Transitive Closure Logic, Nested Tree Walking Automata, and XPath

Balder ten Cate
ISLA, Universiteit van Amsterdam
and
Luc Segoufin
INRIA, ENS-Cachan

We study FO(MTC), first-order logic with monadic transitive closure, a logical formalism in between FO and MSO on trees. We characterize the expressive power of FO(MTC) in terms of nested tree-walking automata. Using the latter we show that FO(MTC) is strictly less expressive than MSO, solving an open problem. We also present a temporal logic on trees that is expressively complete for FO(MTC), in the form of an extension of the XML document navigation language XPath 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) for Boolean, unary and binary queries. We also investigate the complexity of the automata model as well as the XPath dialect. We show that query evaluation be done in polynomial time (combined complexity), but that emptiness (or, satisfiability) is 2ExpTime-complete.

Categories and Subject Descriptors: []:

1. INTRODUCTION

When studying query languages for tree structured data, two natural yardsticks 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. In between the two lies FO(MTC), first-order logic extended with an operator for taking the transitive closure of definable binary relations. We propose FO(MTC) as another yardstick of expressive power, and develop a variant of the XML path language XPath that is expressively complete for FO(MTC).

Our main contributions are the following:

(1) We give an automata theoretic characterization of FO(MTC) in terms of *nested tree walking automata*. Roughly speaking, these extend ordinary (non-deterministic) tree walking automata 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

We would like to thank Mikołaj Bojańczyk and Hendrik-Jan Hoogeboom 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.

show that these automata capture FO(MTC) with respect to Boolean queries, unary queries and binary queries.

Theorem 1 *Nested tree walking automata have the same expressive power as* FO(MTC) *on trees.*

(2) We show that nested tree walking automata, and hence also FO(MTC), are strictly less expressive than MSO on finite trees (and hence also on infinite trees), which solves an open problem raised in [Potthoff 1994; Engelfriet and Hoogeboom 2007]. This result can be seen as the culmination of a line of recent research [Bojańczyk and Colcombet 2008; Bojańczyk et al. 2006] separating increasingly rich versions of tree walking automata from MSO.

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

Note that FO(MTC) and MSO are equally expressive on finite and on infinite strings. However, on infinite trees, FO(MTC) is strictly less expressive than MSO because it is contained in *weak* MSO (the fragment of MSO restricting quantifications to *finite* sets. For instance $[\mathsf{TC}_{xy}\phi](u, v)$ can be expressed in weak MSO as $\exists^{\mathit{fin}} X \forall^{\mathit{fin}} Y (Yu \wedge \forall xy(Yx \wedge \phi \to Yy) \to Yv))$ which is known as being strictly less expressive than MSO over infinite trees [Rabin 1970]. Over infinite trees Theorem 2 shows that FO(MTC) is actually strictly less expressive than weak MSO.

(3) We present a temporal logic on trees that is expressively complete for FO(MTC), in the form of an extension of the XML document navigation language XPath. The logic in question, called Regular XPath(W), extends Core XPath [Gottlob et al. 2002] with the Kleene star for transitive closure, and with the subtree relativisation operator W. We show that Regular XPath(W) has the same expressive power as nested tree walking automata, and therefore FO(MTC), with respect to Boolean, unary and binary queries.

Theorem 3 *Regular XPath*(W) *has the same expressive power as nested tree walking automata on trees.*

Theorem 3 can be seen as providing a normal form for FO(MTC)-formulas on trees, which uses only four variables and a restricted from of transitive closure. We also derive that Regular XPath(W) is closed under the path intersection and complementation, even though these are not primitive operators in the language.

Extending XPath with the Kleene star for transitive closure has been advocated for a variety of reasons, ranging from practical (e.g., [Nentwich et al. 2002]), 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 [Marx 2004]. It also enables view based query rewriting for recursive views [Fan et al. 2007]. The extension of Core XPath with the Kleene star is known as Regular XPath [Marx 2004]. The W operator can be seen as a special case of the XPath *subtree scoping* operator in [Bird et al. 2005], and is closely related to *subtree query composition* as studied in [Benedikt and Fundulaki 2005].

All results hold both on finite and on infinite (ranked or unranked, sibling ordered) trees.

We also determine the computational complexity of basic tasks involving nested tree walking automata and Regular XPath(W) expressions, namely the evaluation problem and the non-emptiness (or, satisfiability) problem. The former can be solved in polynomial time, and the latter is 2ExpTime-complete, both for nested tree walking automata and for Regular XPath(W) (compared to ExpTime-complete for unnested tree walking automata and Core XPath).

The fact that FO(MTC) is strictly included in MSO, in some sense, provides a formal justification for the intuition that sequential finite 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 [Bojańczyk et al. 2006]. Tree walking automata with an unbounded number of invisible pebbles are powerful enough [Engelfriet et al. 2007] but these automata are not strictly speaking finite state automata anymore. Our results imply 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 formalization of "being sequential".

All in all, our results show that the tree languages definable in FO(MTC) form a robust class that can be characterized in several ways.

Related work. Connections between transitive closure logic and TWA have already been observed in [Engelfriet and Hoogeboom 2007]. The automata model used in [Engelfriet and Hoogeboom 2007] 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 [Engelfriet and Hoogeboom 2007] 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 [Bojańczyk et al. 2006]. Our proof of Theorem 2 uses the same separating language, and the same proof technique as in [Bojańczyk et al. 2006]. In Section 8 we will see that pebble TWA can be seen as the *positive* fragment of nested TWA.

The logical core of XPath 1.0, *Core XPath* [Gottlob et al. 2002], has been studied in detail. For instance, the combined complexity of query evaluation is known to be in PTime [Gottlob et al. 2002], static analysis tasks such as satisfiability are ExpTime-complete [Benedikt et al. 2005], and the expressive power of Core XPath has been characterized in terms of FO^2 [Marx and de Rijke 2005], the two-variable fragment of first-order logic on trees. Two extensions of Core XPath have subsequently been proposed and studied that capture full first-order logic, namely Conditional XPath [Marx 2005], extending Core XPath with "until" operators, and Core XPath 2.0 [ten Cate and Marx 2007]. Likewise, extensions of Core XPath have been proposed that are expressively complete for MSO, for instance with least fixed point operators (see, e.g., [ten Cate 2006, Sect. 4.2]).

In [ten Cate 2006], 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 $\mathsf{FO}(\mathsf{MTC})$. It is not known at present whether FO^* is as expressive as full $\mathsf{FO}(\mathsf{MTC})$.

A complete axiomatization for the $\mathsf{FO}(\mathsf{MTC})$-theory of finite trees was recently obtained in [Gheerbrant and ten Cate 2009].

The W operator we use is a generalization to trees of the "now" operator from temporal logics with forgettable past [Laroussinie et al. 2002]. It is closely related to the "within" operator in temporal logics for nested words [Alur et al. 2007], the XPath "subtree scoping" operator proposed in [Bird et al. 2005], and the issue of *subtree query composition* studied in [Benedikt and Fundulaki 2005]. See Section 8 for more details on the connection.

Organization of the paper. In Section 2, we introduce Regular XPath(W), $\mathsf{FO}(\mathsf{MTC})$, and nested TWA. In Section 3 and 4, we prove Theorem 3. 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. We have pushed into the Appendix two proofs that would break the flow of the paper. The first one, in Appendix A, proves an initial normal form for $\mathsf{FO}(\mathsf{MTC})$ that is more or less folklore. The second one, Appendix B, proves that *positive* nested TWA has the same expressive power than pebble TWA. It basically go again through the proof of Theorem 3 and check that the number of negations is preserved.

2. PRELIMINARIES

2.1 Trees

In this paper we consider two kinds of trees: sibling-ordered unranked trees and, as a special case, binary trees. We do not assume that the trees are finite. Thus, unless explicitly stated otherwise, all results hold both for finite trees and for infinite trees.

Fix a finite alphabet Σ. A Σ-tree is a partial mapping $t : \mathbb{N}^* \to \Sigma$ whose domain $dom(t)$ is 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 *context* is a tree with a designated leaf with no label called the *hole* of the context. If C is a context and t a tree, the *concatenation* of C and t, denoted Ct, is the tree constructed from C by replacing the hole of C with a copy of t. Conversely, given a tree t and a node x of t, the context $\mathsf{context}(t, x)$ is constructed by replacing $\mathsf{subtree}(t, x)$ with a hole.

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 Transitive closure logic

$\mathsf{FO(MTC)}$ 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. The parameter-free fragment of $\mathsf{FO(MTC)}$, which only allow for formulas $[\mathsf{TC}_{xy}\phi](u, v)$ where ϕ has no free variables besides x and y, is also known as FO^* [ten Cate 2006].

It is clear from the definition that $\mathsf{FO(MTC)}$ is a fragment of MSO.

To make life easier, we will use a normal form for $\mathsf{FO(MTC)}$-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. $\mathsf{FO(MTC^<)}$ is the fragment of $\mathsf{FO(MTC)}$ 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 4 *On binary trees, every* $\mathsf{FO(MTC)}$ *formula is equivalent to a* $\mathsf{FO(MTC^<)}$ *formula.*

The proof of Lemma 4 uses standard Ehrenfeucht-Fraïssé techniques, and is given in Appendix A. Lemma 4 is only an intermediate result: stronger normal form theorems for $\mathsf{FO(MTC)}$ on arbitrary trees will follow from our main results, see Section 8.

2.3 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 5 *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:*

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

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 I.

Table I. Semantics of Regular XPath(W) expressions

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

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

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

It is not difficult to see that Regular XPath(W) is a fragment of FO(MTC). Table II gives a linear translation $TR(\cdot)$ from Regular XPath(W) to FO(MTC). 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.

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

The proof of Lemma 6 is by induction and is left to the reader. A large part of the paper will be devoted to a translation in the other direction, i.e., from FO(MTC) to Regular XPath(W).

2.4 Nested tree walking automata

For proving our lower bounds on expressive power, 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*".

Table II. Translation from Regular XPath(W) to FO(MTC)

$TR_{u,v}(\alpha)$	$=$	$\exists z.(\forall z'.(z \leq z') \wedge TR^z_{u,v}(\alpha))$
$TR_u(\phi)$	$=$	$\exists z.(\forall z'.(z \leq z') \wedge TR^z_u(\phi))$
$TR^z_{u,v}(\delta)$	$=$	$(z \leq u, v) \wedge R_\delta(u, v)$ for $\delta \in \{\uparrow, \downarrow, \rightarrow, \leftarrow\}$
		where R_δ is the FO formula defining the appropriate successor relation.
$TR^z_{u,v}(.)$	$=$	$(z \leq u, v) \wedge u = v$
$TR^z_{u,v}(\alpha/\beta)$	$=$	$\exists w.(z \leq w \wedge TR^z_{u,w}(\alpha) \wedge TR^z_{w,v}(\beta))$
$TR^z_{u,v}(\alpha \cup \beta)$	$=$	$TR^z_{u,v}(\alpha) \vee TR^z_{u,v}(\beta)$
$TR^z_{u,v}(\alpha[\phi])$	$=$	$TR^z_{u,v}(\alpha) \wedge TR^z_v(\phi)$
$TR^z_{u,v}(\alpha^*)$	$=$	$[\mathsf{TC}_{x,y}\, TR^z_{x,y}(\alpha)](u, v)$
$TR^z_u(\sigma)$	$=$	$(z \leq u) \wedge P_\sigma(u)$
$TR^z_u(\langle\alpha\rangle)$	$=$	$(z \leq u) \wedge \exists v. TR^z_{u,v}(\alpha)$
$TR^z_u(\neg\phi)$	$=$	$z \leq u \wedge \neg TR^z_u(\phi)$
$TR^z_u(\phi \wedge \psi)$	$=$	$TR^z_u(\phi) \wedge TR^z_u(\psi)$
$TR^z_u(\phi \vee \psi)$	$=$	$TR^z_u(\phi) \vee TR^z_u(\psi)$
$TR^z_u(\mathsf{W}\phi)$	$=$	$\exists z((z = u) \wedge TR^z_u(\phi))$

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* (or 0-nested 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 $x \in dom(t)$.

Intuitively, a *nested tree walking automaton of rank* $k > 0$ is a non-deterministic tree automaton A that has finitely many sub-automata of rank less than k, and such that each transition of A may be conditional on whether some of the sub-automata do or do not have an accepting run from the current node, either in general or within the subtree rooted at the current node. Formally, for $k > 0$, a *nested tree walking automaton of rank* k (k-nested 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

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

Accepting runs are defined as before, except that the automaton can now take additional information into account when deciding its next move, given by the $\wp(I) \times \wp(I) \times \wp(I) \times \wp(I)$ in the definition of δ. The first $\wp(I)$ stands for a set of sub-automata A_i that are required to have an accepting run starting in the current node, in order for the transition to be allowed. The second $\wp(I)$ stands for a set of sub-automata A_i that are required *not* to have an accepting run starting in the current node. The third $\wp(I)$ stands for a set of sub-automata A_i that are required to have an accepting run in the subtree $\mathsf{subtree}(t, x)$ rooted at the current node x (starting from the root of this subtree). Finally, the fourth $\wp(I)$ stands for a set of sub-automata A_i that are required *not* to have an accepting run in the subtree $\mathsf{subtree}(t, x)$ rooted at the current node x (starting from the root of this subtree). Note that the notion of an accepting run is defined here by induction on the rank of the automaton.

The same notations are used for k-nested TWA as for 0-nested 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 establishes Theorem 3: Regular XPath(W) path expressions and nested TWA essentially define the same objects. It is proved using the classical translations between regular expressions and finite automata.

Lemma 7 *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, $A(t) = [\![\alpha]\!]^t$.

PROOF. (sketch) The first part of the lemma is proved by induction on the nesting depth of the automaton. TWA without nesting are known to be equivalent to caterpillar expressions, a fragment of Regular XPath(W) (cf. [Neven and Schwentick 2003]). Consider now a nested TWA of nesting depth $k > 0$. By induction we have a Regular XPath(W) expression corresponding to all of its sub-automata of depth less than $k - 1$. We can then apply again the translation of TWA into Regular XPath(W) formulas simulating each tests referring to some sub-automata using the filter operator $[\cdot]$, the W operator and the expressions obtained by induction.

