XPath, Transitive Closure Logic, and Nested Tree Walking Automata*

Balder ten Cate
ISLA, Universiteit van Amsterdam
http://staff.science.uva.nl/~bcate

Luc Segoufin
INRIA, LSV
http://www-rocq.inria.fr/~segoufin

ABSTRACT

We consider the navigational core of XPath, extended with two operators: the Kleene star for taking the transitive closure of path expressions, and a subtree relativisation operator, allowing one to restrict attention to a specific subtree while evaluating a subexpression. We show that the expressive power of this XPath dialect equals that of FO(MTC), first order logic extended with monadic transitive closure. We also give a characterization in terms of nested tree-walking automata. Using the latter we then proceed to show that the language is strictly less expressive than MSO. This solves an open question about the relative expressive power of FO(MTC) and MSO on trees. We also investigate the complexity for our XPath dialect. We show that query evaluation be done in polynomial time (combined complexity), but that satisfiability and query containment (as well as emptiness for our automaton model) are 2ExpTime-complete (it is ExpTime-complete for Core XPath).

Categories and Subject Descriptors
H.2.3 [Database Management]: Languages

General Terms
Languages, Algorithms

Keywords
XML, XPath, Transitive Closure, Tree Walking Automata

1. INTRODUCTION

In the semi-structured model in general, and XML in particular, data is extracted, queried, or used not only based on its content, but also based on its position inside the document tree. Hence, tools for processing XML combine tree navigation with other, more classical database functionalities such as data value joins. The basic formalism used for XML tree navigation is *XPath*. It is a simple and easy-to-use language, and it lies at the core of the XML processing languages XQuery and XSLT.

*We would like to thank Mikołaj Bojańczyk and Hendrik-Jan Hoogeboom, as well as the anonymous reviewers, for their comments. The first author is supported by the Netherlands Organization for Scientific Research (NWO) grant 639.021.508, and partially by NSF grant IIS-0430994.

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'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-108-8/08/06 ...$5.00.

The logical core of XPath 1.0, *Core XPath* [10], has been studied in detail. For instance, the combined complexity of query evaluation is known to be in PTime [10], static analysis tasks such as satisfiability are ExpTime-complete [2], and the expressive power of Core XPath has been characterized in terms of FO^2 [14], the two-variable fragment of first-order logic on trees.

When studying the expressive power of XML query languages, two natural "golden standards" of expressive power are commonly considered: *first order logic* (FO) and *monadic second order logic* (MSO). The former is the traditional yardstick for expressive power of relational query languages, while the latter is a very well-behaved and well-understood logic on trees. Several proposals have been made for extending Core XPath so that it becomes expressively complete for either FO or MSO, while keeping all the nice properties.

Core XPath 2.0, the navigational core of XPath 2.0 [20], extends Core XPath with path intersection and complementation, as well as quantified variables. It has the same expressive power as FO, but unfortunately also the same complexity: query evaluation is PSpace-complete (combined complexity), and satisfiability is non-elementary. *Conditional XPath* [13], extending Core XPath with "until" operators, does better: it has the same expressive power as FO, but the complexity of query evaluation and satisfiability is no higher than in the case of Core XPath (up to a polynomial).

Likewise, extensions of Core XPath have been proposed that are expressively complete for MSO, for instance with least fixed point operators (see, e.g., [18, Sect. 4.2]). However, none of the proposals have the intuitive clarity and simplicity of Core XPath. In particular, expressions containing fixed point operators are notoriously difficult to work with.

A more modest way to add recursion to XPath is by means of the Kleene star, which allows one to take the transitive closure of a path expression. This extension has been advocated for a variety of reasons, ranging from practical (e.g., [15]), to more theoretical. For instance, the transitive closure operator enables us to express DTDs and other schema constraints (e.g., ancestor based patterns) directly inside XPath. It also enables view based query rewriting for recursive views [9]. The extension of Core XPath with the transitive closure operator is called *Regular XPath*.

The starting point of this paper is the question whether Regular XPath is expressively complete for MSO. We show that this is not the case. In fact, we show more. We consider a subtree relativisation operator W, enabling one to evaluate expressions in restricted parts of a document. We denote by Regular XPath(W) the extension of Core XPath with transitive closure and W. We characterize the expressive power of Regular XPath(W) in terms of FO(MTC), first-order logic extended the monadic transitive closure operator:

THEOREM 1. *Regular XPath(*W*) has the same expressive power as* FO(MTC).

More precisely, we show that Regular XPath(W) path expressions have the same expressive power as FO(MTC) formulas with two free variables, and Regular XPath(W) node expressions have the same expressive power as FO(MTC) formulas with one free variable.

THEOREM 2. FO(MTC) *is strictly less expressive than* MSO *on trees.*

This settles an open problem raised in [17, 7].

The W operator we consider is closely related to the XPath *subtree scoping* operator in [5], and the *subtree query composition* considered in [4].

The proof of Theorem 2 is based on an extension of *Tree Walking Automata* (TWA), which we call *nested TWA*. Roughly speaking, these extend ordinary TWA in that the transitions of an automaton may depend on whether or not a subautomaton has an accepting run in the subtree rooted at the current node. We show that nested TWA have the same expressive power as FO(MTC) and Regular XPath(W). We use this automata theoretic characterization in order to show that FO(MTC), and hence Regular XPath(W), is strictly less expressive than MSO on trees.

We also study the complexity of Regular XPath(W). We show that query evaluation for Regular XPath(W) remains in PTime. Satisfiability, on the other hand, is 2ExpTime-complete (compared to ExpTime-complete for Core XPath and Conditional XPath).

All in all, we believe Regular XPath(W) achieves a nice trade off between expressivity, simplicity, and complexity. Our results shows that the regular tree languages definable in FO(MTC) form a robust class that can be characterized in several ways. We believe FO(MTC) should be considered another yardstick of expressive power on trees, in between FO and MSO.

Related work.

Connections between transitive closure logic and TWA have already been observed in [7]. The automata model used in [7] is different than the one we use: it extends TWA with nested pebbles marking positions in the tree. Transitions then depend on the presence or absence of pebbles in the current node. The main difference between pebble TWA and our nested TWA is that the latter can make negative tests. It is still an open problem whether pebble TWA are closed under complementation, while the closure under complementation of nested TWA is immediate from the definition. Also, emptiness of pebble TWA is non-elementary while it is 2ExpTime-complete for our nested TWA model. It is shown in [7] that pebble TWA have exactly the same power as FO(pos-MTC), the fragment of FO(MTC) where the TC operator can only be used in the scope of a even number of negations. Pebble TWA, and hence FO(pos-MTC), were shown to be strictly contained in MSO in [6]. Our proof of Theorem 2 uses the same separating language, and the same proof technique as in [6].

In [18], another variant of Regular XPath was considered, containing *path equalities* instead of W. This language was shown to have the same expressive power as FO^*, the parameter free fragment of FO(MTC). It is not known at present whether FO^* is as expressive as full FO(MTC).

The W operator we use is a generalization to trees of the "now" operator from temporal logics with forgettable past [12]. It is closely related to the "within" operator in temporal logics for nested words [1], the XPath "subtree scoping" operator proposed in [5], and the issue of *subtree query composition* studied in [4]. See Section 8 for more details on the connection.

Organization of the paper.

In Section 2, we introduce Regular XPath(W), FO(MTC), and nested TWA. In Section 3 and 4, we prove Theorem 1. In Section 5, we prove Theorem 2. In Section 6, we determine the complexity of query evaluation and satisfiability for Regular XPath(W). In all cases, attention is restricted to binary trees. In Section 7, however, we show that all results generalize to unranked trees. Finally, we conclude in Section 8 by deriving some further consequences of our results.

2. PRELIMINARIES

2.1 Trees

In this paper we consider two kinds of trees: finite unranked ordered trees and, as a special case, binary trees.

Fix a finite alphabet Σ. A Σ-tree is a partial mapping $t : \mathbb{N}^* \to \Sigma$ whose domain $dom(t)$ is finite, non-empty, prefix-closed, and such that whenever $n(i+1) \in dom(t)$ then also $ni \in dom(t)$. Elements of this domain are called *nodes*, and the *label* of a node x is $t(x)$. The empty sequence ε is called the *root* of t, and all maximal sequences are called *leafs*. For $x, y \in dom(t)$, we write $x < y$ if $y = xz$ for some non-empty $z \in \mathbb{N}^*$ (i.e., if x is an ancestor of y in the tree), and we write $x \prec y$ if $x = zi$ and $y = zj$ for some $z \in \mathbb{N}^*$ and natural numbers $i < j$ (i.e., y is a sibling to the right of x). For any $x, y \in dom(t)$, $lca(x, y)$ denotes the least common ancestor of x and y, i.e., the largest common prefix. Given a node x in a tree t, we denote the subtree of t rooted at x by $\mathsf{subtree}(t, x)$.

