

XPath Leashed

Michael Benedikt

Bell Laboratories

`benedikt@research.bell-labs.com`

Christoph Koch

Universität des Saarlandes

`koch@infosys.uni-sb.de`

This survey gives an overview of formal results on the XML query language XPath. We identify several important fragments of XPath, focusing on subsets of XPath 1.0. We then give results on the expressiveness of XPath and its fragments compared to other formalisms for querying trees, algorithms and complexity bounds for evaluation of XPath queries, and static analysis of XPath queries.

Categories and Subject Descriptors: H.2.3 [**Languages**]: Query languages

1. INTRODUCTION

XPath [Wor99] is a language for matching paths and, more generally, patterns in tree-structured data and XML documents. These patterns may use either just purely the tree structure of an XML document or data values occurring in the document as well.

XPath is used as a component in XML query languages (in particular, XQuery [Wor02] and XSLT [Wor]), specifications (e.g., XML Schema), update languages (e.g., [SHS04]), subscription systems (e.g., [AF00; CFGR00]) and XML access control (e.g., [FCG04]). Because XPath is ubiquitous in programming tools for manipulating XML documents, and XPath processing is a key component of these tools, hundreds if not thousands of papers have appeared over the years dealing with the evaluation and analysis of XPath. Indeed the popularity of XPath as a formalism may be a factor in the explosive growth of XML, as well as an effect.

The XPath standard has its rough edges, but there is an essential navigational core that is an elegant *modal language*. In this core of XPath there is no explicit notion of variable, and modal step expressions allow for navigation relative to a context node and thus can only “see” one element of the document at a time.

An important property of XPath (which follows from its syntactic restrictions that make it a modal language) is that fragments correspond to certain bounded-variable logics. From these logics, XPath inherits nice graph-theoretic properties on the “dependency graphs” of its queries. In particular, the queries have bounded tree-width and bounded hypertree-width. These properties render them amenable to efficient evaluation [GKP05]. XPath is quite unique in the sense that (1) it is a widely used practical language that naturally obeys syntactic restrictions that lead to bounded (hyper)tree-width *and* (2) bounded (hyper)tree-width is of immediate practical relevance to efficient evaluation. (1) is true for modal languages used in verification, but (2) is not, as the query evaluation techniques used in the context of those languages are quite different [BCM⁺90; CGP00].

In this survey, we present the main fundamental results regarding XPath that have been developed since its introduction. These results can be grouped into the categories *expressiveness*, *complexity*, and *static analysis* of XPath.

—We give a detailed account of the known expressiveness results for XPath, but also give a number of new results. In particular, we review the connections between XPath and first-order logic. The main results are that there are first-order queries

not expressible in navigational XPath, but that navigational XPath expresses precisely the two-variable first-order queries over the navigational structure of XML documents. We show that the navigational XPath fragment extended by the aggregation features of XPath does express all first-order queries. We also survey characterizations of fragments of XPath in terms of tree-pattern queries, and characterize XPath in terms of automata.

- We present an in-depth study of XPath complexity and efficient evaluation that revolves around graph-theoretic properties of XPath queries. Large portions of the XPath language can be processed by algorithms that can work in parallel or in streaming fashion. These issues have been studied extensively in the literature, but we present an overview here as well.
- We also survey static analysis problems for XPath, in particular the satisfiability and the containment problem. These have diverse applications such as in the context of XML query optimization, maintaining integrity, and answering queries using views.

The structure of this article is as follows. In Section 2, we present the data model and XPath fragments considered in this article, and give their semantics. Section 3 studies the expressive power of our XPath fragments, relating them to various logics, and the cost (and blow-up) of translating between such languages. Section 4 discusses the main results on the complexity of XPath and of efficient query evaluation, addressing efficient algorithms both in a classical and a stream processing framework, as well as lower bounds. Finally, Section 5 surveys the state of the art of research on static analysis problems for XPath.

For the central results in this survey, proofs are given. In some cases, we give proofs that are simplifications of those in the literature, while in other cases we give new proofs.

2. FRAMEWORK

Any fundamental research study of XPath has to decide what XPath really is – that is, to distinguish which language features of many to focus on. XPath officially refers to the World Wide Web Consortium’s (W3C) standard language. This is a moving target, and indeed while virtually all research on XPath has focussed on the current standard, XPath 1.0 [Wor99], there is already a very substantial extension, XPath 2.0 [Wor02], forthcoming.