The second part of the lemma is proved by induction on the Regular XPath(W) expression. More precisely, we show by simultanous induction that, for every Regular XPath(W) path expression α there is a nested TWA A_α, computable in linear time from α, such that for all trees t, $A_\alpha(t) = [\![\alpha]\!]^t$, and for every Regular XPath(W) node expression ϕ there is a nested TWA A_ϕ, computable in linear time from ϕ, such that for all trees t, $\{x \mid \exists y.(x, y) \in A_\phi(t)\} = [\![\phi]\!]^t$. The inductive step for filter operator $[\cdot]$ uses a non subtree-restricted sub-automata test, while the inductive step for the W operator uses a subtree-restricted sub-automata test. □

3. DEFINING K-ARY QUERIES USING TREE PATTERNS

We aim to translate $\mathsf{FO}(\mathsf{MTC})$ 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, following [Schwentick 2000], 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 17. In this section, we define unions of tree patterns, and show that they form a natural generalization of node and path expressions to k-ary queries with $k > 2$. In particular, tree patterns in at most two variables correspond to node or path expressions. In Section 4, we prove that every $\mathsf{FO(MTC)}$ formula is equivalent to a union of tree patterns in the same free variables. Together with Theorem 3, this implies Theorem 1.

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.

3.1 Downward node and path expressions

Before defining tree patterns, we first introduce a restricted class of path expressions, called *downward path expressions*, that can only move downwards in the tree and test node properties that depend only on the current subtree.

Definition 8 (Downward node and path 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.*

In the sequel we may be using $\downarrow$ in a downward path expression as a shortcut for $\downarrow_1 \cup \downarrow_2$ and $\downarrow^+$ as a shortcut for $\downarrow/\downarrow^*$. Downward path expressions can be seen as defining regular languages over an alphabet whose letters consist of Boolean combinations of node expressions. More precisely, fix any finite set of downward node expressions Φ, and let $\Sigma_\Phi = \wp(\Phi) \times \{r, 1, 2\}$. For any binary tree t with nodes $x < y$, we can associate to the pair (x, y) a finite word w_{xy} over Σ_ϕ describing the path from x to y in t, where each letter in the word encodes which node expressions from Φ are true at the corresponding node in the tree, and whether the node is the initial node x (indicated by r), a left child of another node on the path (indicated by 1) or a right child of another node on the path (indicated by 2). Now, for every downward path expression α built up from downward node expressions in Φ there is a regular word language L_α over Σ_Φ such that for all trees t and nodes $x < y$, $(x, y) \in [\![\alpha]\!]^t$ iff $w_{xy} \in L_\alpha$. Conversely, for every regular word language L over Σ_Φ there is a downward path expression α_L built up from downward node expressions in Φ, such that for all trees t and nodes $x < y$, $(x, y) \in [\![\alpha_L]\!]^t$ iff $w_{xy} \in L$. Both directions follow from the close syntactic correspondence between path expressions and regular expressions. It follows by standard automata theoretic results that the downward path expressions are closed under intersection and complementation.

Lemma 9 *On binary trees,*

(a) For every two downward path expressions α, β *there is a downward path expression* γ*, such that for all binary trees t,* $[\![\gamma]\!] = [\![\alpha]\!] \cap [\![\beta]\!]$.

(b) For every downward path expression α *there is a downward path expression* β *such that, for all binary trees t,* $[\![\beta]\!] = \{(x, y) \mid x \leq y\} \setminus [\![\alpha]\!]$.

Similarly, the following Lemma follows by standard automata theoretic results. We give an explicit proof for the sake of completeness.

Lemma 10 (Uniform splittings) *Every downward path expression α is equivalent to a finite union $\bigcup_{i\leq k}(\alpha_i/\beta_i)$ with α_i, β_i downward path expressions, such that for all trees t with nodes $x < y < z$, $(x,z) \in [\![\alpha]\!]$ if and only if, for some $i \leq k$, $(x,y) \in [\![\alpha_i]\!]$ and $(y,z) \in [\![\beta_i]\!]$.*

PROOF. The proof proceeds by induction on α. If α is of the form $\downarrow_i$ or ., then it has the uniform splitting $(\downarrow_i /.) \cup (./ \downarrow_i)$ or $./.$, respectively. If α of the form $\beta \cup \gamma$, then the uniform splitting of α is simply the union of the uniform splittings of β and γ. If α is of the form $\beta[\phi]$, then, by induction hypothesis, β has a uniform splitting $\bigcup_{i\leq n}(\alpha_i/\beta_i)$, and hence $\bigcup_{i\leq n}(\alpha_i/\beta_i[\phi])$ is a uniform splitting of α.

Next, suppose α is of the form $\alpha^{\mathrm{I}}/\alpha^{\mathrm{II}}$. By induction hypothesis, α^{I} and α^{II} have uniform splittings $\bigcup_{i\leq n}(\alpha_i^{\mathrm{I}}/\beta_i^{\mathrm{I}})$ and $\bigcup_{i\leq m}(\alpha_i^{\mathrm{II}}/\beta_i^{\mathrm{II}})$, respectively. Suppose that $x < y < z$ and $(x,z) \in [\![\alpha]\!]$. Then, by definition, there is a node u such that $(x,u) \in [\![\alpha^I]\!]$ and $(u,z) \in [\![\alpha^{\mathrm{II}}]\!]$. Clearly, either $x < y < u < z$ or $x < u < y < z$. It follows that α is equivalent to $\bigcup_{i\leq n}(\alpha_i^{\mathrm{I}}/(\beta_i^{\mathrm{I}}/\alpha^{\mathrm{II}})) \cup \bigcup_{i\leq m}((\alpha^{\mathrm{I}}/\alpha_i^{\mathrm{II}})/\beta_i^{\mathrm{II}})$, and that the latter is in fact a uniform splitting.

Finally, let α be of the form β^*. By the induction hypothesis, β has a uniform splitting $\bigcup_{i\leq n}(\alpha_i/\beta_i)$. It follows that α is equivalent to $. \cup (\beta^*/\bigcup_{i\leq n}(\alpha_i/\beta_i)/\beta^*)$ and hence to $(./.) \cup \bigcup_{i\leq n}((\beta^*/\alpha_i)/(\beta_i/\beta^*))$, which is a uniform splitting. □

3.2 Tree patterns

A tree pattern will consist of a tree-configuration for the relevant variables, plus a labeling of the edges in the tree with downward path expressions. We first give the definition of *tree configurations*. Unlike with binary trees as in Section 2.1, in the definition of tree configurations, 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 11 (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 12 *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.*

Definition 13 (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 edge of t a downward path expression. In case t consists of only one node (and therefore has no edges), then δ assigns to the node a downward node expression.*

A union of tree patterns *over a finite set of variables V is an expression of the form $P(V) = p_1(V) \cup \cdots \cup p_n(V)$, $n \geq 0$, with $p_1(V), \ldots, p_n(V)$ tree patterns over V.*

For convenience, if $p = (t, \lambda, \delta)$ is a tree pattern and v a variable, then we will often write $p(v)$ meaning $\lambda(v)$.

The semantics of tree patterns is based on the notion of an *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., $f(\varepsilon) = \varepsilon$, (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 14 (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 an 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 $dom(t) = \{\varepsilon\}$, then $\varepsilon \in [\![\delta(\varepsilon)]\!]^{t'}$

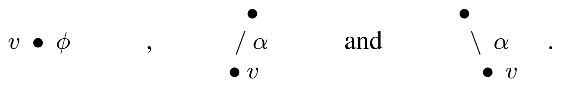
A tree pattern $p(v_1, \ldots, v_n)$ is satisfied *by a n-tuple of nodes $x_1, \ldots, x_n$ in a tree t (notation: $t \models p\,[x_1, \ldots, x_n]$) if there is a faithful embedding of p into t sending $\lambda(v_i)$ to x_i for all $i \leq n$.*[1] *A union of tree patterns $P(v_1, \ldots, v_n) = \bigcup_{i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

Since we have a notion of satisfaction for unions of tree patterns, we can also speak of *equivalence* between unions of tree patterns, node or path expressions, and FO(MTC)-formulas. Unions of tree patterns form a natural generalization of Regular XPath(W) node and path expressions to k-ary queries with $k > 2$. Indeed, as the following Lemma shows, unions of tree patterns in at most two free variables variables correspond to Regular XPath(W) node or path expressions.

Lemma 15 *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 three kinds of tree patterns in a single variable:



In the first case, the equivalent Regular XPath(W) node expression is simply $\neg\langle\uparrow\rangle \wedge \phi$, in the second case it is $\langle(\alpha \cap (\downarrow_1/\downarrow^*))^{-1}[\neg\langle\uparrow\rangle]\rangle$, where α^{-1} is the converse of α, and in the third case it is $\langle(\alpha \cap (\downarrow_2/\downarrow^*))^{-1}[\neg\langle\uparrow\rangle]\rangle$. Note that we use the fact that the downward path expressions are closed under intersection, cf. Lemma 9. It follows that every *union* of tree patterns in v is equivalent to a *disjunction* of such Regular XPath(W) node expressions. □

Furthermore, unions of tree patterns still form a fragment of FO(MTC). Indeed, the translation from node and path expressions to FO(MTC) we gave in Lemma 6 can be extended to tree patterns, showing that every tree pattern is equivalent to a conjunctive

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

query over FO(MTC)-translations of downward node or path expressions. We will show in the next section that there is also a translation in the other direction, and hence unions of tree patterns have exactly the same expressive power as FO(MTC)-formulas. Theorem 1 then follows immediately, since, by Lemma 15 unions of tree patterns in at most two variables are equivalent to Regular XPath(W) node or path expressions which in turn are equivalent to nested TWA by Lemma 7.

4. THE TRANSLATION FROM FO(MTC) TO UNIONS OF TREE PATTERNS

We now turn to the translation from FO(MTC) formulas to unions of tree patterns. In fact, we make use of the normal form fragment FO(MTC$^<$) of FO(MTC) provided by Lemma 4. We show that every atomic formula of FO(MTC$^<$) is equivalent to a union of tree patterns, and next that unions of tree patterns are closed under all desired operations: intersection, complement, existential quantification and the restricted form of transitive closure that is part of FO(MTC$^<$). As in the previous section we restrict attention to binary trees, the generalization to arbitrary unranked ordered trees will be presented in Section 7.

4.1 Translation of atomic formulas and FO connectives

The following proposition provides the base case for our inductive translation.

Proposition 16 *Every atomic* FO(MTC$^<$) *formula is equivalent on binary trees to a union of tree pattern in the same variables.*

PROOF. The atomic formulas of FO(MTC$^<$) are of the formulas of the form $u < v$, $u \prec v$, $P_\sigma(u)$, $u = v$ and $\top$. The atomic formula $u < v$ is equivalent to the union of all tree patterns $p = (t, \lambda, \delta)$ in u, v with $p(u) < p(v)$ where δ assigns to each edge the downward path expression $\downarrow^+$. Note that, by Fact 12, there are only finitely many such tree patterns. The other cases are treated similarly. □

We show that union of tree patterns have the same expressive power FO(MTC$^<$) by showing that union of tree patterns are closed under all first-order operations and under transitive closure. As one would expect, the difficult case is closure under transitive closure. The following sequence of propositions prove closure under all first-order operations.

Proposition 17 (Expansion) *For each union of tree patterns $P(u_1, \ldots, u_n)$ there is a union of tree patterns $P'(u_1, \ldots, u_n, v)$ such that for all binary trees t and nodes $x_1, \ldots, x_n, y$, we have that $t \models P'\ [x_1, \ldots, x_n, y]$ iff $t \models P\ [x_1, \ldots, x_n]$.*

PROOF. For each tree pattern $p(u_1, \ldots, u_n)$ in $P(u_1, \ldots, u_n)$, we consider all possible placements of v relative to $u_1, \ldots, u_n$ and their least common ancestors. There are four cases: (i) v coincides with an existing node of the tree pattern, (ii) v is a descendant of one of the leafs of the tree pattern, (iii) v is located on the path between two nodes of the tree pattern, or (iv) v is a descendant of a node on the path between two nodes of the tree pattern. Correspondingly, we construct a union of tree patterns, one for each possibility. In the first case, we simply take the tree pattern $p(u_1, \ldots, u_n)$ and extend the variable mapping to send v to the corresponding node. In the second case, we extend $p(u_1, \ldots, u_n)$ with an extra node and edge, where the edge is labeled by the trivial downward path expression $\downarrow^+$, and the variable mapping is extended to send v to the new node. In the third case, one of the edges of $p(u_1, \ldots, u_n)$ is split in two, and the variable mapping is extended to send v to the newly added intermediate node. Lemma 10 is used to split the downward path expression of the original edge into two parts. Finally, in the last case, one of the edges of $p(u_1, \ldots, u_n)$ is split in two and also an extra node is added as a child of the newly added intermediate node, Lemma 10 is again used to split the downward path expression of the original edge into two parts, and the second new edge is labeled by the trivial downward path expression $\downarrow^+$. All tree patterns obtained in this way, from each $p(u_1, \ldots, u_n)$ in $P(u_1, \ldots, u_n)$, are collected into a union of tree patterns $P'(u_1, \ldots, u_n, v)$. It follows from immediately from the construction that $t \models P'\ [x_1, \ldots, x_n, y]$ iff $t \models P\ [x_1, \ldots, x_n]$. □

Proposition 18 (Disjunction) *For all unions of tree patterns $P_1(u_1, \ldots, u_n)$ and $P_2(u_1, \ldots, u_n)$ there is a union of tree patterns $P(u_1, \ldots, u_n)$ such that for all binary trees t with nodes $x_1, \ldots, x_n$, $t \models P\ [x_1, \ldots, x_n]$ if and only if either $t \models P_1\ [x_1, \ldots, x_n]$ or $t \models P_2\ [x_1, \ldots, x_n]$.*

PROOF. It suffices to take $P(u_1, \ldots, u_n) = P_1(u_1, \ldots, u_n) \cup P_2(u_1, \ldots, u_n)$. □

Proposition 19 (Conjunction) *For all unions of tree patterns $P_1(u_1, \ldots, u_n)$ and $P_2(u_1, \ldots, u_n)$ there is a union of tree patterns $P(u_1, \ldots, u_n)$ such that for all binary trees t with nodes $x_1, \ldots, x_n$, $t \models P\ [x_1, \ldots, x_n]$ if and only if $t \models P_1\ [x_1, \ldots, x_n]$ and $t \models P_2\ [x_1, \ldots, x_n]$.*

PROOF. Since disjunction distributes over conjunction, it suffices to show that the conjunction of two tree patterns $p_1(u_1, \ldots, u_n), p_2(u_1, \ldots, u_n)$ is definable by a tree pattern $p(u_1, \ldots, u_n)$. The proof proceeds by a case distinction. If the underlying tree configuration of $p(u_1, \ldots, u_n)$ and $p'(u_1, \ldots, u_n)$ is different (i.e., non-isomorphic), then clearly the two cannot be satisfied at the same time. Hence, we can pick $p(u_1, \ldots, u_n)$ to be any inconsistent tree pattern, i.e., a tree pattern in which one of the edges is labeled by .[W⊥]. If, on the other hand, the underlying tree configurations of $p_1(u_1, \ldots, u_n)$ and $p_2(u_1, \ldots, u_n)$ are isomorphic, then $p(u_1, \ldots, u_n)$ can be defined as the tree pattern whose underlying tree configuration is the one of p_1 and p_2 and where each edge is labeled by the intersection of the downward path expressions labeling the edge in $p_1(u_1, \ldots, u_n)$ and $p_2(u_1, \ldots, u_n)$. Note that we use here Lemma 9. □

Proposition 20 (Negation) *For each union of tree pattern $P(u_1, \ldots, u_n)$ there is a union of tree patterns $P'(u_1, \ldots, u_n)$ such that for all binary trees t with nodes $x_1, \ldots, x_n$, $t \models P'\ [x_1, \ldots, x_n]$ if and only if $t \not\models P\ [x_1, \ldots, x_n]$.*

PROOF. Since we already established closure under conjunction and disjunction it suffices to consider negations of a single tree pattern. If a tree pattern $p(u_1, \ldots, u_n)$ fails to be satisfied by a sequence of elements $x_1, \ldots, x_n$ in a tree *t*, it is either because the relative positions of $x_1, \ldots, x_n$ in the tree do not conform to the underlying tree configuration of $p(u_1, \ldots, u_n)$, or because one of the edge labels of $p(u_1, \ldots, u_n)$ is not satisfied. Correspondingly, we define $P'(u_1, \ldots, u_n)$ to be the union of the following (finitely many) tree patterns:

—for each tree configuration for $u_1, \ldots, u_n$ that is different than the one underlying p, take the tree pattern based on this configuration, in which each edge is labeled by the trivial downward path expression $\downarrow^+$.