A Σ-tree is called a *binary tree* if each non-leaf node has exactly two successors. In most of our proofs, we will restrict attention to binary trees, but we will show in Section 7 that this is without loss of generality. When speaking of binary trees, we will use $<_1$ and $<_2$ for the "descendant along the first child" and "descendant along the second child relation". Formally, for any two nodes x, y and for $i \in \{1, 2\}$, $x <_i y$ holds if xi is a prefix of y, i.e., if y is a not-necessarily-strict descendant of x's ith child.

A *tree language* is a set of Σ-trees, for a given Σ. A tree language is *regular* if it is definable by a sentence of monadic second-order logic (MSO) in the signature with binary relations $<, \prec$ and a unary relation for each element of Σ.

2.2 Regular XPath(W)

As we already mentioned, Regular XPath(W) essentially extends Core XPath with a transitive closure operator and a subtree relativisation operator. We now define it formally.

DEFINITION 3. *Regular XPath(*W*) is a two-sorted language, with* path expressions *($\alpha, \beta, \ldots$) and* node expressions *($\phi, \psi, \ldots$). These are defined by mutual recursion as follows:*

$$\alpha ::= \downarrow \mid \uparrow \mid \rightarrow \mid \leftarrow \mid . \mid \alpha/\beta \mid \alpha \cup \beta \mid \alpha[\phi] \mid \alpha^*$$
$$\phi ::= \sigma \mid \langle\alpha\rangle \mid \neg\phi \mid \phi \wedge \psi \mid \mathsf{W}\phi \qquad (\sigma \in \Sigma)$$

Given a tree t, each path expression α defines a binary relation $[\![\alpha]\!]^t$ and each node expression ϕ defines a set of nodes $[\![\phi]\!]^t$. The exact semantics is specified in Table 1.

A path expression essentially describes a movement in a tree. It will sometimes be useful to perform the movement in the reverse order. We denote by $^{-1}$ this operation. Given a path expression α, α^{-1} is the path expression defined by induction as follows: $(\alpha[\phi])^{-1}$ is $.[\phi]\alpha^{-1}$, $(\alpha/\beta)^{-1}$ is β^{-1}/α^{-1}, $(\alpha \cup \beta)^{-1}$ is $\alpha^{-1} \cup \beta^{-1}$, $(\alpha^*)^{-1}$ is $(\alpha^{-1})^*$, $.^{-1}$ is ., and for $\pi \in \{\downarrow, \uparrow, \leftarrow, \rightarrow\}$, π^{-1} is defined by reversing the direction of the arrow, for instance

Table 1: Semantics of Regular XPath(W) expressions

$$
\begin{array}{lcl}
[\![\downarrow]\!]^t & = & \{(x, xi) \mid xi \in dom(t)\} \\
[\![\uparrow]\!]^t & = & \{(xi, x) \mid xi \in dom(t)\} \\
[\![\leftarrow]\!]^t & = & \{(xi, x(i+1)) \mid x(i+1) \in dom(t)\} \\
[\![\rightarrow]\!]^t & = & \{(x(i+1), xi) \mid x(i+1) \in dom(t)\} \\
[\![.]\!]^t & = & \{(x, x) \mid x \in dom(t)\} \\
\\
[\![\alpha/\beta]\!]^t & = & \{(x, y) \in dom(t)^2 \mid \exists z \in dom(t). \\
 & & \quad ((x, z) \in [\![\alpha]\!]^t \text{ and } (z, y) \in [\![\beta]\!]^t)\} \\
[\![\alpha \cup \beta]\!]^t & = & [\![\alpha]\!]^t \cup [\![\beta]\!]^t \\
[\![\alpha[\phi]]\!]^t & = & \{(x, y) \in [\![\alpha]\!]^t \mid y \in [\![\phi]\!]^t\} \\
[\![\alpha^*]\!]^t & = & \text{the reflexive transitive closure of } [\![\alpha]\!]^t \\
\\
[\![\sigma]\!]^t & = & \{x \in dom(t) \mid t(x) = \sigma\} \\
[\![\langle\alpha\rangle]\!]^t & = & \{x \in dom(t) \mid \exists y \in dom(t).(x, y) \in [\![\alpha]\!]^t\} \\
[\![\neg\phi]\!]^t & = & dom(t) \setminus [\![\phi]\!]^t \\
[\![\phi \wedge \psi]\!]^t & = & [\![\phi]\!]^t \cap [\![\psi]\!]^t \\
[\![\mathsf{W}\phi]\!]^t & = & \{x \in dom(t) \mid x \in [\![\phi]\!]^{\mathsf{subtree}(t,x)}\}
\end{array}
$$

$\downarrow^{-1}$ is $\uparrow$. It is easy to see that, with this definition, $(x, y) \in [\![\alpha^{-1}]\!]^t$ iff $(y, x) \in [\![\alpha]\!]^t$.

We will use root as a shorthand for $\neg\langle\uparrow\rangle$.

2.3 Nested tree walking automata

For proving our lower bounds it will be convenient to represent Regular XPath(W) formulas using automata walking in the tree. We introduce here the model that we shall use.

Let $DIR = \{\downarrow, \uparrow, \rightarrow, \leftarrow, .\}$ be the set of basic moves in the tree, corresponding to "*go to a child*" and its converse, "*go to the next sibling*" and its converse, and "*don't move*". Given a node x in a tree, $DIR(x) \subseteq DIR$ denotes the set of possible moves from x (which always contains '.'). In other words, $DIR(x)$ characterizes what type of node x is (root, first child, ...). A *non-deterministic tree walking automaton* (0-TWA) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, and δ is the transition relation:

$$\delta \subseteq (Q \times \Sigma \times \wp(DIR)) \times (DIR \times Q)$$

where $\wp(\cdot)$ means *powerset*. If $(q, \sigma, D, m, q') \in \delta$, this means intuitively that when the automaton is in state q at a node x with label σ and $DIR(x) = D$, then it may choose to make the move m and go to state q'. We require that m always belongs to D.

An *accepting run* of A from a node x to a node y in a tree t is a sequence of pairs $(x_0, q_0), \ldots, (x_n, q_n)$, where $x_0 = x$, $x_n = y$, q_0 is the initial state, and $q_n \in F$, where each transition $(x_i, q_i), (x_{i+1}, q_{i+1})$ conforms to the transition relation, i.e., $(q_i, t(x_i), DIR(x_i), m, q_{i+1}) \in \delta$ for some $m \in DIR$ such that $(x_i, x_j) \in [\![m]\!]^t$. For any tree t, $A(t)$ is the binary relation containing all pairs (x, y) such that A has an accepting run from x to y. A tree t is accepted by an automaton A if $(\varepsilon, x) \in A(t)$ for some $x \in dom(t)$.

For $k > 0$, a *nested tree walking automaton of rank k* (k-TWA) is a tuple $(Q, \Sigma, (A_i)_{i \in I}, \delta, q_0, F)$ with $(A_i)_{i \in I}$ a finite set of nested tree walking automata of rank strictly less than k, where Q, Σ, q_0, F are as before, and

$$\delta \subseteq (Q \times \Sigma \times \wp(DIR) \times \wp(I) \times \wp(I)) \times (DIR \times Q)$$

Accepting runs are defined as before, except that the automaton can now take additional information into account when deciding its next move. The first $\wp(I)$ in the definition of δ stands for the set of exactly those automata A_i that have an accepting run starting in the current node n, and the second $\wp(I)$ stands for the set of exactly those automata A_i that have an accepting run in $\mathsf{subtree}(t, n)$ starting in the root, i.e., n.

The notion of an *accepting run* is then defined in the straightforward way, by induction on the rank of the automaton. The same notations are used for k-TWA as for 0-TWA. In particular, $A(t)$ is the binary relation containing all pairs of nodes (x, y) such that A has an accepting run from x to y, and a tree t is accepted by an automaton A if $(\varepsilon, x) \in A(t)$ for some $x \in dom(t)$.

The following lemma shows that Regular XPath(W) path expression and nested TWA essentially define the same objects. Its proof uses the classical translations between regular expressions and automata and is omitted in this abstract.

