leaf with an arbitrary data value. For an extended state of an inner node, the witnessing tree is constructed bottom-up, by identifying the data values that are described in the states of the witnessing subtrees inductively obtained, and 'merging' them according to an equivalence relation associated to the transition.

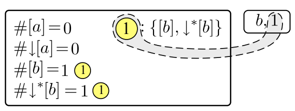
In Proposition 1, we show that given an accepting run of the tree automaton, we can easily build a run of the bip automaton on a witnessing data tree, and in Proposition 2 we show that if there is an accepting run on the bip automaton, there must be a model with an accepting run in the tree automaton. As a byproduct of this reduction, we obtain a small model property. We establish that if a bip automaton accepts at least one data tree, then it accepts in particular a model with polynomial branching width and exponential height, such that for any pair of disjoint subtrees there are only a polynomial number of data values in common.

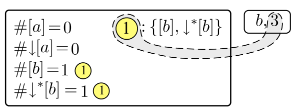
Example 4. We give the main idea for checking emptiness of bip automata by abstracting runs. Let M be the bip that accepts the models satisfying $\varphi = \langle\downarrow[a]\rangle \wedge \neg[a] \neq \downarrow[a] \wedge \downarrow^*[b] \neq \downarrow^*[b]$, and consider the model below that verifies φ.



Below, we give an intuition about the nature of the extended states of the tree automaton $\mathcal{A}_M$ built from M, and what would be an accepting run for this particular data tree.

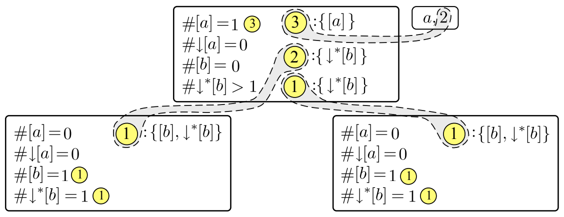
To each subtree, we associate an extended state that contains the description of some data values. Each data value d (here represented by a balloon like ①) is described by a set of states of the pathfinder automaton. Here, for simplicity's sake, such states are represented by (sub)path expressions that can reach d. Additionally, the extended state specifies, for each path expression, whether it can retrieve (a) only one data value, (b) more than one, or (c) no data values. In the case (a), we specify which is the only data value retrieved.



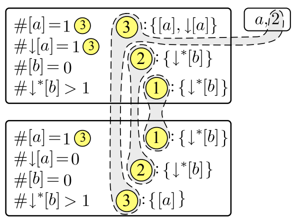


Both leaves have the same extended state. We read the state as follows: there is only one data value reachable by the path $\downarrow^*[b]$, and this data value is denoted by ①. There is also only one data value reached by $[b]$, which is also denoted by ① (hence, they are the same). There are no data values reachable by a path $\downarrow[a]$ (as it is a leaf) or $[a]$ (as it is labeled 'b'). Finally, we describe the data value denoted by ① with the set of path expressions that may reach it: $\{[b], \downarrow^*[b]\}$. Intuitively, ① makes reference to the current node's data value, that is why ① and the datum are surrounded by the same area, to denote that they are equal.

In the next step, we apply a transition, which specifies how the data values between the balloons are merged. In this case we demand that the balloon of the left leaf and that of the right must be *different*, otherwise they would be surrounded in the same grey area. Consequently, we will have that there are more than one data value reached by $\downarrow^*[b]$, which is reflected in the extended state.



In this transition, we define balloons that describe the same data values as the ones of the leaves (the case of ① and ②), and we define ③ as denoting the data value of the current element $\langle a, 2\rangle$. Observe that the extended state of the root depends only on (1) the root's symbol, (2) the descriptions of the data values of its children, and (3) the way of 'merging' these data values.



Here, we show how the last transition leads to a final state. Once this transition is performed we can see that it implies that the formula φ is satisfied. In the following, we show how to build this tree automaton systematically.

Let M be a bip automaton $M = \langle \Sigma, Q, \mu, F, \mathcal{P}\rangle$ with $\mathcal{P} = \langle K, k_I, Q, \nu\rangle$. In the development below we make use of two parameters t_0 (related to the number of data values described in each extended state) and u_0 (the maximum branching width of the witnessing tree) which we assume to be bounded by two polynomials on $|K|$. As usual, we write '$\circledast$' to denote any element of $\{=, \neq\}$. We define a tree automaton $\mathcal{A}_M = \langle \Sigma, Q_\mathcal{A}, \tau, F_\mathcal{A}\rangle$ where $Q_\mathcal{A}$ is the set of extended states, and $F_\mathcal{A}$ is the set of final states. Finally, $\tau \subseteq 2^{Q_\mathcal{A}}_{\leq u_0} \times \Sigma \times Q_\mathcal{A}$ (where $2^{Q_\mathcal{A}}_{\leq u_0}$ stands for the set of subsets of $Q_\mathcal{A}$ with at most u_0 elements) is the transition function that, given the root's symbol and the set of states of its children, labels the root with a state.

Abstracting runs.

An *extended state* is the building block to abstract the runs of M. It is a pair $c = \langle v, D\rangle$ where

- v is a *valuation*, i.e., a function $v : \mathsf{atForm}_M \to \{0, 1\}$ that specifies which of the formulæ of M hold at the abstracted node. Here, atForm_M is the (finite) subset of *atomic formulæ* (i.e., with no boolean connectors) of Form_M.