—for each edge e in the underlying tree configuration of $p(u_1, \ldots, u_n)$, take a copy of $p(u_1, \ldots, u_n)$ in which e is labeled by the complement of the downward path expression labeling e in $p(u_1, \ldots, u_n)$, while all other edges are labeled by the trivial downward path expression $\downarrow^+$. Note that we use here Lemma 9.

It is easily seen that $t \models P'\ [x_1, \ldots, x_n]$ if and only if $t \not\models P\ [x_1, \ldots, x_n]$. □

Proposition 21 (Existential quantification) *For each union of tree patterns $P(u_1, \ldots, u_n, v)$ there is a union of tree patterns $P'(u_1, \ldots, u_n)$ such that for all binary trees t with nodes $x_1, \ldots, x_n$, $t \models P'\ [x_1, \ldots, x_n]$ if and only if there is a node y such that $t \models P\ [x_1, \ldots, x_n, y]$.*

PROOF. We start by taking each tree pattern in $P(u_1, \ldots, u_n, v)$ and removing the label v. The tree patterns obtained in this way are in general not well formed: they may contain nodes that are neither the image of a variable, nor the least common ancestor of nodes that are the image of a variable, nor the root. However, any such redundant nodes can be eliminated from the tree pattern by applying one or more times the rewrite rules in Figure 1, where the redundant node is indicated by $\circ$ and symmetric cases of the rules are left out. Note that we use here again Lemma 9. □

We are left with the task to show that the unions of tree patterns are closed under the transitive closure operator, which we will prove next.

4.2 Closure under TC$^<$

Our aim in this section is to show that the unions of tree patterns are closed under transitive closure. In fact, by Lemma 4 it is enough to show closure under transitive closure for unions of tree patterns of a restricted shape. More precisely, it is enough to prove Proposition 23 below.

Definition 22 *A* 4-pattern *is a tree pattern $p(u, v, x, y)$ satisfying $p(u) < p(v)$, $p(u) \leq p(x), p(y)$ and $p(v) \not< p(x), p(y)$.*

Proposition 23 *Let $P(u, v, x, y)$ be a union of 4-patterns. Then there is a union of tree patterns $P'(u, v, x', y')$ such that for all binary trees t and nodes u, v, x', y', $t \models P'(u, v, x', y')$ iff (x', y') belongs to the transitive closure of the relation $\{(x, y) \mid t \models P(u, v, x, y)\}$.*

We first prove a special case of Proposition 23. We call 4-pattern *linear* if x and y lie on the path from u to v, i.e., if there are no forks in the pattern.

Lemma 24 *For each union of linear 4-patterns $P(u, v, x, y)$ there is a union of linear 4-patterns $P'(u, v, x', y')$ such that for all trees t and nodes u, v, x', y', $t \models P'(u, v, x', y')$ iff (x', y') belongs to the transitive closure of the relation $\{(x, y) \mid t \models P(u, v, x, y)\}$.*

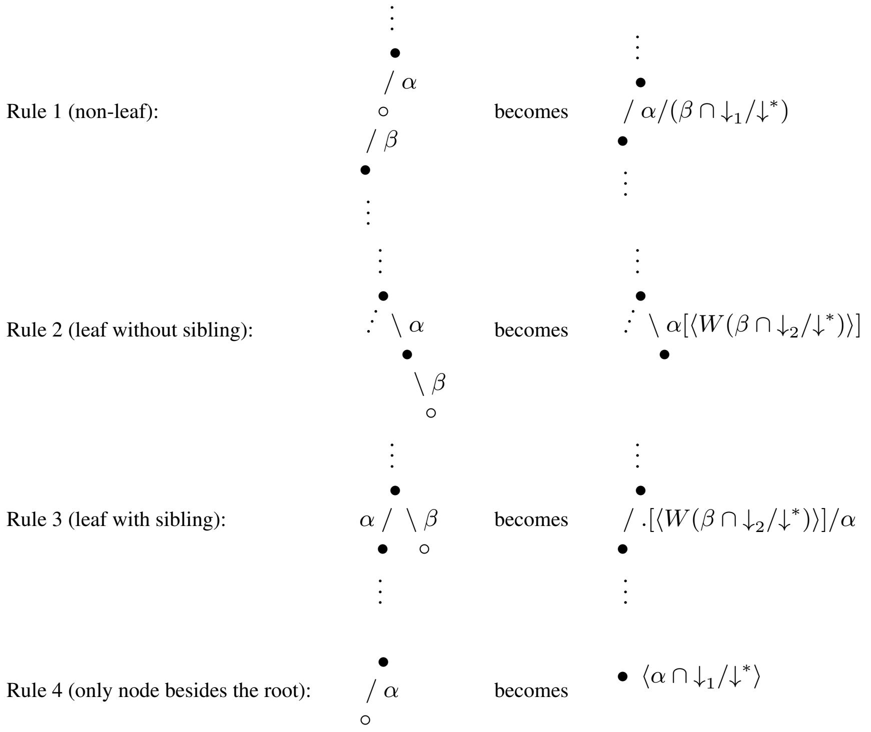

Rule 1 (non-leaf):
$/ \alpha$
$/ \beta$
becomes
$/ \alpha/(\beta \cap \downarrow_1/\downarrow^*)$
Rule 2 (leaf without sibling):
$\setminus \alpha$
$\setminus \beta$
becomes
$\setminus \alpha[\langle W(\beta \cap \downarrow_2/\downarrow^*)\rangle]$
Rule 3 (leaf with sibling):
$\alpha / \ \setminus \beta$
becomes
$/ .[\langle W(\beta \cap \downarrow_2/\downarrow^*)\rangle]/\alpha$
Rule 4 (only node besides the root):
$/ \alpha$
becomes
$\langle \alpha \cap \downarrow_1/\downarrow^* \rangle$

Fig. 1. Rules for eliminating redundant nodes from a tree pattern

PROOF. As described in Section 3.1, downward path expressions can be seen as defining regular string languages over an alphabet consisting of Boolean combinations of downward node expressions and sibling order information. More precisely, if Φ is a finite set of downward node expressions and $\Sigma = \wp(\Phi) \times \{r, 0, 1\}$, then every downward path expressions with downward node tests from Φ can be seen as defining a regular string language over the alphabet Σ. It follows that every union of linear 4-patterns $P(u, v, x, y)$ defines, in the same way, a 4-ary MSO query on finite strings over the alphabet Σ (where the strings correspond to paths in trees from the root to v), and therefore also $[TC_{xy}P(u, v, x, y)](x', y')$.

Let $\phi(u, v, x', y')$ be the MSO formula on finite Σ-strings thus obtained from $[TC_{xy}P(u, v, x, y)](x', y')$. It is well known that, on finite strings, every MSO definable n-ary query $\psi(x_1, \ldots, x_n)$ is equivalent to a disjunction of formulas of the form

$$\begin{aligned}&x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)} \wedge \\ &\psi_{[\ldots,1]}(x_{\pi(1)}) \wedge \psi_{[1,2]}(x_{\pi(1)}, x_{\pi(2)}) \wedge \cdots \wedge \psi_{[n-1,n]}(x_{\pi(n-1)}, x_{\pi(n)}) \wedge \psi_{[n,\ldots]}(x_{\pi(n)})\end{aligned}$$

where $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a permutation and the "segment formulas" $\psi_{[\ldots,\pi(1)]}$, $\ldots$, $\psi_{[\pi(n),\ldots]}$ express that the corresponding segment of the string belongs to some regular language (over Σ). In particular, we may assume that $\phi(u, v, x', y')$ is of this form (with $n = 4$).

By Kleene's theorem, we can translate each segment formula to a regular expression over Σ, which can again be seen as a downward path expression with downward node tests from Φ.

All together, this shows that we can translate $\phi(u, v, x', y')$ to a union of linear 4-patterns $P'(u, v, x', y')$, which then defines $[TC_{xy}P(u, v, x, y)](x', y')$. $\square$

We also know that the unions of linear 4-patterns are closed under union and composition (i.e., for all unions of linear 4-patterns $P_1(u, v, x, y), P_2(u, v, x, y)$ there is a union of linear 4-patterns $P(u, v, x, y)$ such that for all trees t and nodes u, v, x, y, $t \models P(u, v, x, y)$ iff there is a node z such that $t \models P_1(u, v, x, z)$ and $t \models P_2(u, v, z, y)$). Closure under union is trivial, and closure under composition follows from Proposition 19 and Proposition 21 (inspection of the proof of these propositions shows that the resulting tree patterns are indeed again linear 4-patterns). In other words, the unions of linear 4-patterns are closed under all three regular operations: transitive closure, union and composition. This immediately gives us the following:

Lemma 25 *Fix a finite set of linear 4-patterns $P(u, v, x, y)$. For every regular language L over the alphabet $P(u, v, x, y)$, there is a union of linear 4-patterns $P'(u, v, x', y')$ such that, for all trees t and nodes u, v, x', y', the following are equivalent:*

(*1*) $t \models P'(u, v, x', y')$,

(*2*) *there is a word $w = p_1 \ldots p_n \in L$, where each $p_i(u, v, x, y)$ is a linear 4-pattern from $P(u, v, x, y)$, and there is a sequence of nodes $x_1, \ldots, x_{n+1}$, such that $x_1 = x'$, $x_{n+1} = y'$, and for all $i \leq n$, $t \models p_i(u, v, x_i, x_{i+1})$.*

We are ready to proceed with the proof of Proposition 23.

PROOF OF PROPOSITION 23. Let $P(u, v, x, y)$ be any finite set of 4-patterns. We classify these 4-patterns $p(u, v, x, y) \in P$ into three groups: those in which $lca(x, v) < lca(y, v)$, those in which $lca(y, v) < lca(x, v)$ and those in which $lca(x, v) = lca(y, v)$. We call these *downward*, *upward* and *subtree* 4-patterns, reflecting the fact that walking from x to y in the tree pattern involves downward, upward or no movement along the path from u to v, and we denote by $P_{\downarrow}, P_{\uparrow}, P_{subtree}$ the set of downward, upward and subtree 4-patterns in P. Equivalently, a 4-pattern is downward if it is of the form in Figure 2(a) where the sibling order may be changed and some of the edges may be contracted (but not the edge from $lca(x, v)$ to $lca(y, v)$), it is upward if it is of the form in Figure 2(b) where the sibling order may be changed and some of the edges may be contracted (but not the edge from $lca(y, v)$ to $lca(x, v)$), and it is a subtree 4-pattern if it is of the form in Figure 2(c), where the sibling order may be changed and some of the edges may be contracted. The node $lca(y, v)$ has been highlighted in Figure 2 to anticipate the fact that it will play a special role in the construction below, and each edge is marked by a label α_i for ease of reference.

Intuitively patterns of the form $P_{\downarrow}, P_{\uparrow}$ can be reduced to Lemma 25 while patterns of the form $P_{subtree}$ needs to be combined as a downward node expression. In order to achieve the latter, for $p \in P_{subtree}$, we will use α^{p}_{xy} as a shorthand for the path expression $\alpha_3^{-1}[\langle\alpha_2^{-1}[\mathsf{root}]\rangle]/\alpha_4$, which describes a walk from the node x to the node y inside the subtree below $lca(x, v)$ (= $lca(y, v)$), where α_2, α_3 and α_4 are the path expression of the pattern p according to Figure 2. For a set $S \subseteq P_{subtree}$ we will use α^{S}_{xy} as a shorthand for

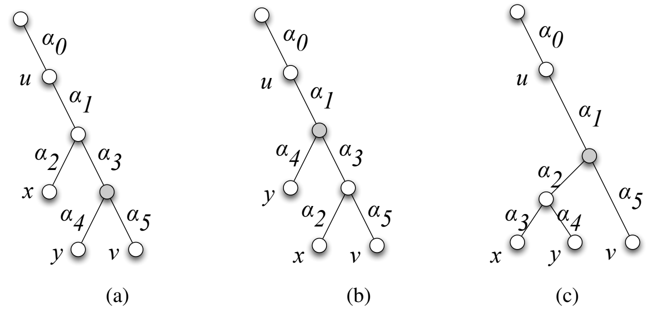


Fig. 2. Downward, upward, and subtree 4-patterns

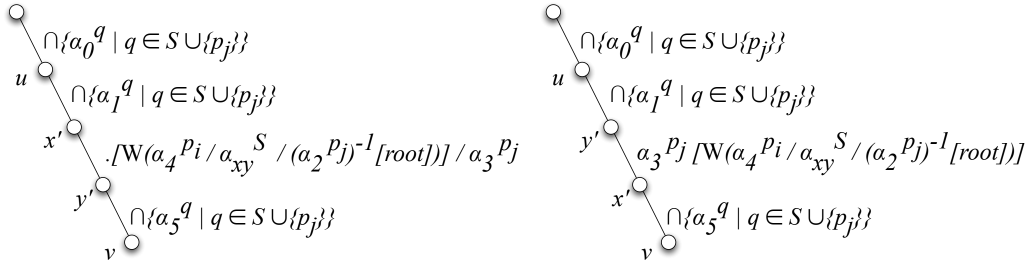
the path expression $(\bigcup_{p \in S} \alpha^p_{xy})^*$ which describes a path in the subtree below $lca(x,v) = lca(y,v)$ where each step satisfies the constraints below $lca(x,v)$ of a tree pattern in S.