LEMMA 4. *For every nested TWA A there is a Regular XPath(W) path expression α, computable in exponential time from A, such that for all trees t, $A(t) = [\![\alpha]\!]^t$.*

Conversely, for every Regular XPath(W) path expression α there is a nested TWA A, computable in linear time from α, such that for all trees t, $A(t) = [\![\alpha]\!]^t$.

2.4 Transitive closure logic

The third formalism that we use for expressing properties on trees is FO(MTC). FO(MTC) is the language of first-order logic extended with a monadic (reflexive) transitive closure operator. Since we work only with Σ-trees, we assume a fixed signature, consisting of binary relations $<, \prec$ and a unary predicate letter for each element of Σ. We use the notation $[\mathsf{TC}_{xy}\ \phi]$ for the (reflexive) transitive closure operator:

$$[\mathsf{TC}_{xy}\ \phi](u, v) \equiv \forall X.(Xu \wedge \forall xy(Xx \wedge \phi \rightarrow Xy) \rightarrow Xv)$$

Note that ϕ may contain other free variables besides x and y. These will be called *parameters* of the transitive closure formula.

It is clear from the definition that FO(MTC) is a fragment of MSO.

To make life easier, we will use a normal form for FO(MTC)-formulas on binary trees. Consider the following restricted version of the transitive closure operator:

$$[\mathsf{TC}^{<}_{xy}\ \phi](u, v) \equiv u < v \wedge [\mathsf{TC}_{xy}(u \leq x, y \wedge v \not< x, y \wedge \phi)](u, v)$$

It expresses that $u < v$ and that there is a sequence $u = x_1, \ldots, x_n = v$ such that each x_i is below u but not strictly below v, and $\phi(x_i, x_{i+1})$ holds for all $i < n$. The idea is that u and v delimit a part of the tree to which the entire sequence belongs.

We denote by $\mathrm{FV}(\phi)$ the free variables of ϕ. FO(MTC$^<$) is the fragment of FO(MTC) in which transitive closure is only allowed in the restricted form of $[\mathsf{TC}^{<}_{xy}\ \phi](u, v)$ with $\mathrm{FV}(\phi) \subseteq \{x, y, u, v\}$.

LEMMA 5. *On binary trees, every* FO(MTC) *formula is equivalent to a* FO(MTC$^<$) *formula.*

The proof of Lemma 5 uses standard Ehrenfeucht-Fraïssé techniques and is omitted in this abstract. Note that this is only an intermediate result: stronger normal form theorems for FO(MTC) on arbitrary trees will follow from our main results, see Section 8.

Table 2 gives a linear translation from Regular XPath(W) to FO(MTC), thereby establishing the easy direction of Theorem 1. Note that, in this translation, the parameter z is used in order to remember the root of the subtree with respect to which the formula has to be evaluated. The translation can easily be modified to use only four variables.

Table 2: Translation from Regular XPath(W) to FO(MTC)

$$
\begin{array}{lcl}
TR_{u,v}(\alpha) & = & \exists z.(\forall z'.(z \leq z') \wedge TR^z_{u,v}(\alpha)) \\
TR_u(\phi) & = & \exists z.(\forall z'.(z \leq z') \wedge TR^z_u(\phi)) \\
 & & \\
TR^z_{u,v}(\delta) & = & (z \leq u, v) \wedge R_\delta(u,v) \text{ for } \delta \in \{\uparrow, \downarrow, \rightarrow, \leftarrow\} \\
TR^z_{u,v}(.) & = & (z \leq u, v) \wedge u = v \\
TR^z_{u,v}(\alpha/\beta) & = & \exists w.(z \leq w \wedge TR^z_{u,w}(\alpha) \wedge TR^z_{w,v}(\alpha)) \\
TR^z_{u,v}(\alpha \cup \beta) & = & TR^z_{u,v}(\alpha) \cup TR^z_{u,v}(\beta) \\
TR^z_{u,v}(\alpha[\phi]) & = & TR^z_{u,v}(\alpha) \wedge TR^z_v(\phi) \\
TR^z_{u,v}(\alpha^*) & = & [\mathsf{TC}_{x,y}\, TR^z_{x,y}(\alpha)](u,v) \\
 & & \\
TR^z_u(\sigma) & = & (z < u) \wedge P_\sigma(u) \\
TR^z_u(\langle\alpha\rangle) & = & (z < u) \wedge \exists v. TR^z_{u,v}(\alpha) \\
TR^z_u(\neg\phi) & = & \neg TR^z_u(\phi) \\
TR^z_u(\phi \wedge \psi) & = & TR^z_u(\phi) \wedge TR^z_u(\psi) \\
TR^z_u(\mathsf{W}\phi) & = & \exists z((z = u) \wedge TR_x u^z(\phi))
\end{array}
$$

LEMMA 6. *For every Regular XPath(W) path expression α and tree t, $[\![\alpha]\!]^t = \{(x,y) \mid TR_{x,y}(\alpha)$ holds on $t\}$. Similarly for node expressions.*

3. DEFINING K-ARY QUERIES ON BINARY TREES USING TREE PATTERNS

We aim to translate FO(MTC) formulas in at most two free variables to Regular XPath(W) expressions. In order to perform the translation inductively, however, we have to be able to handle formulas with more than two free variables. For this reason, we will use *tree patterns* as an intermediate formalism. Intuitively, a tree pattern over a set of variables $v_1, \ldots, v_n$ is a tree-shaped conjunctive query over $v_1, \ldots, v_n$, in which Regular XPath(W) expressions may be used as atomic relations. See for examples Figure 2 on page 7. We prove in Section 4 that every FO(MTC$^<$) formula is equivalent to a union of tree patterns over the same free variables. This suffices for the proof of Theorem 1, because, as we will show in the present section, tree patterns over at most two free variables are equivalent to Regular XPath(W) node or path expressions.

We will restrict now our attention to binary trees, but we will show in Section 7 that all results generalize to arbitrary unranked ordered trees.

We first define *tree configurations* for finite sets of variables. Unlike with binary trees as in Section 2.1 we will use trees with at most two children, i.e. we do *not* require the first child of a node to be defined whenever the second child is, or vice versa. We call those trees *pre-binary trees*.

DEFINITION 7 (TREE CONFIGURATIONS). *A* tree configuration *for a finite set of variables V is a pair (t, λ), where t is an unlabeled finite pre-binary tree and $\lambda : V \to dom(t)$ maps variables to nodes, such that each node of t is either the image of a variable, or is the least common ancestor of two nodes that are images of variables, or is the root.*

FACT 8. *A tree configuration for n variables can have at most $2n + 1$ nodes. Hence, there are only finitely many tree configurations for a given finite set of variables.*

Next, we define a class of path expressions that "only talk about the relevant part of the tree".

DEFINITION 9 (DOWNWARD EXPRESSIONS).

- *A downward node expression is a node expression the form $\mathsf{W}\phi$.*
- *A downward path expression is a path expression generated by the following inductive definition:*

$$\alpha ::= \downarrow_1 \mid \downarrow_2 \mid . \mid \alpha \cup \alpha' \mid \alpha/\alpha' \mid \alpha[\phi] \mid \alpha^*$$

  where $\downarrow_1$ and $\downarrow_2$ are shorthand for $\downarrow[\langle\rightarrow\rangle]$ and $\downarrow[\langle\leftarrow\rangle]$, respectively, and ϕ is a downward node expression.
- *A $\downarrow_i$-path expression ($i = 1, 2$) is a downward path expression of the form $\downarrow_i/\alpha$ or $.[\phi]/\downarrow_i/\alpha$.*

Finally, we can define tree patterns.

DEFINITION 10 (TREE PATTERNS). *A* tree pattern *over a finite set of variables V is a tuple $p(V) = (t, \lambda, \delta)$, where (t, λ) is a tree configuration for V, and δ assigns to each $\downarrow_i$-edge a $\downarrow_i$-path expression ($i = 1, 2$). In case t consists of only one node (the root), then δ assigns to it a downward node expression.*

The semantics of tree patterns is based on the notion of a *tree embedding*. An embedding of a tree t into a tree t' is a map $f : dom(t) \to dom(t')$ such that (i) the root is preserved, i.e., $f(\epsilon) = \epsilon$, (ii) descendant relationships are preserved, i.e., $f(x) <_i f(xi)$, for all $xi \in dom(t)$ with $i \in \{1, 2\}$, and (iii) least common ancestors are preserved, i.e., if x is the least common ancestor of y and z in t, then $f(x)$ is the least common ancestor of $f(y)$ and $f(z)$ in t'.