- $D = \langle D^{\odot}, D^{=}\rangle$ consists in two descriptions of a polynomial number of data values. $D^{=} \in (2^K)^{|K|}$ describes at most $|K|$ data values, and $D^{\odot} \in (2^K)^{t_0}$ at most t_0. We will later detail exactly what this represents.

It is easy to check that $|Q_{\mathcal{A}}|$ is exponential in $|M|$. In Example 4, $D^{=}$ is represented by the path expressions α s.t. $\#\alpha = 1$ and the data descriptions associated to them, and $D^{\odot}$ by the remaining data descriptions. Each extended state represents a class of models. Before presenting the abstraction, let us fix some notation.

Let $\mathcal{T} = \langle T, \sigma, \delta\rangle$ be a data tree and λ be a run of M on $\mathcal{T}$. We denote by $\lambda(\mathcal{T})$ the data tree labeled with the run $\langle T, \lambda, \delta\rangle$. Given a valuation $v : \mathsf{atForm}_M \to \{0,1\}$, we define $\mathcal{C}(v) = \{q \mid q \in Q \text{ with } \mu(q) \text{ true under the valuation } v\}$. If $d \in \Delta$, let $\mathit{Reach}(d) = \{k \mid \text{there is } \rho \text{ run of } \mathcal{P} \text{ on } \lambda(\mathcal{T}), \text{ such that } o(\rho) = (k,d)\}$. We fix χ to be any bijection $\chi : K \to [1..|K|]$, and $\chi(k) = i$ stands just as a correlation between $k \in K$ and the element described in position i of the tuple $D^{=}$ (we note this as $D^{=}(i)$). We then say that an extended state $c = \langle v, D\rangle$ *abstracts* the run λ of M on $\mathcal{T}$ (notation: $\mathcal{T} \rhd c$) iff $\lambda(\varepsilon) = \mathcal{C}(v)$ and the following two conditions are met.

- For every $k \in K$, either $D^{=}(\chi(k)) = \mathit{Reach}(d_0)$ for $d_0 \in \Delta$ such that for every run ρ of $\mathcal{P}$ on $\lambda(\mathcal{T})$, if $o(\rho) = (k,d)$ then $d = d_0$; or $D^{=}(\chi(k)) = \emptyset$ if there is no such d_0.
- We only state that $D^{\odot}$ describes some data values, with the possibility of even have 'empty' descriptions as well if $D^{\odot}(i) = \emptyset$. More formally, there are at most t_0 data values $d_1, \dots, d_{t_0} \in \Delta$ where the ith component $D^{\odot}(i)$ is the empty set, or $D^{\odot}(i) = \mathit{Reach}(d_i)$, provided that d_i is not already described in $D^{=}$. I.e., we must ensure that $(\bigcup_i D^{\odot}(i)) \cap \{k \mid D^{=}(\chi(k)) \neq \emptyset\} = \emptyset$.

The set $Q_{\mathcal{A}}$ consists of *all* the exponentially many possible extended states, and $F_{\mathcal{A}} = \{\langle v, D\rangle \in Q_{\mathcal{A}} \mid \mathcal{C}(v) \cap F \neq \emptyset\}$. We next describe the transition function τ.

We start with the transitions that correspond to the leaves. Let $t = \langle \emptyset, a, c\rangle$ with c an extended state and a a symbol. We define that $t \in \tau$ iff $(a,1) \rhd c$, where $(a,1)$ is the singleton tree with symbol a and datum 1 (it could be any).

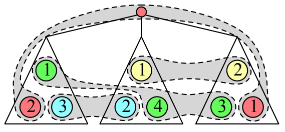
Now we show how to obtain the recursive transitions. We explain how, from $u \leq u_0$ extended states, we can construct the state that is supposed to represent the root of the tree.

For convenience of notation, by c_i we refer to the extended state $\langle v_i, D_i\rangle$. Suppose we have a tuple $t = \langle \{c_1 \dots c_u\}, a, c_0\rangle$ with $u > 0$. We now show how to check if $t \in \tau$. Remember that $\mathcal{A}$ is *non-deterministic* and the different possibilities for c_0 depend on the way the data values in $c_1 \dots c_u$ are merged. We must then describe how these descriptions are merged together, and ensure that transitions are consistent with the merging. For example, if c_0 describes a datum that can be reached in two steps with label a, or in at least one step with label b (e.g. because we want to check '$\downarrow\downarrow[a] = \downarrow\downarrow^*[b]$') then some c_i must describe a datum accessible through '$\downarrow[a]$', some c_j must describe a datum accessible through '$\downarrow^*[b]$', and these two descriptions must be of the *same* data value.

We define that $t \in \tau$ iff there exists a *merging* $\equiv_E$ such that c_0 is *coherent* w.r.t. $\equiv_E$ and $\{c_1 \dots c_u\}$. We next define what is a 'merging' and which are the 'coherence' conditions.

Merging data values.

We describe the ways of merging the $u(t_0 + |K|)$ data values described by the (non-empty) elements of $D_1 \dots D_u$.



$$E = \{\langle(\odot,1,1),(\odot,2,4)\rangle, \langle(\odot,1,1),(\odot,3,3)\rangle, \langle(\odot,1,3),(\odot,2,2)\rangle, \langle(\odot,1,2),(\odot,3,1)\rangle, \langle(\odot,2,4),(\odot,3,3)\rangle, \langle(\odot,2,1),(\odot,3,2)\rangle, \langle\text{root},(\odot,3,1)\rangle, \dots\}$$

Figure 1: An example of merging, where $D^{=}$ may be assumed to be empty.