We have to construct a union of tree patterns $P'(u,v)$ such that for all trees t and nodes u, v, it is the case that $t \models P'(u,v)$ iff (u,v) belongs to the transitive closure of the binary relation $\{(x,y) \mid t \models P(u,v,x,y)\}$. We will use Lemma 25 for this, by constructing a non-deterministic finite automaton over an alphabet consisting of linear 4-patterns. More precisely, define define an NFA $A = (\Sigma, Q, \delta, q_0, F)$ as follows:

—The set of states Q contains a state q_p for each $p(u,v,x,y) \in P_\downarrow \cup P_\uparrow$, plus the initial state q_0. Intuitively, the state q_p represents a situation, during the walk from u to v, where one is currently in a node matching the $lca(y,v)$-node of p, having just arrived there from a node matching the node x of p.

—The alphabet Σ consists of finitely many linear 4-patterns $p_{i,j,S}$ one for each $p_i, p_j \in P_\downarrow \cup P_\uparrow$ and $S \subseteq P_{subtree}$. Intuitively, $p_{i,j,S}(u,v,x',y')$ expresses the following movement in the tree: *starting in a node matching the $lca(y,v)$-node of p_i, move to a node matching the y-node of p_i. Next, move around in the subtree below the node $lca(y,v)$ by making any number of executions of subtree patterns in S. Finally, having arrived at a point that can be matched with the x-node of p_j, move to the node corresponding to the $lca(x,v)$-node of p_j and then to node corresponding to the $lca(y,v)$-node of p_j.* The precise definition of $p_{i,j,S}$ is as in Figure 3. Note that the set S need to be guessed in advance as, for instance, the constraint induced by a pattern in S between the root and u need to be verified from the beginning.

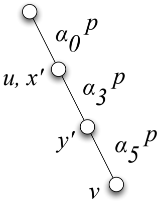
—For each pairs $p_i, p_j \in P_\downarrow \cup P_\uparrow$ and for each subset $S \subseteq P_{subtree}$, there is a transition from the state q_{p_i} to the state q_{p_j} labeled by the linear 4-pattern $p_{i,j,S}(u,v,x',y')$ (which belongs to the alphabet Σ). Furthermore, there is a transition from the initial state q_0 to every state $q_{p(u,v,x,y)}$ for $p \in P_\downarrow$ such that $p(u) = p(x)$, where the transition is labeled by the following linear 4-pattern (assuming that p is of the form in Figure 2(a) but with the edges between x and u contracted):


$\cap\{\alpha_0{}^q \mid q \in S \cup \{p_j\}\}$
u
$\cap\{\alpha_1{}^q \mid q \in S \cup \{p_j\}\}$
x'
$.[\mathrm{W}(\alpha_4{}^{p_i} / \alpha_{xy}{}^S / (\alpha_2{}^{p_j})^{-1}[root])] / \alpha_3{}^{p_j}$
y'
$\cap\{\alpha_5{}^q \mid q \in S \cup \{p_j\}\}$
v
$\cap\{\alpha_0{}^q \mid q \in S \cup \{p_j\}\}$
u
$\cap\{\alpha_1{}^q \mid q \in S \cup \{p_j\}\}$
y'
$\alpha_3{}^{p_j}[\mathrm{W}(\alpha_4{}^{p_i} / \alpha_{xy}{}^S / (\alpha_2{}^{p_j})^{-1}[root])]$
x'
$\cap\{\alpha_5{}^q \mid q \in S \cup \{p_j\}\}$
v

(a) The linear 4-pattern $p_{i,j,S}$ in the case where $p_j \in P_\downarrow$ is of the form depicted in Figure 2(a)

(b) The linear 4-pattern $p_{i,j,S}$ in the case where $p_j \in P_\uparrow$ is of the form depicted in Figure 2(b)

Fig. 3. The linear 4-patterns $p_{i,j,S}$.



—The final states are all those states $q_{p(u,v,x,y)}$ where $p(y) = p(v)$, as well as the initial state q_0.

It is not hard to see that, the union of linear 4-patterns obtained from this automaton by means of Lemma 25 defines the transitive closure of the relation $\{(x, y) \mid t \models P(u, v, x, y)\}$. $\Box$

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 that we use is the same tree language used in [Bojańczyk et al. 2006] to separate the regular tree languages from those recognized by TWA with pebbles. Actually, not much has to be done for the proof as the construction used in [Bojańczyk et al. 2006] contains most of what is needed to show that nested TWA fail to capture all regular tree languages.

The separating language uses only finite and binary trees, two properties expressible in FO(MTC). We therefore show the stronger result that FO(MTC) fail to recognize all regular tree languages over finite and binary trees.

The goal is to construct by induction tree languages L_k such that L_k is not recognized by any $(k-1)$-nested TWA. The separating language is then essentially the union over all k of the L_k. The basis of the induction relies on the regular tree language that is not recognized by any TWA as exhibited in [Bojańczyk and Colcombet 2008]. For technical reasons we actually need a slightly more powerful basis hypothesis and therefore a language slightly more complex as introduced in [Bojańczyk et al. 2006].

The precise description of the regular tree language not recognized by any TWA will not be important here but we will make use of some of its properties that we now describe.

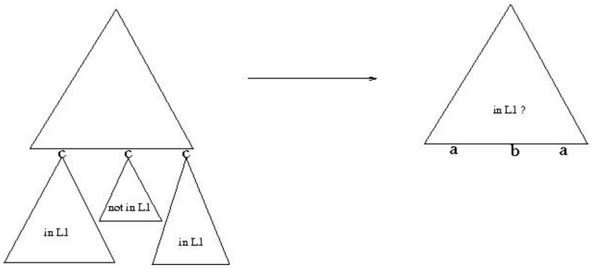


Fig. 4. The construction of L_2. The left tree is 2-separated and belongs to L_2 iff the right one is in L_1.

This tree language is a set of *quasi-blank-trees*. Those are trees over the alphabet $\{\mathbf{a}, \mathbf{b}\}$ in which the label **a** occurs only at leaves. Hence essentially all the useful information about the tree is located into its leaves that may be labeled either with **a** or with **b**. Typically a language over quasi-blank-trees express properties about the structure induced by the **a**-leaves, like the relative positions of their common ancestors and such. The idea is that this structure is difficult to compute for TWA as they "loose their way" in the middle of the tree as it contains only **b**-nodes. The language L_1 contains only quasi-blank-trees and is such that no TWA can recognize L_1 or the language consisting of the quasi-blank-trees not in L_1.

We now give the intuition that underlines the construction of L_2 from L_1 as it already contains all the ideas of the general inductive steps. In order to define L_2 we consider trees that are 2-*separated*. Those are trees that can be decomposed as (i) a top part, that contains only **b**-nodes, (ii) a maximal antichain of **c**-nodes that separate completely the top part from the rest of the tree and, (iii) each such **c**-node is the root of a quasi-blank-tree, denoted **c**-subtree in the sequel. A 2-separated tree belongs to L_2 if after replacing each **c**-subtree with a node of label **a** if the subtree belongs to L_1 or with a node of label **b** otherwise, the resulting tree belongs to L_1. The construction is depicted in Figure 4.

Intuitively a 1-nested TWA cannot recognize L_2 for the following reasons: While the automata is in the top part, it cannot infer any relevant information from its nested subruns because, by induction, those cannot tell whether the subtree rooted at a **c**-node belongs or not to L_1 and hence whether this node should be considered as a leaf of label **a** or of label **b**. Hence at an inner node of the top part it behaves as a TWA and hence, by induction, cannot tell whether this top part is in L_1 or not.

There is one difference however between the behavior of the 1-nested TWA and the behavior of the TWA: Even if the nested subruns cannot say whether a **c**-node should be interpreted as a **a** or as a **b**, they can perform several computations that a TWA cannot do, like testing whether the subtree rooted at the current node has a even number of node, or testing whether the path from the current node to the root has odd length. Hence, on the top part of the tree, a 1-nested TWA behaves like a TWA extended with extra tests that are *blind* in the sense that they cannot see the labels of the nodes. We formalize this concept of blind tests using the notion of an oracle-TWA. In order to make the induction work we thus need to choose L_1 so that no oracle-TWA can recognize it. Such a tree language was constructed in [Bojańczyk et al. 2006].

We will first recall the results on oracle-TWA from [Bojańczyk et al. 2006] that we use, then formally define the tree languages L_k, and finally state and prove the inductive hypothesis.

5.1 Oracle-TWA

Oracle-TWA extend the model of nested TWA by allowing MSO tests, but only on the structure of the tree, without any access to the label of the nodes of the tree. We recall the relevant definitions from [Bojańczyk et al. 2006].

A *structure oracle* $\mathcal{O}$ is a (parallel) deterministic bottom-up tree automaton that is label invariant. That is, any two trees that have the same nodes get assigned the same state by $\mathcal{O}$. Therefore, a structure oracle is defined by its state space Q, an initial state $s_0 \in Q$ and a transition function $Q \times Q \to Q$. We write $t^{\mathcal{O}}$ for the state of $\mathcal{O}$ assigned to a tree t. This notation is extended to contexts: given a context C, $C^{\mathcal{O}} : Q \to Q$ is defined by $C^{\mathcal{O}}(q) = (C[t])^{\mathcal{O}}$, where t is some tree with $q = t^{\mathcal{O}}$. (All states are assumed reachable.)

For a tree t and a node v of t, and a structure oracle $\mathcal{O}$, the *structural $\mathcal{O}$-information* about (t, v) is the pair

$$(\mathsf{context}(t, v)^{\mathcal{O}}, \mathsf{subtree}(t, v)^{\mathcal{O}}) \in Q^Q \times Q \ .$$

It should be noted that the result of any unary query expressible in monadic second-order logic which does not refer to the label predicates can be calculated based on the structural $\mathcal{O}$-information for some $\mathcal{O}$ (and vice-versa). Since the only type of oracles we use in this paper are structure oracles, we just write oracle from now on.

An *oracle-TWA* (OTWA) is a TWA extended by a structure oracle $\mathcal{O}$. The only difference to a usual TWA is in the definition of the transition relation which also takes into account the structural $\mathcal{O}$-information about (t, v), when at a node v of t. The *size* of a OTWA is defined as the total-size of the TWA part plus the number of states of the oracle part.

The *behavior* of an OTWA, or a nested TWA, in a tree is, informally, the set of state transitions corresponding to root-to-root runs of the automaton in the tree. More formally we define the behavior $b_A(t)$ of an OTWA or nested TWA A in a tree t as a subset of $Q \times Q$, where Q is the set of states of A, such that the pair (p, q) is in $b_A(t)$ iff there exists a run of A in t, starting at the root in state p, ending at the root in state q. Similarly we define the behavior of A in a context C by looking at the runs of A starting and ending at the port of C.

The following result of [Bojańczyk et al. 2006] plays an important role in the definition of the languages L_k ($k > 1$): it provides not only for the base case L_1, but it is also used in the inductive definition of L_k with $k > 1$.

Proposition 26 ([Bojańczyk et al. 2006]) *There exists a language of finite quasi-blank-trees L_1 such that for any number m, there exists a tree $s \in L_1$ and a tree $t \notin L_1$, such that for any OTWA A of size bounded by m, the behavior of A on s is the same as the behavior of A on t.*

5.2 Inductive definition of the tree languages L_k

A *k-separated tree* is a tree defined by induction, the base case being the 1-separated trees which are the quasi-blank-trees. A k-separated tree can be decomposed as (i) a top part, that contains only **b**-nodes, (ii) a maximal antichain of **c**-nodes that separate completely the top part from the rest of the tree and, (iii) each such **c**-node is the root of a $(k-1)$-separated tree, denoted by **c**-subtree in the sequel.

Definition 27 *L_k ($k > 1$) is the set of k-separated trees for which the following holds: when each maximal* **c***-subtree is replaced by a node labeled* **a** *in case the subtree belongs to L_{k-1} and with a node labeled* **b** *otherwise, the resulting tree belongs to L_1.*

Given k and a $(k-1)$-nested TWA A, we want to construct two trees s_k and t_k such that A accepts s_k iff A accepts t_k and $s_k \in L_k$ while $t_k \notin L_k$. We shall do this by induction replacing in a sufficiently big tree in L_k several subtrees s_{k-1} by subtrees t_{k-1} in a way that the resulting tree is no longer in L_k. We therefore need a inductive hypothesis stronger than just the fact that A has the same behavior on s_{k-1} and t_{k-1}: we need that A cannot distinguish Cs_{k-1} from Ct_{k-1} for all contexts C such that Cs_{k-1} is a k-separated tree. The technical difficulty is that the behavior of A in s_{k-1} and in t_{k-1} does depend on C because of its nested subcomputations.

We relativize the notion of behavior to a context C. Given a tree t, a context C and a nested TWA A, the *behavior of A in t relative to C* is the set of pairs of states (p, q) such that there is a run of A in Ct, starting at the root of t in state p, ending at the root of t in state q and such that A never leaves t during the run. Note that even if A does not go outside t, its nested subcomputations may, and the behavior of A in t relative to C depends on C. Similarly we define the notion of behavior of A in C relative to t, by considering this time runs of A that starts and ends at the hole of C, stays inside the context C, but its nested subcomputations may visit t.

The relative behavior of A in t is therefore essentially a function from a context C to a behavior. It will be important for us that this function can be finitely represented and computed by a tree automata. Consider for instance the case of a 1-nested TWA A. At any node of t, the transitions of A depend by definition on the existence of runs of some TWA. Those runs may occur in t but also in the context C. But for the part occurring in C it is sufficient to know the behavior of those TWA in C, a finite amount of information! The extension to arbitrary k is a bit tricky: A relative behavior of a k-nested TWA in t depends on the behavior of $(k-1)$-nested TWA in C, but this depends on the relative behaviors of $(k-2)$-nested TWA in t... We note that the finite presentation of relative behavior we use below is different from the one used in [Bojańczyk et al. 2006] for TWA with pebbles. The reason is that the relative behavior of a TWA using $(k-2)$ pebbles depends on the positions of pebbles $k-1$ and k in t, but in our case, there is no such pebbles, hence it depends only on C and t. Therefore the finite presentation of relative behaviors is simpler for nested TWA than for TWA with pebbles.