DEFINITION 11 (SEMANTICS OF TREE PATTERNS). *Given a tree pattern $p(V) = (t, \lambda, \delta)$ and a tree t', a* faithful embedding *of p into t' is a tree embedding $f : t \to t'$ such that*

- *for all $xi \in dom(t)$ (with $i \in \{1, 2\}$), $(f(x), f(xi)) \in [\![\delta(x, xi)]\!]^{t'}$,*
- *if $dom(t) = \{\epsilon\}$, then $\epsilon \in [\![\delta(\epsilon)]\!]^{t'}$*

A tree pattern $p(v_1, \ldots, v_n)$ is satisfied *by a k-tuple of nodes $x_1, \ldots, x_n$ in a tree t (notation: $t \models p\ [x_1, \ldots, x_n]$) if there is a faithful embedding of p into t sending $\lambda(v_i)$ to x_i for all $i \leq n$.*[1]

For convenience, we will often write $p(v)$ instead of $\lambda(v)$, for $p = (t, \lambda, \delta)$ a tree pattern and v a variable.

Since we have a notion of satisfaction for tree patterns, we can speak of *equivalence* of (unions of) tree patterns and FO(MTC)-formulas. It is not hard to see that tree patterns can be translated into FO(MTC):

LEMMA 12. *On binary trees, every union of tree patterns $\bigcup_i p_i$ in $v_1, \ldots, v_n$ is equivalent to an FO(MTC$^<$) formula with free variables $v_1, \ldots, v_n$.*

PROOF SKETCH. Every tree pattern can be straightforwardly translated to a FO(MTC)-formula, in fact, a conjunctive query over the FO(MTC)-translations of downward Regular XPath(W) expressions (see Lemma 6). The existentially quantified variables in the conjunctive query correspond to the "unnamed" nodes of the tree pattern. The *union*, finally, corresponds to a *disjunction*. □

[1]This notation, often used in logic, assumes an ordering on the variables $v_1, \ldots, v_n$, just to improve readability.

We will show in the next section that the converse also holds, and hence unions of tree patterns have exactly the same expressive power as FO(MTC)-formulas:

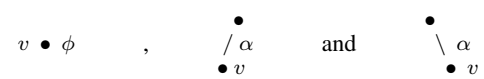
THEOREM 13. *On binary trees, every* $\mathsf{FO}(\mathsf{MTC}^<)$ ϕ *is equivalent to a union of tree patterns over FV*(ϕ).

Before proving Theorem 13 we first show how Theorem 1 follows from Theorem 13. We first need the following Lemma.

LEMMA 14. *On binary trees,*

1. *every union of tree patterns with no variable is equivalent to a Regular XPath(*W*) node expression interpreted at the root of the tree.*
2. *every union of tree patterns in a single variable* v *is equivalent to a Regular XPath(*W*) node expression.*
3. *every union of tree patterns in variables* u, v *is equivalent to a Regular XPath(*W*) path expression.*

PROOF. We prove the second claim, by means of a case distinction. The proofs for the other claims are similar. There are only three types of tree patterns in a single variable, namely



In the first case, the equivalent Regular XPath(W) node expression is simply $\neg\langle\uparrow\rangle \wedge \phi$, while in the second and third case it is $\langle\alpha^{-1}[\neg\langle\uparrow\rangle]\rangle$, where α^{-1} is the converse of α.

It follows that every *union* of tree patterns in v is equivalent to a *disjunction* of such Regular XPath(W) node expressions. □

PROOF OF THEOREM 1. As we will show in Section 7, we may restrict to binary trees without loss of generality. Consider a formula ϕ of $\mathsf{FO}(\mathsf{MTC})$ with at most two free variables, u, v. By Lemma 5, we can assume that ϕ is a $\mathsf{FO}(\mathsf{MTC}^<)$-formula. By Theorem 13, ϕ is equivalent to a union of tree patterns in the same variables. By Lemma 14 this union of tree patterns is equivalent to some Regular XPath(W) node expression or path expression. □

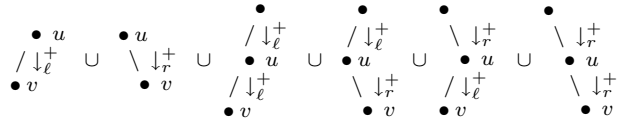
4. THE TRANSLATION FROM $\mathsf{FO}(\mathsf{MTC}^<)$ TO UNIONS OF TREE PATTERNS

We now turn to the proof of Theorem 13. The translation from $\mathsf{FO}(\mathsf{MTC}^<)$ formulas to unions of tree patterns goes by induction on the formula. We show that every atomic formula is equivalent to a union of tree patterns, and that unions of tree patterns are closed under all desired operations: intersection, complement, existential quantification and transitive closure. Theorem 13 then follows.

As in the previous section, we restrict attention to binary trees.

PROPOSITION 15. *Every atomic* $\mathsf{FO}(\mathsf{MTC}^<)$ *formula is equivalent on binary trees to a union of tree pattern.*

PROOF. The atomic formulas of $\mathsf{FO}(\mathsf{MTC}^<)$ are of the formulas of the form $u < v$, $u \prec v$, $P_\sigma(u)$, and $u = v$. We limit ourselves to the first case: $u < v$ is equivalent to the union



where $\downarrow_\ell^+$ is short for the $\downarrow_1$-path expression $\downarrow_1/\downarrow^*$ and $\downarrow_r^+$ is short for the $\downarrow_2$-path expression $\downarrow_2/\downarrow^*$. □

Proposition 16 below establishes closure of unions of tree patterns under the first-order operations. We first need to introduce some terminology. Given tree patterns $p(v_1, \ldots, v_n)$ and $p'(v_1, \ldots, v_n, u)$, we say that p is a *projection* of p' if for all binary trees t with nodes $x_1, \ldots, x_n$, $t \models p\,[x_1, \ldots, x_n]$ iff there is a node y such that $t \models p'\,[x_1, \ldots, x_n, y]$. Likewise, we say that a union of tree patterns $\bigcup_i p'_i(v_1, \ldots, v_n, u)$ is an *expansion* of a tree pattern $p(v_1, \ldots, v_n)$ if for all binary trees t with nodes $x_1, \ldots, x_n, y$, we have that $t \models \bigcup_i p'_i\,[x_1, \ldots, x_n, y]$ iff $t \models p\,[x_1, \ldots, x_n]$.

PROPOSITION 16.

1. *For all tree patterns* $p(V), p'(V)$ *there is a tree pattern* $p''(V)$ *defining the intersection of* $p(V)$ *and* $p'(V)$.
2. *For each tree pattern* $p(V)$ *there is a union of tree patterns* $\bigcup_i p'_i(V)$ *defining the complement of* $p(V)$.
3. *For each tree pattern* $p(V \cup \{u\})$ *there is a tree pattern* $p'(V)$ *that is a projection of* p *onto* V.
4. *For each tree pattern* $p(V)$ *there is a union of tree pattern* $\bigcup_i p'_i(V \cup \{u\})$ *that is an expansion of* $p(V)$.

The proof of Proposition 16 is quite standard. In particular, unions are used in order to cover all possible tree configurations (recall that by Fact 8 there are only finitely many tree configurations over a given finite set of variables). The downward path expression assigned to each edge is computed using classical techniques for regular languages, in particular their closure under the Boolean operations and projection. More details of the proof of Proposition 16 will be given in the full version of this paper.

We are left with the task to show that the unions of tree patterns are closed under the transitive closure operator $\mathsf{TC}^<$, which we will prove next. In order to simplify the presentation we will make use of a powerful model of two-way string automata that we now introduce.