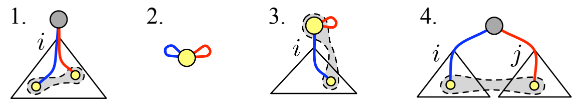


Figure 2: The 4 cases for making $\exists(k_1,k_2)^{=}$ true.

For this purpose, we consider an equivalence relation '$\equiv_E$' on $\{(\odot,i,j) \mid D_i^{\odot}(j) \neq \emptyset\} \cup \{(=,i,j) \mid D_i^{=}(j) \neq \emptyset\} \cup \{\text{root}\}$ that describes exactly *how* these data values are going to be collapsed between them and w.r.t. the root's data value. However we check one condition in $\equiv_E$. If $k_1 \in D_i^{=}(\chi(k_2))$ then either $D_i^{=}(\chi(k_1)) = \emptyset$ or $(=,i,\chi(k_1)) \equiv_E (=,i,\chi(k_2))$, as they make reference to the exact same data value (by definition of $D^{=}$). An example of merging is shown in Fig. 1.

We consider that the *only* data values that can be merged among the trees represented by the u extended states are those described in $\equiv_E$. In other words, we can consider that the witnessing data tree for c_0 is composed of the witnessing trees for each c_i with the following property. For every c_i we associate a data value to each description, and two data values from two different subtrees i,j are equal if and only if (1) they are described in c_i, c_j respectively, and (2) both descriptions are in same equivalence class of $\equiv_E$. Observe that this is a strong restriction, the emptiness algorithm of BIP relies on at most $u_0(t_0 + |K|)$ data values at every point of a branch. However, we remark that the automaton M can make at any step, any number of comparisons between any number of data values that can be found in the subtree.

Checking coherence of c_0 with respect to $\equiv_E$.

We require that $\langle \{c_1 \dots c_u\}, a, c_0\rangle \in \tau$ exactly when there exists a merging $\equiv_E$ such that the conditions below hold true. We first define the following relation: $\mathsf{step\text{-}up}(k',k)$ iff $k'' \in \nu(k', \mathsf{up})$ and $k \in \mathit{cl}(k'', \mathcal{C}(v_0))$.

To start with, we must verify that v_0 is correct. For $a' \in \Sigma$, $v_0(a') = 1$ iff $a' = a$. And $v_0(\exists(k_1,k_2)^{=}) = 1$ iff any of the 4 conditions below holds (they are depicted in Fig. 2).

1. There are some k_1', k_2' states that retrieve equal data in some subtree, and moving one step up with $\mathcal{P}$ we obtain k_1 and k_2. I.e., for some i, $v_i(\exists(k_1',k_2')^{=}) = 1$, $\mathsf{step\text{-}up}(k_1',k_1)$, and $\mathsf{step\text{-}up}(k_2',k_2)$.
2. Both k_1, k_2 can be obtained as runs that start and end at the root (and hence both carry root's data value) $k_1, k_2 \in \mathit{cl}(k_I, \mathcal{C}(v_0))$.
3. k_2 is like in the preceding case, and k_1 retrieves a data value declared in $\equiv_E$ to be equal to the root. For some i, $k_2 \in \mathit{cl}(k_I, \mathcal{C}(v_0))$, $\exists \ell, \alpha\ (\alpha, i, \ell) \equiv_E \text{root}$, $k_1' \in D_i^{\alpha}(\ell)$, $\mathsf{step\text{-}up}(k_1',k_1)$ (or the converse, swapping k_1 and k_2).
4. k_1 and k_2 retrieve data values from different subtrees that are equal according to the merging $\equiv_E$. For some i,j, there exist m, ℓ, α, β s.t. $k_1' \in D_i^{\alpha}(m)$, $k_2' \in D_j^{\beta}(\ell)$, $(\alpha,i,m) \equiv_E (\beta,j,\ell)$, $\mathsf{step\text{-}up}(k_1',k_1)$, $\mathsf{step\text{-}up}(k_2',k_2)$.

And $v_0(\exists(k_1,k_2)^{\neq}) = 1$ iff any of the similar conditions previously described in 1, 3, 4 holds (changing $=$ by $\neq$), or

4. One of the states is *not* described in $D^{=}$, and hence it retrieves at least 2 different data values, which means that the $\neq$ constraint can be met: $D_0^{=}(\chi(k_1)) = \emptyset$ and $v_0(\exists(k_2,k_2)^{=}) = 1$ (or the converse).

We must check now the coherence of $D_0^{=}$ and $D_0^{\odot}$. We concentrate on the former as it is the most involved. Let B_k be the set of all the descriptions of data values that can be reached at the root with state k, $B_k = \{(\alpha, i, j) \mid k' \in D_i^{\alpha}(j), \mathsf{step\text{-}up}(k', k)\}$. We require that $D_0^{=}(\chi(k)) \neq \emptyset$ iff:

- There is no $\langle \odot, i, j\rangle \in B_k$, as this would mean that k retrieves at least *two* data values. Note that any state k' in $D_i^{\odot}(j)$ retrieves more than one data value (otherwise the datum would be described in $D_i^{=}(\chi(k'))$).
- Any pair of data values that may be reached with state k at the root must be equal: for all $a, b \in B_k$, $a \equiv_E b$.
- $D_0^{=}(\chi(k))$ consists of every state from the descriptions of B_k, or from any other description that is declared to be equal in $\equiv_E$. $D_0^{=}(\chi(k)) = \{k' \mid a \in B_k,\ a \equiv_E \langle \alpha, i, j\rangle,\ k'' \in D_i^{\alpha}(j),\ \mathsf{step\text{-}up}(k'', k')\}$.