Thus the first task for a formal study is to isolate a particular subset of the language with attractive properties, and to distinguish essential language features from provisional design decisions. In this survey we focus exclusively on XPath 1.0, and take the modal and step primitives that characterize XPath 1.0 as the definitive features of the language. Furthermore, since XPath 1.0 is still a large language, we concentrate on a sublanguage that exhibits the basic navigation and data manipulation features. The principal aspects that we ignore are string-manipulation, type conversions, and construction of string values from document fragments. For the most part the operations available at the value level do not affect our basic results, but we will comment briefly on their impact in the appropriate sections. The largest language we consider, denoted *OrdXPath*, allows for the selection of nodes based on navigation within the tree structure, data value comparisons, aggregation, and node position arithmetic. Within *OrdXPath*, we will delineate a hierarchy of sublanguages of XPath 1.0 to which more precise expressiveness or complexity bounds apply. We will refer to these sublanguages as XPath fragments. Of particular interest will be Navigational XPath (*NavXPath*), which deals only with the underlying tree structure of the document. All the fragments considered in this survey are formally introduced in Section 2.2.

The languages of this survey can thus be thought of as subsets of XPath 1.0 capturing the more important features of the language. However, even in our sub-languages we make some small departures from the concrete syntax of XPath 1.0. We do this because one of our goals is a clean theoretical model of XPath, amenable to stating and proving properties of the language, while XPath 1.0 syntax is often cumbersome for inductive proofs. We note these deviations from standard syntax in the text, in particular in Section 2.4, and sketch how queries in XPath 1.0 can be translated to conform to our grammar.

2.1 Data Model

A *signature* (or *vocabulary*) is a set of relation and function names. A *relational signature* is one consisting only of relation names (i.e., a relational schema). A σ -*structure* is a structure (or database) of signature σ . As a convention, given a structure \mathcal{A} , we use A (the name of the structure set in roman font) to denote its domain and $|\mathcal{A}|$ to denote the size of the structure in a reasonable machine-representation (cf. e.g. [Imm99; Lib04]).

Let Σ be a finite alphabet of labels. An *unranked ordered tree* is a tree in which nodes may have a variable number of children, with an order among them. An *XML-tree* is a relational structure \mathcal{T} of signature

$$\sigma_{nav} = ((\mathbf{Lab}_L)_{L \in \Sigma}, R_{\text{child}}, R_{\text{next-sibling}}),$$

representing an unranked, ordered tree whose nodes are labeled using the symbols from Σ : each \mathbf{Lab}_L , for $L \in \Sigma$, is a unary relation representing the set of nodes labeled L , R_{child} is the binary parent-child relation among nodes, and $R_{\text{next-sibling}}$ is the binary immediate right-sibling relation. That is, $R_{\text{child}}(x, y)$ means that y is a child of x and $R_{\text{next-sibling}}(x, y)$ means that y is the immediate right-sibling of x . We say that an XML-tree \mathcal{T} of signature σ_{nav} represents the navigational structure of an XML document.

An *XML document* is a structure of signature $\sigma_{dom} = \sigma_{nav} \cup \{\@A_1, \dots, \@A_n\}$ over a two-sorted domain of nodes and values, where the relations from σ_{nav} over nodes are as above and the $\@A_1, \dots, \@A_n$ are a fixed finite set of associated *attribute functions*, which map nodes to values. For simplicity we assume the attribute functions to be total and to take values in the integers. We use $Node(\mathcal{D})$ to mean the nodes of XML document \mathcal{D} ; since \mathcal{D} is usually clear from the context, we will generally write simply *Node*. Similarly, we write $NodeSet(\mathcal{D})$ for the set of all sets of nodes of document \mathcal{D} , omitting the argument \mathcal{D} when it is clear.

Navigational Primitives. In XPath, the primitives employed for navigation along the tree structure of a document are called *axes*. We will consider the axes *self*, *child*, *parent*, *descendant*, *descendant-or-self*, *ancestor*, *ancestor-or-self*, *next-sibling*, *following-sibling*, *previous-sibling*, *preceding-sibling*, *following*, and finally *preceding*. The meaning of axis α is best given by a binary *axis relations* R_α , where R_{child} and $R_{\text{next-sibling}}$ were introduced above, $R_{\text{self}} = \{(n, n) : n \in Node\}$, $R_{\text{descendant}}$ is the transitive closure of R_{child} , $R_{\text{descendant-or-self}}$ is the reflexive and transitive closure of R_{child} , $R_{\text{following-sibling}}$ is the transitive closure of $R_{\text{next-sibling}}$, and finally, $R_{\text{following}}$ is the composition $R_{\text{ancestor-or-self}} \circ R_{\text{following-sibling}} \circ R_{\text{descendant-or-self}}$. By the *inverse* of a binary relation R , we refer to the relation $\{(n', n) : R(n, n')\}$. The relations R_{parent} , R_{ancestor} , $R_{\text{ancestor-or-self}}$, $R_{\text{preceding-sibling}}$, $R_{\text{previous-sibling}}$, and $R_{\text{preceding}}$ are the inverses of the relations R_{child} , $R_{\text{descendant}}$, $R_{\text{descendant-or-self}}$, $R_{\text{next-sibling}}$, $R_{\text{following-sibling}}$, and $R_{\text{following}}$, respectively. We say that an axis α is the inverse of an axis β iff R_α is the inverse of R_β .