4.1 Generalized string walking automata

We introduce a slightly unusual automaton model for the regular string languages, which will be more convenient to work with. The automata can make jumps from one position in a string to another, specified by regular expressions. A *generalized string walking automaton* (GSWA) for an alphabet Σ is a tuple $A = (\Sigma, Q, \delta, q_0, F)$, with $q_0 \in Q$, $F \subseteq Q$, and δ a finite subset of

$$Q \times \{\mathsf{down}^r_s(t), \mathsf{up}^r_s(t) \mid r, s, t \text{ regular expressions over } \Sigma\} \times Q$$

Let $w \in \Sigma^*$ be any string of length n. For $0 \leq i, j < n$ we denote by $w[i, j]$ the string obtained from w by reading the letters of w from position i to position j (we start counting positions at 0). If $j \leq i$, w is read backwards. A *configuration* of a GSWA $A = (\sigma, Q, \delta, q_0, F)$ on w is a pair (q, i) with $q \in Q$ and $0 \leq i < n$. An *accepting run* of A on w is a sequence of configurations starting with $(q_0, 0)$, ending in $(q, n-1)$ for some $q \in F$, and such that for each consecutive pair of configurations $(q, i), (q', j)$, one of the following holds for some regular expressions r, s, t:

- $(q, \mathsf{down}^r_s(t), q') \in \delta$, $i \leq j$, r accepts $w[0, i]$, t accepts $w[i, j]$, and s accepts $w[j, n-1]$.
- $(q, \mathsf{up}^r_s(t), q') \in \delta$, $j \leq i$, r accepts $w[0, j]$, t accepts $w[i, j]$, and s accepts $w[i, n-1]$.

A word is accepted by a GSWA iff there is an accepting run.

THEOREM 17. *The string languages accepted by GSWA are precisely the regular string languages.*

One direction follows from the fact that the regular expressions can already define all regular string languages. The other direction follows from the fact that GSWA can easily be translated to a two-way automaton using one pebble (which are known to define only regular languages). Another way to see this is to encode the run of a GSWA in MSO.

4.2 Closure under $\mathsf{TC}^<$

Recall from Lemma 12 that every union of tree patterns $\bigcup S$ may be thought of as an $\mathsf{FO}(\mathsf{MTC}^<)$-formula. Our aim, in this section, is to show the following:

THEOREM 18. *Let S be a set of tree patterns in x, y, u, v, such that for each $p \in S$, $p(u) \leq p(x), p(y), p(v)$ and $p(v) \not\leq p(x), p(y)$. Then there is a set of tree patterns S' in u, v defining $[\mathsf{TC}_{xy} \bigcup S](u, v)$.*[2]

PROOF. We may assume without loss of generality that all tree patterns in S agree on whether u is the root or not, and, in case u is not the root, all agree on the path expression assigned to the edge from the root to u, say β. We may also assume that $p(u) < p(v)$ for all $p \in S$.

To simplify the notation, below, whenever we have a particular tree pattern p in mind, we will write x instead of $p(x)$, and $lca(x, y)$ instead of $lca(p(x), p(y))$.

The proof proceeds in several steps.

Step 1. Classifying the tree patterns in three groups

Recall that, for any nodes x, y, $lca(x, y)$ denotes their least common ancestor. For each $p \in S$, we can distinguish three cases: $lca(x, v) < lca(y, v)$, or $lca(x, v) = lca(y, v)$, or $lca(x, v) > lca(y, v)$. Correspondingly, we can split S into $S_\downarrow \cup S_{subtree} \cup S_\uparrow$. Intuitively, the subscript indicates what *going from x to y* means in terms of movement on the main path from u to v. See Figure 1, where all possible tree configurations are indicated and classified (modulo horizontal symmetry, and leaving out the root node).

Step 2. Describing each tree pattern by a number of path expressions.

To each $p \in S_{subtree}$ we assign three path expressions:

α^p_{pre} traveling from u to $lca(x, v)$,

α^p_{post} traveling from $lca(x, v)$ to v, and

α^p_{xy} traveling from x to y within the subtree rooted by $lca(x, v)$.

For a prototypical example, see Figure 2(a). The other cases are similar (note that in some cases, some of the above three might be simply the path expression '.'). Note that, while the last of these path expressions might involve upward movement, the first two are always downward.

Likewise, we assign to each element of $S_\downarrow$ and $S_\uparrow$ five path expressions:

α^p_{pre} traveling from u to $lca(x, v)$ (in case $p \in S_\downarrow$) or from u to $lca(y, v)$ (in case $p \in S_\uparrow$)

α^p_{post} traveling from $lca(y, v)$ to v (in case $p \in S_\downarrow$) or from $lca(x, v)$ to v (in case $p \in S_\uparrow$)

α^p_{out} traveling from x to $lca(x, v)$,

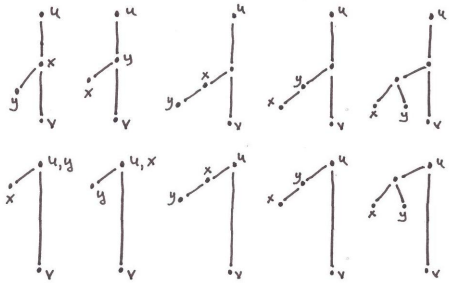
α^p_{main} traveling from $lca(x, v)$ to $lca(y, v)$ (in case $p \in S_\downarrow$) or vice versa (in case $p \in S_\uparrow$)

α^p_{in} traveling from $lca(y, v)$ to y,

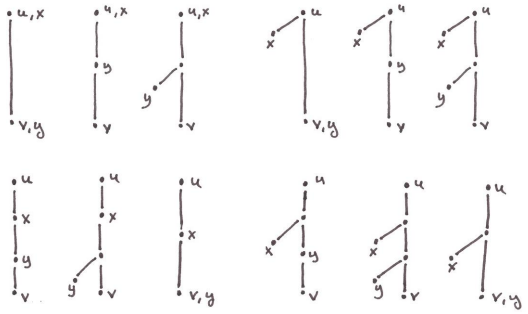
See Figure 2(b) and Figure 2(c). Of the above five path expressions, the first two and the last one are downward, while the third one is upward. The fourth path expression is in general neither downward nor upward.

[2]We are treating $\bigcup S$ here as an $\mathsf{FO}(\mathsf{MTC})$ formula, cf. Lemma 12.

Tree patterns in $S_{subtree}$ (i.e., $lca(x, v) = lca(y, v)$):



Tree patterns in $S_\downarrow$ (i.e., $lca(x, v) < lca(y, v)$):



Tree patterns in $S_\uparrow$ (i.e., $lca(y, v) < lca(x, v)$):

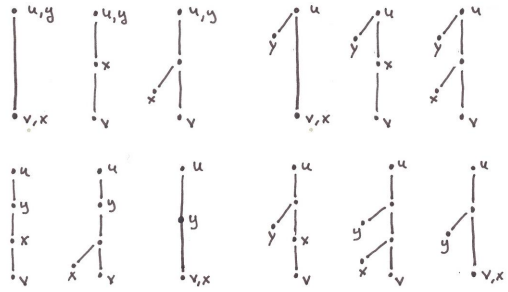
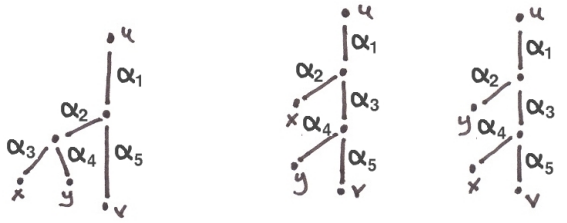


Figure 1: Classification of tree patterns

Step 3. Defining a generalized string walking automaton

Downward path expressions are essentially regular expressions over an alphabet consisting of Boolean combinations of downward node expressions plus $\downarrow_1$ and $\downarrow_2$. Rather than directly specifying a downward path expression describing the overall walk from u to v, we will exploit the connection with regular expressions, and specify it by means of a GSWA. Afterward, we will convert the GSWA to a regular expression, i.e., a downward path expression. The alphabet over which the GSWA is defined will consist of Boolean combinations of downward path expressions, as well as $\downarrow_1$ and $\downarrow_2$. Precisely which path expressions will become clear during the construction, but at any rate it will be only finitely many.