Let A be a nested TWA of rank k and let $(A_j)_{j\in J}$ be all the nested TWA (of rank strictly less than k) occurring in the inductive definition of A. Let Q be the set of states of A and $(Q_j)_{j\in J}$ be the set of states of A_j. The *basic relative behavior type* of A on a tree t is a function τ from $\Pi_{j\in J}\wp(Q_j \times Q_j)$ to a subset of $Q \times Q$ such that for any context C, if for all $j \in J$, α_j is the behavior of A_j in C relative to t, then $\tau((\alpha_j)_{j\in J})$ is the behavior of A in t relative to C. In the degenerated case $k = 0$ this is simply a subset of $Q \times Q$. The *relative behavior type* of A on t is the set $\{\tau, (\tau_j)_{j\in J}\}$ where τ is the basic relative behavior type of A on t while τ_j is the basic relative behavior type of A_j on t.

We are now ready to state and prove our induction hypothesis:

Proposition 28 *For any numbers m and $k \geq 1$, there exists a tree $s_k \in L_k$ and a tree $t_k \notin L_k$, such that for any $(k-1)$-nested TWA A of size bounded by m, the relative behavior type of A in s_k is the same as the relative behavior type of A in t_k.*

PROOF. The proof is by induction on k. The base case of $k = 1$ is a special case of Proposition 26 with a trivial structure oracle. Assume now that $k > 1$. Fix m and assume by induction that s_{k-1} and t_{k-1} satisfies the statement of the proposition for m and $k-1$. Assume also by induction that s_1 and t_1 satisfies the statement of the proposition for $f(m)$ and $k = 1$ for an appropriate function f that will be apparent from the proof below. Construct s_k as follows. Consider s_1 and replace each leaf of label **a** with a copy of s_{k-1} after relabeling its root with **c**, and each leaf of label **b** with a copy of t_{k-1} after relabeling its root with **c**. Similarly we construct t_k starting from t_1. Consider a nested TWA A of size smaller than m and of rank smaller of equal to $k-1$. We show that the relative behavior type of A in s_k is the same as its relative behavior type in t_k. This last assertion is proved by induction on the rank of A. We only prove here the most difficult case when the rank of A is exactly $k-1$. The cases when the rank is strictly smaller than $k-1$ are special (and simpler) cases of this one.

Assume $(A_j)_{j\in J}$ are all the nested TWA occurring in the inductive definition of A, and let l_j be the rank of A_j and let σ_j be the basic relative behavior type of A_j on s_k and τ_j be the basic relative behavior type of A_j on t_k. Similarly let σ and τ be the basic relative behavior type of A_j on s_k and t_k. Because A_j has a rank strictly smaller than $k-1$, we already know that $\tau_j = \sigma_j$ and $\tau = \sigma$. It remains to prove that $\tau = \sigma$.

Consider a context C and let, for any $j \in J$, α_j be the behavior of A_j in C relative to t_k. Note that because $\tau_j = \sigma_j$ we know that α_j is also the behavior of A_j in C relative to s_k. Hence α_j is independent of whether t_k or s_k is attached to C.

To show that $\tau = \sigma$ we will construct an OTWA B of size bounded by $f(m)$ that simulates in s_1 the behavior of A in the top part of s_k and simulates in t_1 the behavior of A in the top part of t_k and we conclude that they are the same using Proposition 26. The idea is that because the A_j have a rank strictly smaller than $k-1$ and a size smaller than m, they cannot distinguish s_{k-1} from t_{k-1} and hence their behavior in the top part of s_k or t_k can be simulated by a structure oracle.

For $j \in J$, let θ_j be the relative behavior type of A_j in s_{k-1} (and hence also in t_{k-1} by induction hypothesis).

Based on $(\alpha_j)_{j\in J}$ and $(\theta_j)_{j\in J}$ we construct for any $j \in J$ and any pair of state (p,q) of A_j, a MSO formula $\varphi^j_{p,q}(x,y)$ such that the formula is true for a pair of nodes x, y of s_1 (resp. t_1) iff A_j has a run in Cs_k (resp. Ct_k) starting in state p at the copy x' of x inside the top part of s_k (resp. t_k) and ending in state q at y' the copy of y inside the top part of s_k (resp. t_k). Similarly we construct a MSO formula $\phi^j_{p,q}(x,y,z)$ that moreover restrict the runs of A_j to the subtree rooted at z. The formulas are *blind* in the sense that they don't use the unary predicates labeling each node. They are constructed by induction on the rank of A_j. We only give the proof for $\varphi^j_{p,q}(x,y)$ as $\phi^j_{p,q}(x,y,z)$ uses the same construction but moreover restricts all quantifications to the subtree rooted at z.

Assume first that A_j is a TWA. The runs of A_j do not depend on any other automata and θ_j is a set of pairs of states of A_j. The formula $\varphi^j_{p,q}(x,y)$ builds on the classical construction for simulating the run of a TWA in a tree using MSO formulas (see also Section 6). We apply this construction to the TWA A' that uses the transition table of A_j on inner nodes and switch state according to α_j when visiting the root of the tree and according to θ_j when visiting a leaf of the tree. As A_j cannot distinguish s_{k-1} from t_{k-1}, the labels of the leaf are irrelevant and the formulas built from A' have the desired properties.

Assume now that A_j is of rank $l_j > 0$. Let $(A_i)_{i\in I}$ be the nested TWA occurring in the inductive definition of A_j. Again the formula we construct uses the classical simulation using MSO of the following TWA A'. At an inner node, the transitions of A' are given by the transition table of A_j together with the formulas $(\varphi^i)_{i\in I}$ and $(\phi^i)_{i\in I}$ constructed by induction in order to test whether subruns are accepting of not. At the root of the tree the transition of A' is made according to α_j. At a leaf x the transition of A' is made according to $\theta_j((\alpha'_i)_{i\in I})$ where α'_i is the behavior of A_i in the context CC' where C' is the context formed from Ct by placing the hole in x in the current tree, i.e ignoring the subtree rooted at x. We are done if we can express α'_i in MSO but that can be achieved by composing the relations $(\alpha_j)_{j\in J}$ with the formulas $(\varphi^i)_{i\in I}$ and $(\phi^i)_{i\in I}$ computed by induction, and this composition can be expressed in MSO.

From the formulas just computed it is now possible to compute an oracle $\mathcal{O}$ such that for any node v of s_1 or t_1, the structural information about v implies all possible runs $(A_j)_{j\in J}$ may have starting from v and therefore which transitions of A at the copy of v in s_k or t_k can be taken. The size of $\mathcal{O}$ can be deduced from the formulas computed above and is a function g of m.

Based on $\mathcal{O}$, we construct the desired OTWA B as follows. The states of B are exactly the states of A and B relies on the structural automaton $\mathcal{O}$. Let τ' be the basic relative behavior type of A in s_{k-1} and σ' be the basic relative behavior type of A in t_{k-1}. Notice that σ' and τ' may be different as we only know by induction that $(k-2)$-nested TWA cannot distinguish s_{k-1} from t_{k-1}. At an inner node of label **b**, the transitions of B are set according to the transitions of A, using the structural $\mathcal{O}$-information for deciding whether A_i has an accepting run or not in the appropriate subtree. At a leaf of label **a**, the transitions of B is set according to $\sigma'((\alpha'_j)_{j\in J})$ where α'_j is the behavior of A_j in the context formed by placing a hole at the current leaf and, as above, can be computed from the information provided by $\mathcal{O}$ and $(\alpha_j)_{j\in J}$. Similarly, at a leaf of **b**, the transition of B is set according to $\tau'((\alpha'_j)_{j\in J})$. By construction it is now clear that B is in state q at a node x of s_1 iff A was also in state q at x in s_k. Similarly for t_1 and t_k. Hence the behavior of B in s_1 is the same as the behavior of A in s_k while the behavior of B in t_1 is the same as the behavior of A in t_k. From the discussion above, the size of B is $f(m) = O(mg(m))$ and by Proposition 26 the two behaviors must be equal. □

5.3 Deriving the strict containment

We now apply Proposition 28 to derive strict containment results. In [Bojańczyk et al. 2006] it is shown that L_1 is recognizable by a TWA using 1 pebble. A simple modification of their argument actually show that L_1 is recognizable by a 1-nested TWA. Moreover, a simple induction on k then shows that for any k, L_k is recognized by a k-nested TWA. We refrain from giving the argument here, only for the reason that it would require a detailed definition of the definition of the language L_1.

It follows by Proposition 28 that for nested TWA, more nesting depth provides more expressive power:

Theorem 29 *L_k is recognizable by a k-nested TWA but not by a $(k-1)$-nested TWA.*

It is easy to see that the union over k of all the L_k is not a regular language. The language $\mathcal{L}_{\text{sep}}$ separating MSO from nested TWA is therefore formed from the union of all L_k by relaxing the constraint of being k-separated for some k. It will be defined such that the intersection of $\mathcal{L}_{\text{sep}}$ with the set of k-separated trees is exactly L_k. In particular, all the trees s_k from Proposition 28 belong to $\mathcal{L}_{\text{sep}}$, but none of the trees t_k does. Therefore, no nested TWA can recognize $\mathcal{L}_{\text{sep}}$. More formally, $\mathcal{L}_{\text{sep}}$ is the set of trees with labels in $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ such that **a** appears only at the leaves and such that the following rewriting process ends with a tree in L_1. The rewriting process replace a subtree whose root has label **c** and which does not contain any other **c**-node, with a leaf of label **a** if this subtree belongs to L_1, or with a leaf of label **b** otherwise. It ends when there is no more node of label **c** in the tree. The following fact is easy to prove.

Fact 30 ([Bojańczyk et al. 2006]) *$\mathcal{L}_{sep}$ is a regular tree language, i.e., it is definable in* MSO.

From Proposition 28 it follows that $\mathcal{L}_{\text{sep}}$ cannot be recognized by a nested TWA and Theorem 2 is proved.

Theorem 31 *$\mathcal{L}_{sep}$ cannot be recognized by a nested TWA.*

6. COMPLEXITY ISSUES

In this section we study the complexity of two types of problems for Regular XPath(W) and nested TWA. The *(finite) satisfiability*, or *(finite) non-emptiness*, problem is to determine the existence of a (finite) tree that satisfies a given Regular XPath(W) expression (or is accepted by a given nested TWA). The *evaluation* problem is to compute, given a finite tree and a Regular XPath(W) path expression (or nested TWA), the set of pairs satisfying the path expression (or, accepted by by the nested TWA).

We first comment on the size of a nested TWA. Given a nested TWA A, its state-size $|A|$ will be the total number of states of all the nested TWA that are used in the inductive definition of A. The total-size $||A||$ of A will be 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 $|A|$. The translation from Regular XPath(W) to nested TWA described in Lemma 7, since it runs in linear time, always produces a nested TWA whose total-size is linearly bounded by the length of the input expression. Hence, for the results below, it is enough to prove all our upper bounds for nested TWA model while proving the lower bounds for Regular XPath(W) expressions.

We start with the evaluation problem.

Theorem 32 *(i) Given a nested TWA A and a finite tree t, one can compute $A(t)$ in time polynomial in $||A||$ and $|t|$.*

(ii) Given a Regular XPath(W) *path expression α, and a finite tree t, one can compute $[\![\alpha]\!]^t$ in time polynomial in $|\alpha|$ and $|t|$.*

PROOF. Since Regular XPath(W) expressions can be translated into nested TWA in linear time, it suffices to prove the first part of the theorem. Let A be any nested TWA, let $A_0, \ldots, A_{n-1}$ be all subautomata used (recursively) by A, in order of increasing nesting depth, let $A_n = A$ and let t be any finite tree. For each $i \leq n$ starting with $i = 0$, and for each node $x \in t$, we compute and store the relation $A_i(t_x)$. Hence it is enough to consider the case of TWA. Using classical automata techniques, this can be done in polynomial time. □

Note that the naive algorithm used in the above proof is not likely to be optimal. We leave it as an open question whether non-emptyness of $A(t)$ can be tested $O(||A|| \cdot |t|)$.

Theorem 32 shows 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, the *emptiness* problem for nested TWA, and the *satisfiability* problem for Regular XPath(W) path expressions, are 2-ExpTime-complete. Note that satisfiability is only ExpTime-complete for Core XPath and for Regular XPath, while it is non-elementary for $\mathsf{FO}(\mathsf{MTC})$ (and in fact, already for FO) [ten Cate and Lutz 2007].

Theorem 33 *The following problems are* 2-EXPTIME*-complete:*

(i) The satisfiability and finite satisfiability problems of Regular XPath(W).

(ii) The non-emptiness and finite non-emptiness problems for nested TWA, both when the complexity is measured in terms of the state-size and when it is measured in terms of the total-size.

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 [ten Cate and Lutz 2007]). Theorem 33 is to be compared with a similar result over nested words obtained in [Alur et al. 2007]. In [Alur et al. 2007] 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 $\mathsf{FO}(\mathsf{MTC})$.

The remainder of this section is devoted to a proof of Theorem 33. We will consider in this section only the binary tree case but it will follow immediately from Section 7 that this extends to arbitrary unranked ordered trees. We therefore assume for the rest of this section that the tree are binary, possibly infinite.

The upper bounds are based on a double exponential time translation from nested TWA to top-down tree automata. It is well known that non-emptiness of top-down tree automata can be tested in PTime. The lower bounds are based on a simulation of exponential space bounded alternating Turing machines using Regular XPath(W) formulas. Since finiteness of a tree is definable by Regular XPath(W) expressions and by nested TWA, finite satisfiability and finite non-emptiness reduce to satisfiability and non-emptiness, respectively. Therefore, it is enough to prove the upper bounds for the case of arbitrary, possibly infinite trees while proving the lower bounds for the case of finite trees only.

We start with the translation of nested TWA into top-down tree automata on infinite trees. Our goal is to show the following propositions by simultaneous induction on the rank of the nested TWA. These propositions make use of the following notation. Given an alphabet Σ and a unary predicate P we denote by Σ^P the alphabet constructed from Σ and P - hence Σ^P has two copies per letter of Σ, distinguishing the positions where P holds from those where P does not hold. Given a Σ^P-tree t we denote by t_Σ the Σ-tree constructed from t by ignoring the extra predicate P.