With respect to $D_0^{\odot}$ we simply need to state that it cannot contain elements already in $D_0^{=}$. This completes the definition of τ. It is easy to see that checking all the preceding conditions consumes at most an exponential amount of time and hence that $\mathcal{A}_M$ can be built in ExpTime.

We have completely described the tree automaton $\mathcal{A}_M$. As it contains an exponential amount of extended states, the emptiness problem for this automaton can be computed in ExpTime. We have that M is empty iff $\mathcal{A}_M$ is empty. $\square$

Proposition 1. *(Soundness) For every accepting run of $\mathcal{A}_M$ on $\mathcal{T}$ there is an accepting run of M on some $\mathcal{T}'$.*

Proof. It is easy to show by induction that for any run of $\mathcal{A}_M$ on $\mathcal{T}$ we can define data values for all the nodes of $\mathcal{T}$ such that the data tree defined corresponds to an *abstraction* of the extended states in the run. The way of merging the data values in the inductive step is completely described by the relation $\equiv_E$. $\square$

Proposition 2. *(Completeness) For every accepting run of M on $\mathcal{T}$ there is an accepting run of $\mathcal{A}_M$ on some $\mathcal{T}'$.*

Proof. The proof can be sketched as follows. We first show that BIP automata are closed under subtree duplication. More concretely, the data tree $\langle a, d\rangle(t_1, t_2)$ (where $\langle a, d\rangle$ is the root and t_1, t_2 its two immediate subtrees) is accepted by a BIP M iff $\langle a, d\rangle(t_1, t_2, t_2)$ is accepted by M.

We then show a bounded-branching model property:

1. For each atomic formula $\exists(k_1,k_2)^{\neq}$ that holds at a node z, we mark (at most) two of their immediate subtrees that 'witness' this fact. It could be that we only need to mark one or none, if one of the two components k_1 or k_2 is directly witnessed at z. Each marking consists in a label that indicates a state and data value necessary to witness the formula, for example '(k', d')'. We proceed similarly for $\exists(k_1,k_2)^{=}$.
2. Moreover, this can be done in such a way that we don't mark twice the same subtree. We can ensure this by duplicating sibling subtrees if necessary.
3. In this way, we have that each node marks at most some bounded number of subtrees (say N), and at the same time it may be marked by its father. It is easy to see that N is polynomially bounded by the number of states $|K|$. Given a marking of a node z (coming from its father) it could be that (1) the marking is 'witnessed' at z, or (2) that it actually needs a subtree. In the case of (2) we add the marking to the corresponding subtree, always making sure that no subtree is marked twice.
4. We then have that each node has marked (at most) $N+1$ immediate subtrees. The rest of the subtrees that are not marked can be safely removed from the tree. This can be done with a top-down algorithm, where the root is the only node to mark at most N nodes (as it has no father).

Finally, we associate an extended state of $\mathcal{A}_M$ to each node. Consider the marking just explained. For each node z we build the extended state, where $D^{=}$ is completely determined by the tree $\mathcal{T}|_z$, and we use $D^{\odot}$ to ensure that for each of the markings (k', d') generated by z, d' is described. There are at most $N+1$ such markings, and we especially choose a sufficiently large size of $D^{\odot}$ (i.e., of t_0) to be able to accommodate all of them. We can then check that this is indeed a correct accepting run of $\mathcal{A}_M$. $\square$

Corollary 1. *SAT-$\mathsf{regXPath}(\downarrow, =)$ is in* ExpTime.

Proof. A direct consequence of Theorems 3 and 4. $\square$

In the presence of document type definitions.

The BIP automaton can be extended to have transitions that may demand conditions on the states of the child nodes stating, for example, that a node can be labeled by state q_1 if it has a child tagged with state q_2. It is actually easy to see that this can be simulated using the pathfinder automaton. Furthermore, consider the extension of its formulæ by *positive occurrences* of $\#q \geq n$ where $n \in \mathbb{N}$ is a constant, with the intended meaning that it is verified whenever there are *at least* n child nodes labeled with state q. Similarly, $\#q = 0$ states that there are no children with state q, but formulæ $\#q \leq n$ are not allowed.

It can be checked that a similar emptiness algorithm can be applied, the only difference being that the maximum branching width of the algorithm depends on the greatest constant n_0 used in the definition of the automaton. We obtain then an algorithm of time exponential in n_0.

So, in the case where n_0 is fixed, or where we consider constraints $\#q \geq n$ with n encoded with a unary representation, we still have an ExpTime algorithm for emptiness.

Consider the document types definable with a tree automaton on unranked trees with this 'zero/many' counting ability, where at each transition we can check either that there is no child with a certain state, that there are at least n with a certain state for some n, or conjunctions and disjunctions of these kind of conditions. It is possible thus to check the satisfiability of any $\mathsf{regXPath}(\downarrow, =)$ under these document types in ExpTime.

To verify this, we should mention that intersection of two BIP automata M_1 and M_2 can be simply obtained by a BIP with the $Q_{M_1} \times Q_{M_1}$ an defining $\mu_{M_1 \cap M_2}(q_1, q_2) = \mu_{M_1}(q_1) \wedge \mu_{M_2}(q_2)$. Observe that all positive occurrences of the $\#q \geq n$ formulæ remain positive.

Inclusion and equivalence problems.