The automaton has a state for each $p \in S_\uparrow \cup S_\downarrow$, plus an initial state q_0. Intuitively, a state p represents the situation where *"the automaton came from $p(x)$ and is currently standing at $lca(p(y), p(v))$, ready to move towards $p(y)$."* A transition from p to p' then intuitively corresponds to *"moving to $p(y)$, then moving around in the subtree any number of times using the tree patterns in $S_{subtree}$, then moving from $p'(x)$ to $lca(p'(x), p'(v))$*



$\alpha_{pre} = \alpha_1$
$\alpha_{post} = \alpha_5$
$\alpha_{xy} = \alpha_3^{-1}[\langle\alpha_2^{-1}[\mathsf{root}]\rangle]/\alpha_4$

(a)

$\alpha_{pre} = \alpha_1$
$\alpha_{post} = \alpha_5$
$\alpha_{out} = \alpha_2^{-1}$
$\alpha_{main} = \alpha_3$
$\alpha_{in} = \alpha_4$

(b)

$\alpha_{pre} = \alpha_1$
$\alpha_{post} = \alpha_5$
$\alpha_{out} = \alpha_4^{-1}$
$\alpha_{main} = \alpha_3$
$\alpha_{in} = \alpha_2$

(c)

Figure 2: Associating path expressions to the tree patterns

as indicated in p', and finally moving from $lca(p'(x), p'(v))$ to $lca(p'(y), p'(v))$." Formally, if $p, p' \in S_\downarrow \cup S_\uparrow$ and $\mathcal{S} \subseteq S_{subtree}$, then there is a transition from p to p' labeled by

$$\mathsf{down}^{\bigcap\{\alpha^q_{pre}|q\in\mathcal{S}\cup\{p'\}\}}_{\bigcap\{\alpha^q_{post}|q\in\mathcal{S}\cup\{p'\}\}}(.[\mathsf{W}(\alpha^p_{in}/\Big(\bigcup_{q\in\mathcal{S}}\alpha^q_{xy}\Big)^*/\alpha^{p'}_{out})]/\alpha^{p'}_{main})$$

in case $p' \in S_\downarrow$, or

$$\mathsf{up}^{\bigcap\{\alpha^q_{pre}|q\in\mathcal{S}\cup\{p'\}\}}_{\bigcap\{\alpha^q_{post}|q\in\mathcal{S}\cup\{p'\}\}}(.[\mathsf{W}(\alpha^p_{in}/\Big(\bigcup_{q\in\mathcal{S}}\alpha^q_{xy}\Big)^*/\alpha^{p'}_{out})]/\alpha^{p'}_{main})$$

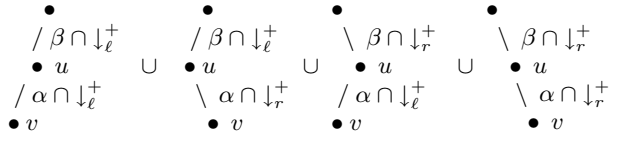
in case $p' \in S_\uparrow$. Furthermore, from the initial state there is a transition to each tree pattern p satisfying $p(x) = p(u)$, labeled by

$$\mathsf{down}^{\alpha^p_{pre}}_{\alpha^p_{post}}(\alpha^p_{main})\,.$$

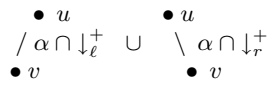
The final states are all tree patterns p satisfying $p(y) = p(v)$.

Step 4. Converting the GSWA to a downward path expression

Since GSWA define regular string languages, we can convert the above automaton to a regular expression, i.e., a downward path expression α. It follows from the construction that the union of tree patterns



or, in case u is the root,



is equivalent to the initial transitive closure formula. Here, $\downarrow^+_\ell$ and $\downarrow^+_r$ are again shorthands for $\downarrow_1/\downarrow^*$ and $\downarrow_2/\downarrow^*$, respectively. We use here the fact that for downward path expressions α, $\alpha \cap \downarrow^+_\ell$ and $\alpha \cap \downarrow^+_r$ can be written as a $\downarrow_1$-path expression and $\downarrow_2$-path expression, respectively, as can be shown by induction on the expression α. □

5. STRICT CONTAINMENT IN MSO

In this section we prove Theorem 2, which says that nested TWA (or equivalently, FO(MTC)) fail to recognize all regular tree languages. The separating tree language $\mathcal{L}_{\text{sep}}$ that we use is the same tree language used in [6] to separate the regular tree languages from those recognized by TWA with pebbles. Actually, not much needs to be done: the construction used in [6] already implicitly shows that nested TWA fail to capture all regular tree languages. For the sake of completeness, we present the separating language here and the proof of [6] adapted to our setting and with our notations will be provided in the full version of this paper.

A *quasi-blank-tree* is a tree over the alphabet $\{\mathbf{a}, \mathbf{b}\}$ in which the label **a** occurs only at leaves of the tree. Every quasi-blank-tree t induces a *branching structure* $\text{bs}(t)$, which is an unlabeled tree whose nodes are all the **a**-leaves of t, plus all nodes that are the least common ancestor of two **a**-leaves in t. The descendant-relation of the nodes of $\text{bs}(t)$ is inherited from t.

Recall that in a finite binary tree each node can be naturally addressed by a $\{1,2\}$-string describing the path from the root to the node, where 1 corresponds to taking the left child of a node, and 2 corresponds to taking the right child. In this spirit, a 1^*2-*node* is a right child of a node of the leftmost path starting at the root. $\mathcal{L}_{\text{even}}$ is the set of quasi-blank binary trees t such that $\text{bs}(t)$ has an even number of 1^*2-nodes v whose subtree has all branches of even length.

The language $\mathcal{L}_{\text{even}}$ is the building block of the separating language. Consider trees over the alphabet $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$. Call such trees *well formed* if all **a**-nodes are leaves, and the root is labeled by **c**. With a slight abuse of notation, we will view quasi-blank trees as a special case of well formed trees, where the root is the unique node with label **c**. Let t be a well formed tree and v a **c**-node of t, such that no descendant of v is labeled **c** (hence $\mathsf{subtree}(t, v)$ is a quasi-blank tree). The *rewriting of t at v* is tree obtained by replacing $\mathsf{subtree}(t, v)$ by a single node of label **a** if $\mathsf{subtree}(t, v) \in \mathcal{L}_{\text{even}}$ and of label **b** otherwise. This defines a confluent rewriting process over well formed binary trees that eventually reduces every tree to a single node labeled either **a** or **b**. Let $\mathcal{L}_{\text{sep}}$ be the set of well formed binary trees that reduce to **a** by this rewriting process. Note that the quasi-blank trees of $\mathcal{L}_{\text{sep}}$ are exactly those of $\mathcal{L}_{\text{even}}$.

It is not hard to see that the rewriting process described above can be simulated by a simple bottom-up automaton. Hence,

FACT 19 ([6]). *$\mathcal{L}_{sep}$ is a regular tree language, i.e., it is definable in MSO.*

It therefore only remains to show:

THEOREM 20. *$\mathcal{L}_{sep}$ cannot be recognized by a nested TWA.*

The proof, omitted in this abstract, goes by induction on the rank of the nested TWA. Roughly speaking, no Boolean combination of nested TWA of rank 0 (i.e., simple TWA) can recognize $\mathcal{L}_{\text{even}}$, as was shown in [6]. Nested TWA of rank 1 can recognize $\mathcal{L}_{\text{even}}$, but get lost in trees containing an intermediate layer of **c**-nodes, etc. As $\mathcal{L}_{\text{sep}}$-trees can contain arbitrarily many such layers, no nested TWA can recognize it.

6. COMPLEXITY ISSUES

A standard bottom up algorithm shows that query evaluation for Regular XPath(W) can be performed in PTime combined complexity, and the same holds for nested TWA.

THEOREM 21. *Given a tree t and a Regular XPath(W) path expression α, $[\![\alpha]\!]^t$ can be computed in polynomial time.*

This means that query evaluation for Regular XPath(W) is no harder than for Core XPath, up to a polynomial. For static analysis tasks, the situation is a bit different: as we will show next, emptiness for nested TWA, and satisfiability for Regular XPath(W) expressions, are 2-ExpTime-complete. Note that satisfiability for Core XPath is only ExpTime-complete.

THEOREM 22. *The satisfiability problem of Regular XPath(*W*) is* 2-EXPTIME*-complete.*

The proof of this result will appear in the full version of this paper. The upper bound is based on a double exponential translation from Regular XPath(W) formulas to bottom-up tree automata. It is well known that emptiness of bottom-up tree automata can be tested in PTime. The lower bound is based on a reduction from exponential space bounded alternating Turing machines.

The same holds for many other static analysis problems for Regular XPath(W), such as *query containment* and *query equivalence*, since they are all interreducible, using the negation operator in node expressions (cf. for instance [19]).

Theorem 22 is to be compared with a similar result over nested words obtained in [1]. In [1] the authors considered an extension of CaRet with a "within" operator and satisfiability over nested words was shown to be 2-EXPTIME-complete. As nested words are trees presented as words and CaRet is a temporal logic, the two results are similar in spirit. However the extension of CaRet with the "within" operator has the expressive power of FO over trees while Regular XPath(W) has the stronger expressive power of $\mathsf{FO}(\mathsf{MTC})$.