Proposition 34 *Let A be a nested TWA using the alphabet Σ. There exists a top-down tree automata B over Σ^P, computable in time doubly exponential in $|A|$, such that B accepts* *a Σ^P-tree t iff all nodes of t marked by P are exactly those where the subtree of t_Σ rooted at that node is accepted by A.*

Proposition 35 *Let A be a nested TWA using the alphabet Σ. There exists a top-down tree automata B over Σ^P, computable in time doubly exponential in $|A|$, such that B accepts a Σ^P-tree t iff all nodes of t marked by P are exactly those where A has an accepting run in t_Σ starting from that node.*

Before we prove these propositions we note that each of them immediately yields a doubly exponential time transformation of nested TWA into top-down tree automata and hence the upper bounds for Theorem 33.

PROOF. Let A be a nested TWA. The constructions of both top-down tree automata are made by simultaneous induction on the rank of A.

The base cases is when A is a TWA. Consider first the top-down automata required by Proposition 35. From A we construct an alternating TWA A' over Σ^P-trees which essentially simulates A on every node of a Σ^P-tree. More precisely, A' uses universality to investigate all nodes of the tree and, for each node, starts a simulation of A from that node if its label contains P or starts a simulation of $\bar{A}$ if its label does not contain P, where $\bar{A}$ is the alternating TWA for the complement of A. The construction of A' is done in time linear in $||A||$ and $|A'|$ is linear in $|A|$. We then transform A' into a top-down tree automata using classical constructions in time exponential in $|A'|$ [Muller and Schupp 1995]. Altogether the construction requires a time exponential in $|A|$. Consider now the top-down tree automaton required by Proposition 34. It is constructed similarly as above with the extra difficulty that in order to simulate A inside the appropriate subtree it is necessary to remember where the simulation started. We avoid this problem by first making A *one way*. This extra step induces an extra exponential blow-up. More precisely, from A we construct a top-down tree automata B which recognize exactly the same tree language as A. Using [Muller and Schupp 1995] B is constructed in time exponential in $|A|$. We then view B as an alternating TWA B' where each transitions of the top-down tree automata is replaced by a universal move to each of the children of the current node. Note that $|B'| = |B|$ and that B' never goes up in the tree. We then proceed as in the previous case starting with B' and get the desired top-down tree automata in time exponential in $|B'|$. Altogether the construction requires a time doubly exponential in $|A|$.

Assume now that A has rank $k > 0$. Let $A_1, \cdots, A_l$ be the nested TWA of rank $< k$ occurring in the definition of A. By induction we can construct for each $1 \leq i \leq l$ a top-down tree automata B_i over Σ^{P_i}-trees such that P_i holds on each node where A_i has an accepting run on the corresponding subtree. By induction we can also construct for each $1 \leq i \leq l$ a top-down tree automata C_i over Σ^{Q_i}-trees such that Q_i holds on each node where A_i has an accepting run starting from that node in an initial state. Let Γ be the alphabet extending Σ with all the propositions $P_1, \cdots, P_l, Q_1, \cdots, Q_l$. Let A' be the TWA over Γ-trees simulating A by assuming at each node that A_i has an accepting run in the subtree rooted at that node exactly when the label P_i holds at that node and assuming that A_i has an accepting run starting from that node exactly when Q_i holds at that node. Let D be a top-down tree automata obtained by induction from A' on Γ^P-trees. A top-down tree automata required for Proposition 34 from A is constructed as follows. At each node the automata guesses the values of the predicates $P_1, \cdots, P_l, Q_1, \cdots, Q_l$ and P and then simulates simultaneously all the automata $B_1, \cdots, B_l, C_1, \cdots, C_l$ and D. By induction each of the top-down tree automata were constructed in time doubly exponential in $|A_i|$, hence the total construction is performed in time doubly exponential in $|A|$. A top-down tree automata required for Proposition 35 from A is obtained similarly.

This concludes the proof of Proposition 35 and Proposition 34. □

We now turn to the simulation of exponential space bounded alternating Turing machines using Regular XPath(W) formulas. We start with a well known fact about alternating Turing machines.

Fact 36 *There is an alternating Turing machine M that uses at most 2^n tape cells on inputs of size n, and such that whether M accepts an input string w is a 2-ExpTime-hard problem in the length of w. Moreover, we may assume that every configuration of M has exactly two successor configurations.*

Lemma 37 *Let M be as above. For each string w of size n, there is a Regular XPath*(W) *node expression ϕ computable in time polynomial in n, such that ϕ is satisfiable on finite trees iff M accepts w.*

PROOF. A configuration of M is a description of the current content of the tape, the current position of the head, and the current state. As M uses only 2^n cells on w, this can be coded by a string of length 2^n, where each position carries the information of the tape at the corresponding position. Moreover the position where the head is also carries the current state of M.

We code this with a finite binary tree as follows. Each position of the tape is coded in binary by a string of length n that we view as a labeled tree of depth n where the string corresponds to the leftmost branch, the rest of the tree being labeled with a dummy symbol. If i is a number let $T[i]$ be the corresponding coding tree.

A configuration is now coded as a sequence of length 2^n where the label of the i^{th} element carry the content of the i^{th} position of the configuration, its left child is $T[i]$ and its right child the next element. In order to separate configurations we also assume one extra #-node at the end of the sequence. Our encoding for runs of M is depicted in Figure 5.

Sequences of configurations can now code successive configurations of M. The branching structure of the tree can be used to code alternation of M. We require that the topmost part of the tree codes the initial configuration, that is essentially w, that configurations coding an existential state have only one successive configuration in the tree, while configurations coding a universal state must have two successive configurations in the tree. All the configurations occurring at the bottom of the tree must be accepting.

A bit of terminology. A node x of the tree is said to be *important* if its left subtree is $T[i]$ for some i. Hence an important node is expected to code the content of a cell in a configuration. The *position* of an important node x is then i.

Right now all the description given above can be enforced by a Regular XPath(W) node expression (no W is needed). Indeed the formula checks that (i) the children of a #-symbol are important and their positions are 0, (ii) two successive important nodes have successive positions (this requires $O(n)$ formulas of size $O(n)$ checking the incrementation bit per bit), (iii) the parent of a #-node is important and its position is $2^n - 1$, (iv) the #-nodes following an existential state have only one child, (v) the #-nodes following a

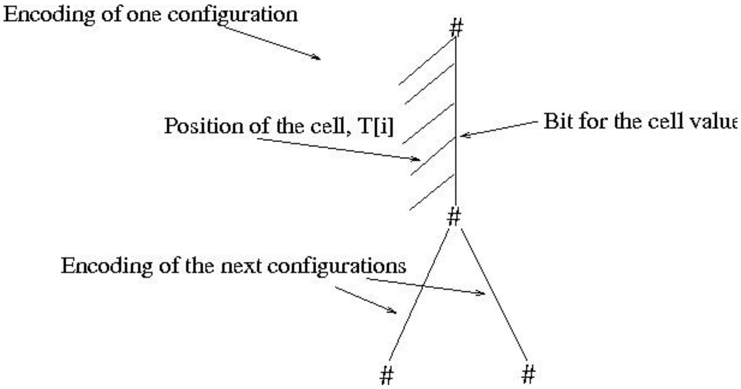

Encoding of one configuration
#
Position of the cell, T[i]
Bit for the cell valu
#
Encoding of the next configurations
#
#

Fig. 5. Encoding of runs of M.

universal state have exactly two children, (vi) the initial configuration codes w and (vii) all configurations with no successor are accepting.

The difficulty is to enforce that two configurations successive in the tree are consistent with the transitions of M. To do this we first describe a Regular XPath(W) node expression $\phi_{eq}(\chi)$ —taking as argument any Regular XPath(W) node expression χ, such that $\phi_{eq}(\chi)$ is true at an important node x with position i if and only if $\mathsf{W}\chi$ *holds at the important node with position i in some successor configuration*. This can be done by first restricting the scope to the subtree below x, then walking down passing exactly one $\#$ on the way, and then successively checking that each bit of the position is the same as in the original node. The corresponding Regular XPath(W) node expression is:

$$\phi_{eq}(\chi) = \mathsf{W}(\Big\langle(\downarrow[\neg\#])^*/\downarrow[\#]/(\downarrow[\neg\#])^*/\downarrow[\neg\# \wedge \bigwedge_i \varphi_i \wedge \mathsf{W}\chi]\Big\rangle)$$

where φ_i is $\langle(\downarrow_1)^i[0]\rangle \leftrightarrow \langle\uparrow^*[\mathsf{root}]/(\downarrow_1)^i[0]\rangle$. It follows that $\neg\phi_{eq}(\neg\chi)$ is true at an important node with position i if and only if $\mathsf{W}\chi$ holds at the important node with position i in *every* successor configuration. Once this is done, it is straightforward (but tedious) to construct the Regular XPath(W) node expression enforcing a correct transition of M between each two successive configurations of the tree. □

This concludes the proof of Theorem 33.

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.

Intuitively, for any tree t, $\mathsf{bin}(t)$ is the binary tree constructed from t by using the first child relation and the next subling relation as the two new successor relations. More formally $\mathsf{bin}(t)$ is the tree t' such that there exists a bijective relation f from the nodes of t to the nodes of t' verifying the following properties: (i) $f(\epsilon) = \epsilon$, (ii) for all $x \in t$ such that $x1 \in t$, $f(x1) = f(x)1$ and (iii) for all $x \in t$ such that $\exists y\ x = yi$ and $i > 1$, $f(x) = f(y(i-1))2$. Moreover, 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 38 *For any set of trees L,* $\mathsf{bin}(L)$ *is definable in Regular XPath*(W) *iff L is.*

In order to prove Theorem 38, 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. I). 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 39 *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 sub-programs 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 40 *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.*

PROOF. (sketch) The proof goes by induction. The only interesting case is when ϕ is of the form $\langle\alpha\rangle$ for some path expression α. By induction hypothesis, we may assume that all node expressions inside α are of the given form. Assume that $\chi_1, \ldots, \chi_n$ are all the corresponding forest tests and $\xi_1, \ldots, \xi_m$ are all the corresponding tree-local formulas.

Hence, as in the proof of Lemma 7, one can see α as a TWA A such that each transition assumes the validity of some of the χ_i and some of the ξ_j.

The run of A can be decomposed into: (i) the initial subrun that stays into the tree where the computation started, (ii) navigation between the roots of the forests and (iii) the final part of the run, which starts at the root of some tree and stay within that tree.

When staying within a tree, the subset of $\chi_1, \ldots, \chi_n$ that contains the valid forest tests does not depend on the node of the tree but only on its root. Therefore this subset can be guessed once for all and, for each guess, the whole subcomputation of A, including the tests for ξ_j, can be simulated with an appropriate tree-local node expression using arguments as in Lemma 7.

This implies that subruns of A of the form (ii) and (iii) can be combined into forest tests as the top navigation and the appropriate guesses can be performed in MSO.

Similarly for each subset of $\chi_1, \ldots, \chi_n$, the subrun (i) is uniquely determined by the tree where the computation started and can be simulated by a tree-local node expression. The result now follows by taking a big disjunction over all possible subsets and, for each subset, combining the appropriate tree-local node expression with the appropriate forest test. □

We are now ready to prove Theorem 38.

PROOF OF THEOREM 38. 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 39, 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 40. To this end, consider $\mathsf{W}'\chi$, for any Regular XPath(W) node expression χ. By proposition 40, 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 $\to$ 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 41 *Note that the proof of the right-to-left direction of Theorem 38 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

Our results show that the tree languages definable in $\mathsf{FO}(\mathsf{MTC})$ form a robust class of tree languages that can be characterized in several ways. It lies strictly between the class of FO definable tree languages and the class of regular tree languages. We conclude by listing some additional consequences of our results, and some open problems.

8.1 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 (although it would increase the complexity of satisfiability). Note that this does not follow immediately from the definition of Regular XPath(W). In [Benedikt et al. 2005], 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 [Benedikt and Fundulaki 2005] introduced and motivated the notion of *subtree composition closure*. For any path expression α and XML tree t, let $[\![\alpha]\!]^t_\varepsilon = \{x \mid (\varepsilon, x) \in [\![\alpha]\!]^t\}$, 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, $[\![\gamma]\!]^t_\varepsilon = \bigcup\{[\![\beta]\!]^{\mathsf{subtree}(t,x)}_\varepsilon \mid x \in [\![\alpha]\!]^t_\varepsilon\}$.* In [Benedikt and Fundulaki 2005], a number of XPath fragments were shown to be closed under subtree composition. Theorem 3 easily implies that Regular XPath(W) is closed under subtree composition.

Corollary 42 *Regular XPath*(W) *is closed under path intersection, path complementation, and subtree composition.*

We do not know whether W adds any expressive power to Regular XPath. However we can show that this issue reduces to the question whether Regular XPath is closed under subtree composition:

Theorem 43 *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 42 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 $.[\langle\alpha^{-1}[\mathsf{root}]\rangle]$. $\Box$

It is worth noting that a variant of W for path expressions was proposed in [Bird et al. 2005], under the name *subtree scoping*. It has the following semantics: $[\![\mathsf{W}'\alpha]\!]^t = \{(x,y) \mid x \leq y$ and $(x,y) \in [\![\alpha]\!]^{t_x}\}$. It is easy to see, using this variant of the operator, $\mathsf{W}\phi$ can be expressed as $\langle\mathsf{W}'(.[\phi])\rangle$, and that the subtree-composition of path expressions α and β can be expressed as $\alpha/\mathsf{W}'\beta$. Theorem 3 implies that Regular XPath(W) is closed even under the W'-operator.

8.2 Normal forms for FO(MTC)

Theorem 3 implies a normal form for $\mathsf{FO(MTC)}$ over trees. Let $\mathsf{FO}^4\mathsf{(MTC)}$ be the set of $\mathsf{FO(MTC)}$ 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 $\mathsf{FO(MTC)}$ formula φ is *single-parameter* if for all subformulas $[\mathsf{TC}_{xy}\psi](u,v)$ of φ, ψ has only one parameter. The formula is *parameter-free* if it does not use any parameter. Finally a $\mathsf{FO(MTC)}$-formula ϕ is said to be "looping" if all subformulas with main connective TC are of the form $[\mathsf{TC}_{xy}\ \phi](u,u)$.

The following normal form combines Theorem 3 and the translation of TWA into transitive closure logic from [Neven and Schwentick 2003].