We can finally mention that as $\mathsf{regXPath}(\downarrow, =)$ is closed under negation and boolean operations, we also get a decision procedure for the equivalence and the inclusion problems for *node expressions* ($\varphi \subset \psi$ iff $\varphi \wedge \neg\psi$ is not satisfiable). However, we cannot solve the problems of *path expressions* inclusion or equivalence with this kind of automata.

4.2 Lower bound

In this section we prove EXPTIME-hardness of satisfiability of $\mathsf{XPath}(\downarrow^*, =)$. Remarkably, this logic cannot express a *one step* down in the tree as it does not possess the $\downarrow$ axis, and this will be the major obstacle in the coding.

THEOREM 5. *SAT-*$\mathsf{XPath}(\downarrow^*, =)$ *is* EXPTIME*-hard.*

PROOF. The proof is by reduction from the *two-player corridor tiling game*. An instance of this game consists in a size of the corridor n (encoded in unary), a set of tiles $T = \{T_1, \ldots, T_s\}$, a special winning tile T_s, the set of initial tiles $\{T_1^0 \ldots T_n^0\}$, and the horizontal and vertical tiling relations $H, V \subseteq T \times T$. The game is played in an $n \times \mathbb{N}$ board where the initial configuration of the first row is given by $T_1^0 \ldots T_n^0$. At any moment during the play any pair of horizontal consecutive tiles must be in the relation H and every pair of vertical consecutive tiles in the relation V. The game is played by two players: Abelard and Eloise. Each player takes turn in placing a tile of his choice, filling the board from left to right, from bottom to top, always respecting the horizontal and vertical constraints H and V. Eloise is the first to move, and she wins iff during the play the winning tile T_s is placed on the board. If the play ends without this configuration being reached, or if it runs infinitely, the play is won by Abelard. It is known that deciding whether Eloise has a winning strategy is EXPTIME-complete. For more details on this game we refer the reader to [3].

Representation of a winning strategy.

It is easy to see that in this game Eloise has a winning strategy iff she has a strategy to win before the row s^n of the board is reached (where s is the number of tiles). Then each play between Eloise and Abelard can be coded as a succession of at most s^n rows of n tiles each. Wlog we assume that n is an even number, and hence all odd positions are played by Eloise, while even ones by Abelard. We can then represent a winning strategy as a tree, where at each even position there exists one branch for every possible move of Abelard and where all branches of the tree contain the winning tile T_s.

We must now come up with a way to encode all possible winning strategies of Eloise in $\mathsf{XPath}(\downarrow^*, =)$. As usual, a strategy of Eloise will be coded as a tree with a different branch for every possible move of Abelard.

Our alphabet consists in the symbols $I_1 \ldots I_n$ that indicate the current column of the corridor, the symbols $b_0 \ldots b_m$ where $m = \lceil (n+1).log(s) \rceil$ that act as *bits* to count from 0 to s^n (it is enough that they count *at least* up to s^n), and the symbols $T_1 \ldots T_s$ to code the tile placed at each move. The coding makes use of a symbol # to separate rows, and an extra symbol \$ whose role will be explained later.

Each block of nodes between two consecutive # codes the evolution of the play for a particular row. Each node labeled I_i has a tile associated, coded as a descendant node T_j with the same data value. In Fig. 3 the first column I_1 of the current row is associated to the tile T_3, because $\langle T_3, 1\rangle$ is a descendant of $\langle I_1, 1\rangle$ with the same data value. Similarly, each occurrence of # is associated to a number, coded by the b_i elements. In the example, $\langle \#, 0\rangle$ is associated to the bits b_0 and b_2 that give the binary number 101.

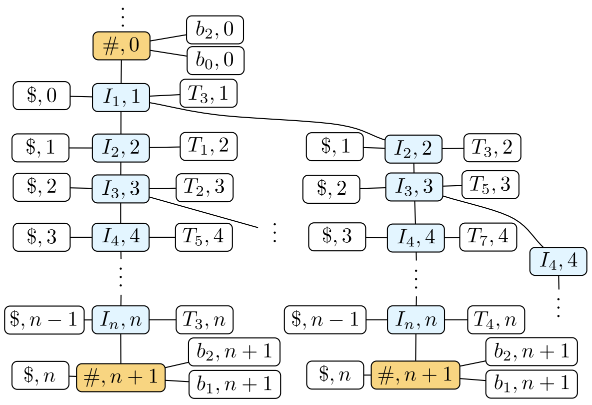
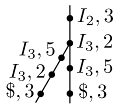


Figure 3: Part of the model coding all the moves of row 5, which is between the #-element associated to 5 (101 in binary), and the element # with number 6 (110).

Finally, the symbol \$ is used to delimit the region where the next element of the coding must appear, this will be our way of perceiving the *next step* of the coding. Intuitively, between I_i and the \$ with the same data value, only a I_{i+1} may appear. This mechanism of coding a very relaxed 'one step' is the building block of our coding. As the logic lacks the $\downarrow$ axis, we need to restrict the appearance of the next move of the play to a limited fragment of the model. By means of this element \$, we can state, for example, that whenever we are in a I_2 element, then in this restricted portion I_3 must be true by stating $\varepsilon =\downarrow^* [I_3] \downarrow^* [\$]$. In a similar way we can demand that *all* elements verify I_3 (except, perhaps, a prefix of I_2 elements).



However, we cannot avoid having more than one element before the \$ as shown in the figure. We may have 'repeated' elements or extra branches, but this does not spoil the coding.

We are actually able to force properties for *all* branches and all possible extra elements that the tree may contain. Intuitively, any extra element or branching induces more copies of winning strategies for Eloise.