In fact we can prove stronger results than those of Theorem 21 and Theorem 22, as we can show that they holds if the input is a nested TWA. We first comment on the size of a nested TWA. Given a nested TWA A, its state-size $|A|$ is the total number of states of all the nested TWA that are used in the inductive definition of A. The total-size of $||A||$ is the sum of the state-size of A and the total number of transition of all the nested TWA that are used in the inductive definition of A. Note that $||A||$ can be exponential in $|A|$. Recall that the translation from Regular XPath(W) to nested TWA described in Lemma 4, since it runs in linear time, always produces a nested TWA whose *total-size* is linearly bounded by the length of the input expression.

THEOREM 23. *Given a nested TWA A, testing emptiness of A is* 2-EXPTIME*-complete and computing $A(t)$ can be done in polynomial time. This holds both when the complexity is measured in terms of the* state-size *and when it is measured in terms of the* total-size*.*

7. FROM BINARY TO UNRANKED TREES

In this section, we discuss a well known encoding of unranked ordered trees into binary ones, that allows us to generalize our results to the general, unranked case.

For any tree t, $\mathsf{bin}(t)$ is a binary tree that has the same nodes as t but in which each node has at most two children: the first child of a node x in $\mathsf{bin}(t)$ corresponds to the first child of x in t, and the second child of x in $\mathsf{bin}(t)$ corresponds to the next-sibling of x in t if it exists. To ensure that every non-leaf node has exactly two children, we pad $\mathsf{bin}(t)$ with extra nodes as needed, having a designated label $\#$. For any set of unranked ordered trees L, we denote by $\mathsf{bin}(L)$ the set of all binary encodings of trees in L.

It is well known that this encoding preserves MSO definability: if L is an MSO definable set of unranked ordered trees then $\mathsf{bin}(L)$ is also MSO definable, and vice versa, and likewise for unary and binary MSO queries. It is not hard to see that the same holds for $\mathsf{FO}(\mathsf{MTC})$. It is less easy to show that the binary encoding also preserves (in both directions) definability in Regular XPath(W). Below, we will prove this for Boolean queries, i.e., tree languages. The same arguments apply to unary and binary queries.

A set of trees L is said to be *definable in Regular XPath(*W*)* if there is an Regular XPath(W) node expression ϕ such that L is precisely the set of all trees satisfying ϕ at the root.

THEOREM 24. *For any set of trees L, $\mathsf{bin}(L)$ is definable in Regular XPath(*W*) iff L is.*

In order to prove Theorem 24, it is convenient to introduce a *subforest* predicate W′. By a *forest*, we will mean an ordered sequence of trees, such as obtained by removing the root node from a tree. Regular XPath(W) expression can be evaluated on forests in the same way as they are evaluated on trees (cf. 1). In particular, trees are a special case of forests. For any forest f and node x, we denote by $\mathsf{subforest}(f, x)$ the subforest of f consisting of x and its descendants, plus all siblings to the right of x and their descendants. Now, the W′ operator has the following semantics:

$$[\![\mathsf{W}'\phi]\!]^f = \{x \in dom(f) \mid x \in [\![\phi]\!]^{\mathsf{subforest}(f,x)}\}$$

It is easy to see that Regular XPath(W) definability of $\mathsf{bin}(L)$ implies definability of L in Regular XPath(W′), and vice versa. Indeed, applying the W operator in $\mathsf{bin}(t)$ precisely corresponds to applying the W′ operator in t. Thus,

PROPOSITION 25. *For any set of trees L, $\mathsf{bin}(L)$ is definable in Regular XPath(*W*) iff L is definable in Regular XPath(*W′*).*

It remains to show that, over unranked ordered trees, W′ does not give us any more expressive power than W, i.e., all occurrences of W′ can be eliminated in the favor of W. The main idea behind the proof is the following. Consider any node expression of the form $\mathsf{W}'\phi$, where ϕ itself is a Regular XPath(W) node expression, i.e., does not use W′. When $\mathsf{W}'\phi$ is evaluated at a node x of t, the evaluation of ϕ on $\mathsf{subforest}(t, x)$ can be decomposed into horizontal navigation along the sequence of roots of $\mathsf{subforest}(t, x)$ and subtree tests (i.e., using W) at each of these roots. We rewrite the formula so that this horizontal navigation becomes uni-directional from left to right, and hence the formula depends only on the selected subforest, which means that may drop the W′ operator.

In order to make this precise, we need to introduce two notions. We call a Regular XPath(W) node expression *tree-local* if its subprograms never attempt to leave the tree in which it started. In other words every left or right move is preceded by a test to ensure that the current node is not a root. On trees, of course every Regular XPath(W) node expression is equivalent to a tree-local one, but on forests this is in general not the case. Still, every downward node expression, i.e., Regular XPath(W) node expression of the form $\mathsf{W}\phi$, is equivalent on forests to a tree-local one. A *forest test* is an MSO formula in one free (first-order) variable over the vocabulary consisting of $\prec$ plus a unary predicate for each downward node expression. We say that a forest test is true at a node x in a forest iff the MSO formula is true when evaluated on the sequence of roots of the forest, where the free variable is interpreted as the root of the tree to which x belongs.

PROPOSITION 26. *On forests, every Regular XPath(*W*) node expression is equivalent to a Boolean combination of tree-local Regular XPath(*W*) node expressions and forest tests.*

The proof is by induction and is omitted here.

We are now ready to prove Theorem 24.

PROOF OF THEOREM 24. First, suppose L is a set of trees defined by a Regular XPath(W) node expression ϕ. Let ϕ' be obtained from ϕ by

- replacing all occurrences of $\downarrow$ by $\downarrow_1/\downarrow_2^*[\neg\#]$,
- replacing all occurrences of $\rightarrow$ by $\downarrow_2[\neg\#]$,
- replacing all occurrences of $\uparrow$ by $(\downarrow_2^{-1})^*/\downarrow_1^{-1}$, and
- replacing all occurrences of $\leftarrow$ by $\downarrow_2^{-1}$.

where $\downarrow_1$ and $\downarrow_2$ are short for $\downarrow[\neg\langle\leftarrow\rangle]$ and $\downarrow[\neg\langle\rightarrow\rangle]$, respectively. Furthermore, let χ be a Regular XPath(W) node expression defining the class of all trees that are the binary encoding of some other tree (it is not hard to find such a node expression). Then $\phi' \wedge \chi$ defines $\mathsf{bin}(L)$.

Next, suppose that ϕ defines $\mathsf{bin}(L)$. By Proposition 25, L is defined by a Regular XPath(W′) node expression ψ. We show how to eliminate all occurrences of W′ from ψ in the favor of W, using Proposition 26. To this end, consider $\mathsf{W}'\chi$, for any Regular XPath(W) node expression χ. By proposition 26, χ is equivalent to a Boolean combination of tree-local node expressions and forest tests. Since χ is evaluated at the left-most root of the subforest, all the tree-local node expressions may be freely prefixed with W, and may be considered as part of the forest test. Thus, χ essentially expresses that the sequence of roots of the subforest belongs to a regular language over an alphabet consisting of Boolean combinations of downward node expressions. Any such regular string language can be represented by a regular expression, which is nothing more than a Regular XPath(W) path expression α built up from $\rightarrow$ and downward node expressions. It follows that $\mathsf{W}'\phi$ is equivalent to the Regular XPath(W) node expression $\langle\alpha\rangle$. A simple induction based on this argument shows that every Regular XPath(W′) node expression, including ψ, is equivalent on arbitrary forests, hence also on trees, to a Regular XPath(W) node expression. □

REMARK 27. *Note that the proof of the right-to-left direction of Theorem 24 is based on a simple linear time translation. Thus, the satisfiability problem for Regular XPath(*W*) on arbitrary trees polynomially reduces to the satisfiability problem on binary trees.*

8. DISCUSSION

We conclude by some additional consequences of our results.

Closure properties.

An important corollary of our expressive completeness theorem is that Regular XPath(W) is closed under the path intersection and complementation operators of XPath 2.0, in the sense that adding these operators would not increase the expressive power (though it would increase the complexity of satisfiability). Note that this does not follow immediately from the definition of Regular XPath(W). In [3], closure under path intersection and complementation was investigated for a variety of XPath fragments, and most results were negative.