Theorem 44 *Every $\mathsf{FO(MTC)}$ formula with at most one free variables is equivalent to a single-parameter looping formula of $\mathsf{FO}^4\mathsf{(MTC)}$.*

PROOF. The proof is an induction on the rank of the nested TWA equivalent to the $\mathsf{FO(MTC)}$ formula obtained using Theorem 3 and Lemma 7. We show that for all nested TWA A, there exists a single-parameter looping formula $\varphi(x,y)$ of $\mathsf{FO}^4\mathsf{(MTC)}$ such that for all tree t and nodes x, y of t with $y < x$,

$$(x,x) \in A(\mathsf{subtree}(t,y)) \Leftrightarrow t \models \varphi(x,y)$$

The basis and the induction steps make use of the following result which generalize slightly the statement of the main result of [Neven and Schwentick 2003] but follows directly from its proof. The main differences with the result as stated in [Neven and Schwentick 2003] are: (i) we consider runs that start at a given node x of the tree and not just the root of the tree (ii) the proof of [Neven and Schwentick 2003] uses predicates counting the depth of a node modulo some number. These predicates are definable by a parameter-free looping $\mathsf{FO(MTC)}$-formula.

Proposition 45 ([Neven and Schwentick 2003]) *For all TWA A, there exists a parameter-free looping formula $\varphi(x)$ of $\mathsf{FO}^3\mathsf{(MTC)}$ such that for all tree t and node x of t,*

$$(x,x) \in A(t) \Leftrightarrow t \models \varphi(x)$$

The base case of our induction, i.e. A is a TWA, follows immediately by relativizing all quantifications in the formula obtained in Proposition 45 to the descendants of y. When doing this the number of variables increases to four and y becomes a parameter of the TC formulas.

Now consider a nested TWA A of rank $k > 0$. Let $A_1, \cdots, A_n$ be the nested TWA of rank $(k-1)$ occurring in the definition of A. By induction there exists a looping single-parameter formula $\phi_i(u,v)$ equivalent to A_i for each $i \leq n$. By Proposition 45 there exists a looping parameter-free formula $\xi(x)$ that is equivalent to A, assuming the presence of predicates $P_i(u)$ and $Q_i(u)$ that hold on a node u of t iff A_i has an accepting run starting from node u in the subtree rooted in u in the case of $P_i(u)$ and in the whole tree in the case of $Q_i(u)$. As before we relativize all quantifications to the descendants of y and this gives a looping single-parameter formula $\varphi(x,y)$ of $\mathsf{FO}^4(\mathsf{MTC})$ that simulates A. When doing this the predicates $P_i(u)$ remain untouched but $Q_i(u)$ now becomes $Q_i(u,y)$ with the obvious meaning. Replacing the predicates $P_i(u)$ and $Q_i(u,y)$ using the formula $\phi_i(u,v)$ yields the desired normal form. □

When the formula of $\mathsf{FO}(\mathsf{MTC})$ has more than two free variables, looping can no longer be enforced. However the direct translation from Regular XPath(W) into $\mathsf{FO}(\mathsf{MTC})$ shows that $\mathsf{FO}(\mathsf{MTC})$ still has the 4-variable property.

Theorem 46 *Every* $\mathsf{FO}(\mathsf{MTC})$ *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 $\mathsf{FO}(\mathsf{MTC})$ 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 $\mathsf{FO}(\mathsf{MTC})$, denoted by FO^* in [ten Cate 2006], has exactly the same expressive power as Regular XPath$^{\approx}$ (see also our discussion of related work in Section 1) and that FO^* has the three variable property [ten Cate 2006].

8.3 Positive nested TWA and pebble automata

Our methods can also be used to obtain a new characterization of $\mathsf{FO}(\mathsf{pos\text{-}MTC})$, the fragment of $\mathsf{FO}(\mathsf{MTC})$ in which all occurrences of the TC-operator are in the scope of an even number of negation signs. It was shown in [Engelfriet and Hoogeboom 2007] that tree walking automata with nested pebbles, *pebble TWA* for short, have the same expressive power as $\mathsf{FO}(\mathsf{pos\text{-}MTC})$ in terms of definable tree languages. It turns out that by restricting the use of negation in Regular XPath(W) expressions or in nested TWA, an analogue of Theorem 3 for $\mathsf{FO}(\mathsf{pos\text{-}MTC})$ can be obtained.

We call a node or path expression *positive* if all occurrences of the transitive closure operator $*$ are in the scope of an even number of negation signs. We call a nested TWA *positive* if it does not make any negative tests, by which we mean that its transitions do not rely on the non-existence of runs of some of its nested TWA, except for testing whether a node is the root, is a leaf, is a first child, or is a last child.

Theorem 47 *The following three formalisms have the same expressive power:*

(*1*) $\mathsf{FO}(\mathsf{pos\text{-}MTC})$,

(*2*) *positive Regular XPath*(W) *formulas,* *(3) positive nested TWA*

The result applies again to Boolean queries, unary queries, and binary queries. The proof of equivalence between (2) and (3) is along the same lines as in Lemma 7, using the fact that, in positive Regular XPath(W) formulas, negations can be pushed down to the atomic subformulas. Concerning the equivalence of (1) and (2), a simple induction shows that every positive Regular XPath(W) expressions is equivalent to an FO(pos-MTC) formula. The proof of the converse, that each FO(pos-MTC) formula is equivalent to a positive Regular XPath(W) formula, is done via tree patterns as in Section 4 by realizing that each step preserves the number of negations. The details can found in Appendix B.

Perhaps Theorem 47 can help in resolving the open question whether FO(MTC) is strictly more expressive than FO(pos-MTC) on trees.

8.4 Open problems

It is open whether Regular XPath is strictly included in Regular XPath(W), i.e., whether W adds expressive power. This is equivalent to the problem whether Regular XPath is closed under subtree composition (cf. Section 8.1).

It is open whether FO(pos-MTC) is strictly included in FO(MTC) on trees (cf. Section 8.3). This is equivalent to the problem whether pebble automata are closed under complementation.

It is open whether the hierarchy of nesting of TCs in FO(MTC) is strict.

It is open whether FO* is strictly included in FO(MTC) on trees, i.e., whether the fact that parameters are allowed in FO(MTC) is essential.

It is open whether there are finitely many generalized quantifiers $Q_1, \ldots, Q_n$ (as in [Lindström 32]) such that FO extended with the quantifiers $Q_1, \ldots, Q_n$ has the same expressive power as FO(MTC) on trees. Note that the monadic transitive closure operator is an example of a generalized quantifier, but that second order quantifiers are not. We conjecture that the answer is positive.

In order to present the last open problem, we need to introduce some terminology. By a k-ary *logical operation on binary relations* we mean a map that takes as input a set D and binary relations $R_1, \ldots, R_k$ on D, and outputs a new binary relation on D, in a way that respects isomorphisms. Thus, for instance, union, complementation and composition are logical operations on binary relations. It follows from the results in [ten Cate 2006] that there is a finite set of operations on binary relations (viz. union, complement, composition, and transitive closure) such that every binary FO*-query on trees can be defined by an algebraic expression built up from the atomic binary relations $\uparrow$, $\downarrow$, $\leftarrow$, $\rightarrow$, ".", and label tests p? (interpreted as subrelations of the identity relation) using the given operations. It is open whether a similar result holds for FO(MTC) and MSO. (It is important to note here that W is not a logical operation, since the denotation of Wϕ in a model is not determined by the denotation of ϕ in the same model.)

REFERENCES

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

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

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

BENEDIKT, M. AND FUNDULAKI, I. 2005. XML subtree queries: Specification and composition. In *DBPL*. Number 3774 in LNCS. 138–153.

BIRD, S., CHEN, Y., DAVIDSON, S. B., LEE, H., AND ZHENG, Y. 2005. Extending XPath to support linguistic queries. In *PLAN-X*. 35–46.

BOJAŃCZYK, M. AND COLCOMBET, T. 2008. Tree-walking automata do not recognize all regular languages. *SIAM J. Comput. 38,* 2, 658–701.

BOJAŃCZYK, M., SAMUELIDES, M., SCHWENTICK, T., AND SEGOUFIN, L. 2006. Expressive power of pebble automata. In *ICALP (1)*. 157–168.

ENGELFRIET, J. AND HOOGEBOOM, H. J. 2007. Nested pebbles and transitive closure. *Logical Methods in Computer Science 3,* 2.

ENGELFRIET, J., HOOGEBOOM, H. J., AND SAMWEL, B. 2007. XML transformation by tree-walking transducers with invisible pebbles. In *PODS*, L. Libkin, Ed. ACM Press, 63–72.

FAN, W., GEERTS, F., JIA, X., AND KEMENTSIETSIDIS, A. 2007. Rewriting regular xpath queries on XML views. In *ICDE*. 666–675.

GHEERBRANT, A. AND TEN CATE, B. 2009. Complete axiomatizations of MSO, FO(TC1) and FO(LFP1) on finite trees. In *LFCS*, S. N. Artëmov and A. Nerode, Eds. Lecture Notes in Computer Science, vol. 5407. Springer, 180–196.

GOTTLOB, G., KOCH, C., AND PICHLER, R. 2002. Efficient algorithms for processing XPath queries. In *VLDB*. 95–106.

GRÄDEL, E. 1991. On transitive closure logic. In *CSL*, E. Börger, G. Jäger, H. K. Büning, and M. M. Richter, Eds. LNCS, vol. 626. Springer, 149–163.

LAROUSSINIE, F., MARKEY, N., AND SCHNOEBELEN, P. 2002. Temporal logic with forgettable past. In *LICS*. 383–392.

LINDSTRÖM, P. 32. First-order predicate logic with generalized quantifiers. *Theoria 32*, 186–195.

MARX, M. 2004. XPath with conditional axis relations. In *Proceedings of EDBT 2004*. Lecture Notes in Computer Science, vol. 2992. Springer.

MARX, M. 2005. Conditional XPath. *Transactions on Database Systems 30,* 4, 929–959.

MARX, M. AND DE RIJKE, M. 2005. Semantic characterizations of navigational XPath. *SIGMOD Record 34,* 2, 41–46.

MULLER, D. E. AND SCHUPP, P. E. 1995. Simulating alternating tree automata by nondeterministic automata: new results and new proofs of the theorems of rabin, mcnaughton and safra. *Theor. Comput. Sci. 141,* 1-2, 69–107.

NENTWICH, C., CAPRA, L., EMMERICH, W., AND FINKELSTEIN, A. 2002. xlinkit: a consistency checking and smart link generation service. *ACM Transactions on Internet Technology 2,* 2, 151–185.

NEVEN, F. AND SCHWENTICK, T. 2003. On the power of tree-walking automata. *Inf. Comput. 183,* 1, 86–103.

POTTHOFF, A. 1994. Logische klassifizierung regularer baumsprachen. Ph.D. thesis, Christian-Albrechts Universität Kiel.

RABIN, M. O. 1970. Weakly definable relations and special automata. In *Mathematical Logic and Foundations of Set Theory*. North-Holland, 1–23.

SCHWENTICK, T. 2000. On diving in trees. In *MFCS*, M. Nielsen and B. Rovan, Eds. Lecture Notes in Computer Science, vol. 1893. Springer, 660–669.

TEN CATE, B. 2006. The expressivity of XPath with transitive closure. In *PODS*. 328–337.

TEN CATE, B. AND LUTZ, C. 2007. The complexity of query containment in expressive fragments of xpath 2.0. In *PODS*. ACM, 73–82.

TEN CATE, B. AND MARX, M. 2007. Axiomatizing the logical core of XPath 2.0. In *ICDT*.

A. A FIRST NORMAL FORM FOR FO(MTC) ON BINARY TREES

The aim of this section is to prove Lemma 4. Throughout the section, we restrict attention to binary trees.

The following Ehrenfeucht-Fraïssé (EF) game for FO(MTC) was introduced by Grädel [Grädel 1991]:

Definition 48 (EF games for $\mathsf{FO(MTC)}$) *The EF game for $\mathsf{FO(MTC)}$ are defined as the standard one for FO, except that, besides the usual moves, there is one more type of move, which consists of the following four steps (counted all together as one round):*

(*1*) *Spoiler selects two pebbles already on the board, say u and v, and chooses one of the two models. He then plays a finite sequence $d_1, \ldots, d_k$ of elements of the chosen model, such that d_1 is the node labeled u and d_k is the node labeled v.*

(*2*) *Duplicator responds by playing a sequence (not necessarily of the same length) $e_1, \ldots, e_\ell$ in the other model, where, again, e_1 is the node labeled u and e_ℓ is the node labeled v.*

(*3*) *Spoiler selects a pair (e_i, e_{i+1}) (with $i < \ell$) and places new pebbles x and y on the these nodes.*

(*4*) *Duplicator responds by selecting a pair (d_i, d_{i+1}) (with $i \leq k$) and placing the pebbles x and y on the respective nodes.*

The game continues with the new pebble configuration.

We use the following notation for configurations in the game: (M, N, k, Z) denotes the situation in the game between M and N where there are k moves left to go, and the finite partial bijection constructed so far is $Z \subseteq M \times N$. Let the *depth* of an $\mathsf{FO(MTC)}$ formula be the maximal combined nesting depth of quantifiers and/or TC-operators in the formula. We use $\mathsf{FO(MTC)}_k$ to denote the collection of $\mathsf{FO(MTC)}$ formulas of depth at most k. Given models M, N for the same finite relational vocabulary, and $d_1, \ldots, d_n \in M$, $e_1, \ldots, e_n \in N$, we write $(M, d_1, \ldots, d_n) \equiv_{\mathsf{FO(MTC)}_k} (N, e_1, \ldots, e_n)$ if both satisfy the same $\mathsf{FO(MTC)}_k$-formulas in n free variables.

Theorem 49 ([Grädel 1991]) *For all models M, N in the same finite relational vocabulary, $d_1, \ldots, d_n \in M$ and $e_1, \ldots, e_n \in N$, the following are equivalent:*

(*1*) $(M, d_1, \ldots, d_n) \equiv_{\mathsf{FO(MTC)}_k} (N, e_1, \ldots, e_n)$

(*2*) *Duplicator has a winning strategy in the $\mathsf{FO(MTC)}$ game $(M, N, k, \{(d_i, e_i) \mid i \leq n\})$.*

Similarly, we can define a game for $\mathsf{FO(MTC^<)}$.