In Fig. 3 we show an example of a possible extract of the tree between the # associated to the counting of 5 until the next # of counting 6. The coding forces a branching as it contains all possible answers of Abelard at even positions.

Building up the coding.

Let us define some useful predicates. $\mathsf{s}_\sigma^k(\varphi)$ evaluates φ at a node at k-steps (with our way of coding a step as we have seen before) from the current point of evaluation, given that the current symbol is σ. For this purpose we first define $next(I_i) := I_{i+1}$ (if $i < n$), $next(I_n) := \#$ and $next(\#) := I_1$. Hence, for $a \in \{\#, I_1, \ldots, I_n\}$,

$$\mathsf{s}_a^0(\varphi) := a \wedge \varphi \quad \mathsf{s}_a^{k+1}(\varphi) := a \wedge \varepsilon =\downarrow^* [\mathsf{s}_{next(a)}^k(\varphi)] \downarrow^* [\$]$$

Similarly, t_j checks that the tile of the current node I corresponds to T_j, bit_i checks that the i-bit of the counter's binary encoding of a #-node is one (1), and G forces a property to hold at all nodes of the tree.

$$\mathsf{t}_i := \varepsilon =\downarrow^* [T_i] \quad \mathsf{G}(\varphi) := \neg\langle\downarrow^* [\neg\varphi]\rangle \quad bit_i := \varepsilon =\downarrow^* [b_i]$$

We now describe all the conditions to force the aforementioned encoding. We also exhibit the XPath formula counterparts of the non-trivial conditions.

1. Every I_i, T_i, and $\#$ along the tree has different data value. As we have only the transitive closure axis, we actually express that whenever there are two elements with the same symbol a such that there is a third element in the middle with another symbol different from a (and then they can be effectively distinguished), they must have different data value. Actually, the fact that there could be a sequence of elements with equal label does not cause any problem. Let us see the case for I_i: $\neg \downarrow^* [I_i \wedge \varepsilon = \downarrow^* [\neg I_i] \downarrow^* [I_i]]$.

2. Every I_i has a next element, unless it contains the winning tile, $\mathsf{G}(I_i \wedge \neg \mathsf{t}_s \to \mathsf{s}^1_{I_i}(\top)) \wedge \mathsf{G}(\# \to \mathsf{s}^1_{\#}(\top))$.

3. \$ are leaves, in the sense that no other symbol may appear as descendant: $\neg\langle \downarrow^* [\$ \wedge \langle \downarrow^* [\neg \$]\rangle]\rangle$.

4. Every I_i has its corresponding \$: $\mathsf{G}(I_i \to \varepsilon = \downarrow^* [\$])$.

5. Each I_i has a unique tile: $\mathsf{G}(\neg(\mathsf{t}_\ell \wedge \mathsf{t}_j))$ for $\ell \neq j$.

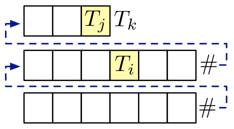
6. All I_{i+1} inside a step along a branch must have the same tile. (And a similar condition for I_1.) That is, for every $i < n$ and $j \neq k$, $\mathsf{G}(I_i \to \neg\varepsilon = \downarrow^* [I_{i+1} \wedge \mathsf{t}_j] \downarrow^* [I_{i+1} \wedge \mathsf{t}_k] \downarrow^* [\$])$.

7. Between I_i, $i < n$ and its corresponding \$ only I_{i+1} may appear. (And also for I_n and $\#$, and for $\#$ and I_1.) That is, for any $i < n$ and $j \notin \{i, i+1\}$, $\mathsf{G}(I_i \to \neg\varepsilon = \downarrow^* [I_j] \downarrow^* [\$])$, and $\mathsf{G}(I_i \to \neg\varepsilon = \downarrow^* [I_{i+1}] \downarrow^* [I_i] \downarrow^* [\$])$.

8. The tiles match horizontally: for every k and T_i, T_j such that $\neg H(T_i, T_j)$, $\neg\langle \downarrow^* [I_k \wedge \mathsf{t}_i \wedge \mathsf{s}^1_{I_k}(\mathsf{t}_j)]\rangle$. Moreover, the tiles match vertically, for every k and T_i, T_j such that $\neg V(T_i, T_j)$, $\neg\langle \downarrow^* [I_k \wedge \mathsf{t}_i \wedge \mathsf{s}^{n+1}_{I_k}(\mathsf{t}_j)]\rangle$.

9. All the elements corresponding to the *first* row match with $T^0_1 \dots T^0_n$. That is, for all $i \in [1..n]$ and tile T_j such that $\neg V(T^0_i, T_j)$, then $\neg \mathsf{s}^i_{\#}(\mathsf{t}_j)$ must hold at the root.

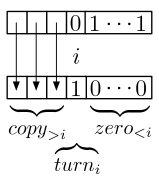
10. All possible moves of Abelard are taken into account. For every triple of tiles T_i, T_j, T_k such that $H(T_j, T_k), V(T_i, T_k)$, each time Abelard *can* play T_k, he *must* play it.



$$\neg\left\langle \downarrow^* \left[I_{2\ell} \wedge \mathsf{t}_i \wedge \mathsf{s}^n_{I_{2\ell}}\left(I_{2\ell-1} \wedge \mathsf{t}_j \wedge \neg \mathsf{s}^1_{I_{2\ell-1}}(\mathsf{t}_k)\right)\right)\right\rangle$$

11. There is no $\#$ element that has all the b_i bits in 1. Because that would mean that Eloise was not able to put a T_s tile in less than s^n rounds.