Along the same lines, Benedikt and Fundulaki [4] introduced and motivated the notion of *subtree composition closure*. For any path expression α and XML tree t, let $[\![\alpha]\!]_\varepsilon^t = \{x \mid (\varepsilon, x) \in [\![\alpha]\!]^t\}$, i.e., the set of nodes reachable from the root by α. An XPath dialect is said to be *closed under subtree composition* if the following holds: *for any two path expressions α, β, there is a path expression γ such that for any XML tree t, $[\![\gamma]\!]_\varepsilon^t = \bigcup\{[\![\beta]\!]_\varepsilon^{\mathsf{subtree}(t,x)} \mid x \in [\![\alpha]\!]_\varepsilon^t\}$*. In [4], a number of XPath fragments were shown to be closed under subtree composition. Theorem 1 easily implies that Regular XPath(W) is closed under subtree composition.

COROLLARY 28. *Regular XPath(*W*) is closed under path intersection, path complementation, and subtree composition.*

Moreover, the question whether W contributes to the expressive power of Regular XPath(W) reduces to the question whether Regular XPath is closed under subtree composition:

THEOREM 29. *The following are equivalent*

1. *Regular XPath is as expressive as Regular XPath*(W)
2. *Regular XPath is closed under subtree composition.*

PROOF. One direction is easy: if Regular XPath has the same expressive power as Regular XPath(W), then by Corollary 28 it is closed under subtree composition. Conversely, suppose Regular XPath is closed under subtree composition, and let ϕ be a Regular XPath node expression. Let α be the subtree composition of $\downarrow^*$ and $.[\phi]$, and α^{-1} its converse (recall that Regular XPath path expressions are closed under converse). Then $\mathsf{W}\phi$ is equivalent to $.[\langle\alpha^{-1}\rangle]$. □

Normal forms for FO(MTC) .

Theorem 1 implies a number of normal form theorems for FO(MTC) over trees. Given $n \in \mathbb{N}$, we say that a logic *has the n-variable property* if every formula of the logic with at most n free variables can be equivalently written with n variables in total.

Let $\mathsf{FO}^4(\mathsf{MTC})$ be the set of FO(MTC) formulas containing at most four variables, free or bound. Also, recall that in formulas of the form $[\mathsf{TC}_{xy}\phi](u, v)$, ϕ might have other free variables besides x, y, and that we call these additional free variable *parameters* of the transitive closure formula. We say that a FO(MTC) formula φ is "single-parameter" if for all subformulas $[\mathsf{TC}_{xy}\psi](u, v)$ of φ, ψ has only one parameter, and we say that φ is "parameter-free" if for all subformulas $[\mathsf{TC}_{xy}\psi](u, v)$ of φ, x and y are the only free variables of ψ. The direct translation from Regular XPath(W) into FO(MTC) shows that FO(MTC) has the 4-variable property.

THEOREM 30. *Every* FO(MTC) *formula with at most four free variables is equivalent to a single-parameter* $\mathsf{FO}^4(\mathsf{MTC})$ *formula.*

One might wonder whether parameters are needed at all. Perhaps every FO(MTC) formula is equivalent to a parameter-free formula? We do not know the answer to this question at present. However we know that the parameter-free fragment of FO(MTC), denoted by FO^* in [18], has exactly the same expressive power as $\mathsf{Regular\ XPath}^{\approx}$ (see also our discussion of related work in Section 1) and that FO^* has the three variable property [18].

From the nested TWA characterization of FO(MTC) we can derive a second normal form theorem for FO(MTC). A FO(MTC)-formula ϕ is said to be "looping" if all subformulas with main connective TC are of the form $[\mathsf{TC}_{xy}\ \phi](u, u)$. In [16] it is shown that TWA can be translated into looping FO(MTC)-formulas. Actually the translation yields formulas containing only two nestings of TC, the inner for computing the depth of the current node modulo a fixed number, and the outer for simulating a run of the automaton. With an induction using this idea we believe one can show:

Every FO(MTC)*-formula with at most one free variable is equivalent to a looping* FO(MTC)*-formula.*

On the power of sequential tree automata.

Theorem 2, in some sense, provides a formal justification for the intuition that sequential automata cannot capture all regular tree languages. Finding a sequential type of automata that could capture all regular tree languages is something desirable, and several attempts have been made. Tree walking automata, even with pebbles, are known to be not powerful enough [6]. Tree walking automata with an unbounded number of invisible pebbles are powerful enough [8] but these automata are not strictly speaking *finite state automata* anymore. Our Theorem 2 implies that, in fact, *no type of sequential automata translatable to* FO(MTC) *is powerful enough*. We leave it to the reader to decide if being translatable into FO(MTC) is a natural formalisation of "being sequential".

We also believe that our results imply a new characterization of tree walking automata with nested pebbles. Call a Regular XPath(W) expression *positive* if negation is only used in the form $\neg\sigma$ ($\sigma \in \Sigma$), $\neg\langle\uparrow\rangle$, $\neg\langle\downarrow\rangle$, $\neg\langle\leftarrow\rangle$ and $\neg\langle\rightarrow\rangle$. Recall that FO(pos-MTC) is the fragment of FO(MTC) in which all occurrences of TC-operators are in the scope of an even number of negations. By an easy inductive translation, every positive Regular XPath(W) expression is equivalent to an FO(pos-MTC) formula. Conversely, we believe that our translation can be fine-tuned in order to "preserve negation", so that FO(pos-MTC) formulas are mapped to *positive* Regular XPath(W) expressions. The latter in turn are equivalent to *positive nested TWA*, where only positive tests are possible, of the form: "if (at least) these subautomata have an accepting run from the current node then do this". In particular, this would imply that these positive nested TWA have exactly the same expressive power as the pebble TWA from [7]. Perhaps this new characterization could help in resolving the open question whether FO(MTC) is strictly more expressive than FO(pos-MTC) on trees.

Infinite trees.

A close inspection of our proofs shows that only the complexity results of Theorem 22 and Theorem 23 use the fact that the trees are finite (by computing a bottom-up automata). Hence Regular XPath(W), nested TWA and FO(MTC) have the same expressive power over infinite trees, and are strictly less expressive than MSO. Note that each of the formalisms can express the fact that the tree is finite by, for instance, visiting sequentially all the leaves of the tree. As for complexity, our techniques relying on bottom-up tree automata no longer work for infinite trees. However, we think that the same complexity can be achieved with appropriate top-down tree automata.

9. REFERENCES

[1] R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *LICS*, pages 151–160, 2007.

[2] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *PODS*, pages 25–36, 2005.

[3] M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. *Theoretical Computer Science*, 336(1):3–31, 2005.

[4] M. Benedikt and I. Fundulaki. XML subtree queries: Specification and composition. In *DBPL*, number 3774 in LNCS, pages 138–153, 2005.

[5] S. Bird, Y. Chen, S. B. Davidson, H. Lee, and Y. Zheng. Extending XPath to support linguistic queries. In *PLAN-X*, pages 35–46, 2005.

[6] M. Bojańczyk, M. Samuelides, T. Schwentick, and L. Segoufin. Expressive power of pebble automata. In *ICALP (1)*, pages 157–168, 2006.

[7] J. Engelfriet and H. J. Hoogeboom. Nested pebbles and transitive closure. *Logical Methods in Computer Science*, 3(2), 2007.

[8] J. Engelfriet, H. J. Hoogeboom, and B. Samwel. XML transformation by tree-walking transducers with invisible pebbles. In L. Libkin, editor, *PODS*, pages 63–72. ACM Press, 2007.

[9] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting regular xpath queries on XML views. In *ICDE*, pages 666–675, 2007.

[10] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106, 2002.

[11] E. Grädel. On transitive closure logic. In E. Börger, G. Jäger, H. K. Büning, and M. M. Richter, editors, *CSL*, volume 626 of *LNCS*, pages 149–163. Springer, 1991.

[12] F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *LICS*, pages 383–392, 2002.

[13] M. Marx. Conditional XPath. *Transactions on Database Systems*, 30(4):929–959, 2005.

[14] M. Marx and M. de Rijke. Semantic characterizations of navigational XPath. *SIGMOD Record*, 34(2):41–46, 2005.

[15] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2002.

[16] F. Neven and T. Schwentick. On the power of tree-walking automata. *Inf. Comput.*, 183(1):86–103, 2003.

[17] A. Potthoff. *Logische Klassifizierung regularer Baumsprachen*. PhD thesis, Christian-Albrechts Universität Kiel, 1994.

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

[19] B. ten Cate and C. Lutz. The complexity of query containment in expressive fragments of xpath 2.0. In *PODS*, pages 73–82. ACM, 2007.

[20] B. ten Cate and M. Marx. Axiomatizing the logical core of XPath 2.0. In *ICDT*, 2007.