Definition 50 (EF games for $\mathsf{FO(MTC^<)}$) *The EF game for $\mathsf{FO(MTC^<)}$ is defined as the one for $\mathsf{FO(MTC)}$, except that the following modifications are made to the TC move:*

—*Spoiler must choose u, v and the sequence $x_1, \ldots, x_n$ such that $u < v$ and $u \leq x_k$ and $v \not< x_k$, for all $k \leq n$.*

—*After a TC move is finished, all pebbles except u, v, x, y are removed from the board.*

Theorem 51 *For all models M, N in the same finite relational vocabulary, $d_1, \ldots, d_n \in M$ and $e_1, \ldots, e_n \in N$, the following are equivalent:*

(*1*) $(M, d_1, \ldots, d_n) \equiv_{\mathsf{FO(MTC^<)}_k} (N, e_1, \ldots, e_n)$

(*2*) *Duplicator has a winning strategy in the $\mathsf{FO(MTC^<)}$ game $(M, N, k, \{(d_i, e_i) \mid i \leq n\})$.*

PROOF. A straightforward modification of the proof of Theorem 49. □

Theorem 52 *For binary trees M, N, if Duplicator has a winning strategy for the* $\mathsf{FO}(\mathsf{MTC}^<)$ *game from $(M, N, 3k + |Z| + 2, Z)$, then he has a winning strategy for the* $\mathsf{FO}(\mathsf{MTC})$ *game from (M, N, k, Z).*

PROOF SKETCH. Call a configuration $Z \subseteq M \times N$ *full* if the roots of the two structures are pebbled, and whenever two nodes in one of the trees are pebbled, then their least common ancestor is also pebbled (by *pebbled*, we mean that the node belongs to the domain respectively codomain of Z). A full configuration naturally divides the trees into regions, bounded by pebbles: if $x <_i y$ are pebbled nodes, and for every pebbled node z with $x <_i z$, it holds that $y \leq z$, then we denote by $[x, y]$ the set consisting of x and all $<_i$-descendants of x that are not descendants of y, i.e., "the region bounded by x and y". If a pebbled node x has no pebbled descendants, then we denote by $[x, \ldots]$ the set of x and its descendants, i.e., "the region bounded by "y". Thus, full configurations are convenient, because they allow us to reason by region.

It is not hard to see that every consistent configuration Z can be extended to a full one by adding at most $|Z| + 1$ extra pairs. We call a strategy of Spoiler *tidy* if she only plays TC-moves when the configuration is full. It is not hard to see that if Spoiler has a winning strategy in the $\mathsf{FO}(\mathsf{MTC})$-game (M, N, k, Z), then she has a *tidy* winning strategy in the $\mathsf{FO}(\mathsf{MTC})$-game $(M, N, 2k + |Z| + 2, Z)$, as the extra moves allow her to make the configuration full initially and to restore fullness after every round. Hence, in order to show that Duplicator has a winning strategy in the $\mathsf{FO}(\mathsf{MTC})$-game (M, N, k, Z), it is enough to show the following

Claim: Duplicator has a strategy in the $\mathsf{FO}(\mathsf{MTC})$-game $(M, N, 2k + |Z| + 2, Z)$ that wins from any *tidy* strategy of Spoiler.

The proof proceeds by induction on the number of rounds left. The case when there are zero rounds left is trivial. If there is more than one round left, and Spoiler's first move is a quantifier move, we simply respond using the given $\mathsf{FO}(\mathsf{MTC})$ strategy and use the induction hypothesis. If Spoiler's first move is a TC-move $u = x_1, \ldots, x_n = v$ (and the current configuration is full) we proceed as follows:

We may assume without loss of generality that $u \neq v$, and no x_i other than the first and the last one is a pebbled node. We split up the sequence $x_1, \ldots, x_n$ into maximal segments each belonging to a single region in the tree. We then prefix and postfix each segment by the boundary points of the corresponding region, and we apply Duplicator's given strategy in the $\mathsf{FO}(\mathsf{MTC}^<)$ game to each extended segment. (In the case some segment belongs to a region that has only one pebbled boundary point, we first play an extra quantifier move, selecting any node that is not an ancestor of any node in the segment, thus ensuring that the region containing the segment is bounded by two pebbled nodes). By IH, for each such subgame Duplicator has a winning strategy. Combining these we get an overall winning strategy. □

Lemma 4 now follows by a standard argument.

B. POSITIVE

The goal of this Appendix is to prove Theorem 47. It only remain to show each $\mathsf{FO}(\mathsf{pos}\text{-}\mathsf{MTC})$ formula is equivalent that positive Regular XPath(W) expression.

Recall that a node or path expression of Regular XPath(W) is *positive* if all occurrences of the transitive closure operator $*$ are in the scope of an even number of negation signs.

Similarly, a node or path expression is said to be negative if all occurrences of the transitive closure operator are in the scope of an odd number of negation signs, except for the occurrences that are not inside any node expression. Thus, for instance, a path expression α of the form $(\downarrow/\downarrow)^*[\sigma]$ is considered to be both positive and negative, whereas the node expressions $.[\langle\alpha\rangle]$ and $.[\neg\langle\alpha\rangle]$ are considered positive and negative, respectively. The following gives a more intuitive characterization of the positive path expressions.

Proposition 53 *A node or path expression is positive if and only if it is equivalent to one in which negation is only used in the form $\neg\sigma$ (with $\sigma \in \Sigma$) and $\neg\langle\alpha\rangle$ (with $\alpha \in \{\uparrow,\downarrow,\leftarrow,\rightarrow\}$).*

PROOF. We show the result for node expressions, the case for path expressions then follows immediately. Let ϕ be any positive node expression. To reduce the number of cases, we first replace in ϕ every sub-expression of the form $\alpha[\psi]$ by $\alpha/.[\psi]$. The following rewrite rules then show how to push the negation signs down to the atoms:

$$\begin{array}{lcl}
\neg(\phi \wedge \psi) & = & \neg\phi \vee \neg\psi \\
\neg(\phi \vee \psi) & = & \neg\phi \wedge \neg\psi \\
\neg\neg\phi & = & \phi \\
\neg\langle\alpha \cup \beta\rangle & = & \neg\langle\alpha\rangle \wedge \neg\langle\beta\rangle \\
\neg\langle\alpha/\beta\rangle & = & \begin{cases}
\neg\langle\alpha\rangle \vee \langle\alpha[\neg\langle\beta\rangle]\rangle & \text{for } \alpha \in \{\uparrow,\leftarrow,\rightarrow\} \\
\langle\downarrow[\neg\langle\leftarrow\rangle \wedge \neg\langle\beta\rangle/(\rightarrow[\neg\langle\beta\rangle])^*[\neg\langle\rightarrow\rangle]]\rangle & \text{for } \alpha = \downarrow \\
\neg\langle\alpha_1/\beta\rangle \wedge \neg\langle\alpha_2/\beta\rangle & \text{for } \alpha \text{ of the form } \alpha_1 \cup \alpha_2 \\
\neg\langle\alpha_1/(\alpha_2/\beta)\rangle & \text{for } \alpha \text{ of the form } \alpha_1/\alpha_2 \\
\neg\phi \vee \neg\langle\beta\rangle & \text{for } \alpha \text{ of the form } .[\phi]
\end{cases}
\end{array}$$

They are all self-explaining except maybe for the one for $\neg\langle\downarrow/\beta\rangle$ that requires a bit more explanation. A node expression of the form $\langle\downarrow/\beta\rangle$ does not hold if there is no child of the current node that satisfies $\langle\beta\rangle$, hence we need to investigate all those children one by one and this is exactly what the rewriting formula does. □

The translation of FO(pos-MTC) formulas into positive Regular XPath(W) expressions is done by induction via positive or negative tree patterns as in Section 4. A tree pattern is said to be positive (negative) if all its labeling expressions are positive (negative). The fact that positive tree patterns with only one variable correspond to positive Regular XPath(W) node expressions is immediate. Similarly it is immediate to see that positive tree patterns with two variables correspond to positive Regular XPath(W) path expressions is immediate. The reader can verify that the constructions given in Section 4 showing that tree patterns are closed under FO operations and unary-TC operations preserve the positiveness or the negativeness of tree patterns, except of course for negation that transform one into the other, assuming the construction of Lemma 9 does.

In the remaining part of this section we show that Lemma 9 do preserve polarity.

Lemma 54 *On binary trees,*

(a) *For every two downward positive (negative) path expressions α, β there is a positive (negative) downward path expression γ, such that for all binary trees t, $\llbracket\gamma\rrbracket = \llbracket\alpha\rrbracket \cap \llbracket\beta\rrbracket$.*

(b) For every positive (negative) downward path expression α there is a negative (positive) downward path expression β such that, for all binary trees t, $[\![\beta]\!] = \{(x, y) \mid x \leq y\} \setminus [\![\alpha]\!]$.

We first develop some general theory on what we call *regular expression with tests*. Incidentally, this notion is only used in this section, and will not play any further role in later sections.

Definition 55 (Regular expressions with tests) *Let $\mathcal{P}$ be a finite set of proposition letters. A* regular expression with tests over $\mathcal{P}$ *(or, a* "RET over $\mathcal{P}$"*) is any expression generated by the following recursive definition:*

$$\alpha ::= ?\phi \mid \downarrow \mid \alpha/\beta \mid \alpha \cup \beta \mid \alpha^*$$

where ϕ is a propositional formula over $\mathcal{P}$ (or, inductively, $\phi ::= p \mid \top \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi$, with $p \in \mathcal{P}$).

A RET over a set of proposition letters $\mathcal{P}$ naturally defines a language over the alphabet $2^{\mathcal{P}}$. Expressions of the form $?\phi$ are interpreted as tests, and $\downarrow$ can be seen as an instruction to move to the next node. Thus, for instance, $(?p\ /\downarrow)^*/?\neg q$ accepts precisely those words $s_1 \cdots s_n$ (with $n \geq 1$ and each $s_i \in 2^{\mathcal{P}}$), for which it holds that p is true under the assignments $s_1, \ldots, s_{n-1}$, and q is false under the assignment s_n. One can give a formal, inductive, definition of the language accepted by a RET, but we leave this to the reader.

Fact 56 (RETs define the regular languages) *Every RET over $\mathcal{P}$ defines a regular language over the alphabet $2^{\mathcal{P}}$. Conversely, every regular language over the alphabet $2^{\mathcal{P}}$ is recognized by some RET over $\mathcal{P}$.*

PROOF SKETCH. One direction is shown by translating RETs into formulas of monadic second-order logic. The other direction follows by translating regular expressions over $2^{\mathcal{P}}$ into RETs over $\mathcal{P}$. The latter is done, roughly speaking, by replacing atomic regular expressions of the form S, with $S \in 2^{\mathcal{P}}$, by RET expressions $?(\bigwedge_{p \in S} p \wedge \bigwedge_{p \notin S} \neg p)/\downarrow$, but one has to take care that no $\downarrow$ occurs after the last test in the RET. □

It follows that the family of languages defined by RETs is closed under intersection and complementation. In what follows, we will improve this result a bit further. What makes RETs interesting, compared to ordinary regular expressions, is that the symbols of the alphabet have a bit of internal structure. In particular, since they are composed of proposition letters, a natural notion of monotonicity arises.

Definition 57 (Monotonicity) *Given words w, w over the alphabet $2^{\mathcal{P}}$, and given a proposition letter $p \in \mathcal{P}$, we say that $w \subseteq_p w'$ if w and w' are identical except that at some positions p is false in w and true in w'. A language $\mathcal{L}$ over $2^{\mathcal{P}}$ is said to be* monotone with respect to the proposition letter p *if whenever a word $w \in \mathcal{L}$ and $w \subseteq_p w'$ then also $w' \in \mathcal{L}$. Likewise, $\mathcal{L}$ is said to be* inversely monotone with respect to p *if whenever $w \in \mathcal{L}$ and $w' \subseteq_p w$ then also $w' \in \mathcal{L}$.*

Clearly, if a RET only contains positive, or only negative, currency's of a proposition letter p, then the language it defines is monotone, respectively inversely monotone, in p. The observation can in fact be turned into a syntactic characterization of monotonicity:

Proposition 58 *For every regular language $\mathcal{L}$ over $2^{\mathcal{P}}$, there is a RET α that defines it, such that for all p,*

—*there is an negative occurrence of p in α iff $\mathcal{L}$ is not monotone with respect to p*

—*there is a positive occurrence of p in α iff $\mathcal{L}$ not inversely monotone with respect to p*

PROOF. By Fact 56, every regular language over $2^{\mathcal{P}}$ is defined by *some* RET. Moreover, it follows from our earlier observation that, if a language is not monotone (not inversely monotone) in p, then any RET defining it must contain negative (positive) occurrences of p. Hence, only the left-to-right direction of the two conditions requires our concern.

Suppose α defines a language that is monotone in p, but α contains negative occurrences of p. We may assume without loss of generality that α is in disjunctive normal form (i.e., of the form $\bigvee_i \bigwedge_j (\neg) p_{ij}$). Let α' be obtained from α by replacing each occurrence of $\neg p$ by $\bot$. We claim that β defines the same language as α. One direction of this claim is easy: since $\bot$ logically implies $\neg p$, the language defined by α' is contained in the language defined by α. Conversely, suppose w is accepted by α. Let w' be obtained from w by making p true (and hence $\neg p$ false) at every position. Then w' belongs to the language defined by α iff w belongs to the language defined by α'. By monotonicity and the fact that $w \subseteq_p w'$, w' belongs to the language of α. Hence, w belongs to the language defined by α', and therefore α and α' define the same language.

Likewise, if α defines a language that is inversely monotone in p, then any positive occurrence of p in α may be replaced by $\bot$. Repeating this process for all relevant proposition letters p, we obtain a RET satisfying the required conditions. □

Corollary 59 (Polarity preserving intersection) *For all RETs α and β there is a RET γ defining the intersection of the languages defined by α and β, such that for all proposition letters p, p occurs positively (negatively) in γ iff p occurs positively (negatively) in α or β.*

Corollary 60 (Polarity switching complement) *For each RET α there is a RET β defining the complement of the language defined by α, such that for all proposition letters p, p occurs positively (negatively) in β iff it occurs negatively (positively) in α.*

Finally, we apply the general theory on RETs developed above, in order to give a more elaborate proof of the closure properties of downward path expressions.

PROOF OF LEMMA 54. We will only prove the case for complementation of positive downward path expressions (the other cases are similar). Let α be any positive downward path expression. Let α' be the RET obtained from α by replacing each test of the form $?\phi$ by $?p_\phi$, for some fresh proposition letter p_ϕ. Apply Corollary 60 in order to obtain a RET β, that defines the complement of the language defined by α'. Finally, replace each proposition letter p_ϕ in β by the corresponding original node expression ϕ. It is easy to see that the resulting downward path expression is negative and has the desired property. □