12. The data value of a $\#$ element is associated to a counter. It is easy to code that the first $\#$ is all-zero. The increment of the counter between two $\#$ is coded as $\mathsf{G}(\# \wedge \mathit{flip}(i) \to \mathit{zero}_{<i} \wedge \mathit{turn}_i \wedge \mathit{copy}_{>i})$, where



$$\begin{aligned}
\mathit{flip}(i) &= \neg \mathit{bit}_i \wedge \bigwedge_{j<i} \mathit{bit}_j \\
\mathit{zero}_{<i} &= \bigwedge_{j<i} \neg \mathsf{s}^{n+1}_{\#}(\mathit{bit}_j) \\
\mathit{turn}_i &= \neg \mathsf{s}^{n+1}_{\#}(\neg \mathit{bit}_i) \\
\mathit{copy}_{>i} &= \bigwedge_{j>i} (\mathit{bit}_j \wedge \neg \mathsf{s}^{n+1}_{\#}(\neg \mathit{bit}_j)) \vee \\
&\quad (\neg \mathit{bit}_j \wedge \neg \mathsf{s}^{n+1}_{\#}(\mathit{bit}_j))
\end{aligned}$$

This completes the coding. It is easy to see that each one of the formulæ described has a polynomial length on s and n. It can be shown then that Eloise has a winning strategy in the two-player corridor tiling game iff the conjunction of the formulæ just described is satisfiable. As none of the above formulæ use unions of path expressions, the lower bound holds also in the absence of such expressions. □

5. PSPACE FRAGMENTS

We now turn to some other downward fragments of XPath. We complete the picture of the complexity for all possible combinations of downward axis in the presence and in the absence of data values. We first need to introduce a basic definition that we use throughout the section.

Definition 2. We say that the logic $\mathcal{L}$ has the *poly-depth model property* if there exists a polynomial f such that for every formula $\varphi \in \mathcal{L}$, φ is satisfiable iff φ is satisfiable in a data tree model of depth at most $f(|\varphi|)$.

We can now prove the following statement that we will use to show PSPACE-completeness for $\mathsf{XPath}(\downarrow, =)$.

THEOREM 6. *Every fragment $\mathcal{L}$ of $\mathsf{regXPath}(\downarrow, =)$ with the poly-depth model property is in* PSPACE.

PROOF. Suppose that if a formula $\eta \in \mathcal{L}$ is satisfiable in a model, then it is satisfiable in a model of height h with $h \leq f(|\eta|)$ where f is a polynomial.

We can then translate η into a BIP automaton M. We show next how to modify the emptiness algorithm to make it work in non-deterministic polynomial space by means of f. We define an algorithm by recursion on the height of the tree h. The algorithm receives *three* parameters: (1) a BIP automaton M, (2) an extended state c_0, and (3) h, the maximum height of the tree to reach the extended state. The algorithm must verify that c_0 can be reached in a tree of height at most h in non-deterministic polynomial space in h. Now the emptiness algorithm for $\mathcal{A}_M$ must be done *on the fly*, that is, we do not build the set of all possible extended states. The base case is when $h = 1$. In this case we can easily check the existence of a singleton tree that satisfies c_0 in polynomial space.

Suppose now the height is $h = n + 1$. The algorithm guesses $c_1 \dots c_u$ extended states corresponding to the immediate subtrees and verifies that c_0 and $c_1 \dots c_u$ are in a transition of $\mathcal{A}_M$. To do this, we guess a relation $\equiv_E$ and test that all the conditions described in the construction of $\mathcal{A}_M$ seen before are satisfied. Finally, by inductive hypothesis we can check that each one of $c_1 \dots c_u$ extended states are satisfied in a model of depth at most n, and this test can be done in space polynomial in n. In terms of space complexity this algorithm uses (a) the space to store $c_1 \dots c_u$ where u is a polynomial in M and the space required to store each c_i is polynomial in M, (b) the space to store the relation $\equiv_E$ to check their correctness, that can be bounded by $2.u.(2|K|^2 + 3|K| + 2)$ and also remains polynomial in M, and (c) some polynomially bounded space on n (call it $S(n)$) to do the u recursive calls. Then the space required is $S(n+1) = \text{(a)} + \text{(b)} + S(n)$. It is then immediate that the algorithm is in NPSPACE.

The main algorithm can be sketched as follows. Given η, we compile η into M in PTIME, we guess an extended state c_0 that contains a final state of M and we check the guessing is correct by calling the algorithm just described. Thus, as NPSPACE = PSPACE the theorem follows. □

PROPOSITION 3. *SAT-$\mathsf{XPath}(\downarrow, =)$ is* PSPACE*-complete.*

PROOF. $\mathsf{XPath}(\downarrow)$ is shown to be PSPACE-hard in [1]. For the upper bound, we show the poly-depth model property. It is easy to show that if η is satisfiable in $\mathcal{T}$, then it is satisfiable in $\mathcal{T} \upharpoonright n$ where n is the maximum quantity of nested $\downarrow$ of the formula, and $\mathcal{T} \restriction n$ is the submodel of $\mathcal{T}$ consisting of all the nodes that are at distance at most n from the root. Hence, by Theorem 6, XPath($\downarrow$) is in PSpace. □

So far we have that, in the presence of data values, the ability to have the descendant axis ($\downarrow^*$) produces an increase in the complexity from PSpace to ExpTime[1]. However, we argue that it is not the ability to test for data equality of distant elements what produces this raise in complexity. It is, as a matter of fact, in the ability to test data values against that of the root in formulæ like $\varepsilon = \downarrow^* [a]$. We show that if we actually eliminate this kind of data tests, we can prove the resulting logic to be only in PSpace.

Definition 3. We denote by XPath($\downarrow^*, =$)$\setminus\varepsilon$ the fragment of XPath($\downarrow^*, =$) where the ε path formulæ are forbidden, and in general where there are no ε-testing in a path (like in $[\varphi] \downarrow^*$), $\alpha ::= \downarrow^* \mid \alpha[\varphi] \mid \alpha\beta \mid \alpha \cup \beta$.

Proposition 4. *SAT-XPath($\downarrow^*, =$)$\setminus\varepsilon$ is PSpace-complete.*

Proof (sketch). The proof is done by proving the poly-depth model property. The key observation is that any XPath($\downarrow^*, =$)$\setminus\varepsilon$ path expression that is satisfied at a node n of a tree, is also satisfied in any ancestor of n, this is basically because all path expressions start with a $\downarrow^*$ axis. This means that for any pair of nodes n, n' such that n is an ancestor of n', the set of formulæ of the type $\langle p \rangle$, $p = p'$ or $p \neq p'$ (with p, p' path expressions) that are satisfied in n' is a *subset* of those that are satisfied in n. Thus, if φ is a formula satisfied in $\mathcal{T}$, for a given branch there is only a polynomial number of configurations of the (sub)paths in φ verified in each of its nodes. Long branches with a repeated description can actually be pruned into a shorter branch, preserving satisfiability of φ in $\mathcal{T}$. □

Proposition 5. *SAT-XPath($\downarrow^*$) is PSpace-complete.*

Proof (sketch). The lower bound is shown by coding the QBF problem. The upper bound is shown via the poly-depth model property. It is slightly involved and requires to show a normal form of the model with the following property. If for some node both $\langle\alpha\rangle$ and $\langle\beta\rangle$ hold, then α and β are witnessed by two *disjoint* branches of polynomial depth. □

6. CONCLUDING REMARKS

We have shown the complexity of various downward fragments of XPath. The highest complexity class we obtained is ExpTime. In the presence of data equality tests, this is a well behaved fragment considering that in the presence of all the axes XPath is undecidable. One important reason for this, is the absence of sibling axis. Actually, in the presence of arbitrary DTDs we can show that the satisfiability problem of the downward fragment is either undecidable, or decidable with a non-primitive recursive algorithm. We have shown however that we can evaluate some restricted fragment of DTDs that cannot express sibling order nor limit the quantity of occurrences of nodes of a certain type, but that can demand, for example, that any a has at least five b children and no c child. By solving the satisfiability problem we are also able to solve the inclusion and equivalence problems of node expressions for free. We leave open the question of whether the inclusion of path expressions (as binary relations) is also decidable in ExpTime.

[1]In the case PSpace $\neq$ ExpTime.

$\downarrow$	$\downarrow^*$	$=$	**Complexity**	**Details**
+			PSpace-complete	Prop 3
	+		PSpace-complete	Prop 5
+	+		ExpTime-complete	[1]
+		+	PSpace-complete	Prop 3 and [1]
	+	+	ExpTime-complete	Cor 1, Th 5
+	+	+	ExpTime-complete	Cor 1, Th 5
regXPath($\downarrow, =$)			ExpTime-complete	Cor 1, Th 5
XPath($\downarrow^*, =$) $\setminus\varepsilon$			PSpace-complete	Prop 4

All the results hold also in the absence of path unions.

Figure 4: Summary of results.

We introduced the new class of bip automata that capture all the expressivity of regXPath($\downarrow, =$). By the proof of decidability, we conclude that there is a very strong normal form of the model for this logic. If a formula $\eta \in$ regXPath($\downarrow, =$) is satisfiable, then it is satisfiable in a model of exponential height and polynomial branching width, whose data values are such that only a polynomial number of data values can be shared between any two disjoint subtrees. This small model property is reflected by the fact that the emptiness of the automaton only depends on a polynomial number of data values at every point of a branch. However, there is no syntactic restriction in the automaton, it can retrieve and compare any number of data values between them and the root's data value at each step of the execution.

Acknowledgments. I am grateful to Luc Segoufin and Stéphane Demri for helpful discussions and for critically reading this manuscript.

7. REFERENCES

[1] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *J.ACM*, 55(2), 2008.

[2] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data trees and XML reasoning. In *PODS*, pages 10–19. ACM, 2006.

[3] B. S. Chlebus. Domino-tiling games. *J. Comput. Syst. Sci.*, 32(3):374–392, 1986.

[4] J. Clark and S. DeRose. XML path language (XPath). Website, November 1999. W3C Recommendation. `http://www.w3.org/TR/xpath`.

[5] F. Geerts and W. Fan. Satisfiability of XPath queries with sibling axes. In *DBPL*, volume 3774, pages 122–137. Springer, 2005.

[6] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[7] M. Jurdziński and R. Lazić. Alternating automata on data trees and XPath satisfiability. *CoRR*, abs/0805.0330, 2008.

[8] M. Marx. First order paths in ordered trees. In *ICDT*, volume 3363, pages 114–128. Springer, 2005.

[9] B. ten Cate. The expressivity of XPath with transitive closure. In *PODS*, pages 328–337. ACM Press, 2006.

[10] B. ten Cate and L. Segoufin. XPath, transitive closure logic, and nested tree walking automata. In *PODS*, pages 251–260. ACM Press, 2008.