2.2 XPath Fragments Considered in this Survey

Navigational XPath. Many results on XPath apply to the fragment that deals only with the navigational structure of an XML document. This fragment, denoted **NavXPath**, consists of expressions whose input is a node and whose output is either a set of nodes (an element of $NodeSet$) or a Boolean. The latter are also referred to as *qualifiers* or *filters*. We will generally use $p, p' \dots$ to vary over general XPath expressions, of any type, while $q, q' \dots$ will be used to denote qualifiers. Expressions are built up from the grammar

$$\begin{aligned} p &::= step \mid p/p \mid p \cup p \\ step &::= axis \mid step[q] \\ q &::= p \mid lab() = L \mid q \wedge q \mid q \vee q \mid \neg q, \end{aligned}$$

where *axis* stands for the axes named above, L denotes the labels in Σ , and \wedge, \vee, \neg stand for *and* (conjunction), *or* (disjunction) and *not* (negation), respectively.

An expression p in **NavXPath** over a σ_{nav} -structure \mathcal{D} is interpreted as a function $\llbracket p \rrbracket_{NodeSet}$ from a node to a set of nodes, while a qualifier q is interpreted as a unary predicate $\llbracket q \rrbracket_{Boolean} : Node \rightarrow NodeSet$. In both cases, we refer to the input node to these functions as the *context node*. The semantic functions are defined inductively on the structure of p, q . For $NodeSet$ expressions p we have

$$\begin{aligned} (P1) \quad &\llbracket axis \rrbracket_{NodeSet}(n) := \{n' : R_{axis}(n, n')\}. \\ (P2) \quad &\llbracket step[q] \rrbracket_{NodeSet}(n) := \{n' : n' \in \llbracket step \rrbracket_{NodeSet}(n) \wedge \llbracket q \rrbracket_{Boolean}(n') = \text{true}\}. \\ (P3) \quad &\llbracket p_1/p_2 \rrbracket_{NodeSet}(n) := \{v : \exists w \in \llbracket p_1 \rrbracket_{NodeSet}(n) \wedge v \in \llbracket p_2 \rrbracket_{NodeSet}(w)\}. \\ (P4) \quad &\llbracket p_1 \cup p_2 \rrbracket_{NodeSet}(n) := \llbracket p_1 \rrbracket_{NodeSet}(n) \cup \llbracket p_2 \rrbracket_{NodeSet}(n). \end{aligned}$$

For qualifiers q we have

$$\begin{aligned} (Q1) \quad &\llbracket lab() = L \rrbracket_{Boolean}(n) :\Leftrightarrow \text{Lab}_L(n) \\ (Q2) \quad &\llbracket p \rrbracket_{Boolean}(n) :\Leftrightarrow \llbracket p \rrbracket_{NodeSet}(n) \neq \emptyset \\ (Q3) \quad &\llbracket q_1 \wedge q_2 \rrbracket_{Boolean}(n) :\Leftrightarrow \llbracket q_1 \rrbracket_{Boolean}(n) \wedge \llbracket q_2 \rrbracket_{Boolean}(n) \\ (Q4) \quad &\llbracket q_1 \vee q_2 \rrbracket_{Boolean}(n) :\Leftrightarrow \llbracket q_1 \rrbracket_{Boolean}(n) \vee \llbracket q_2 \rrbracket_{Boolean}(n) \\ (Q5) \quad &\llbracket \neg q \rrbracket_{Boolean}(n) :\Leftrightarrow \neg \llbracket q \rrbracket_{Boolean}(n) \end{aligned}$$

First-Order XPath (FOXPath). We now extend the above language to allow queries that can look at the data value structure of an input document of signature σ_{dom} . **FOXPath** adds path expressions of the form

$$\text{id}(p/@A)$$

and qualifiers of the forms

$$i \text{ RelOp } i \qquad p/@A \text{ RelOp } i \qquad p/@A \text{ RelOp } p'/@B$$

to the syntax of **NavXPath**, where p and p' are path expressions, $@A$ and $@B$ are attributes, $\text{RelOp} \in \{=, \leq, <, >, \geq, \neq\}$, and i is a nonterminal denoting the constant integers.

FOXPath operates on σ_{dom} -structures with an attribute function $@ID$. The $\text{id}(p/@A)$ expressions model the $\text{id}()$ function of XPath, and to be fully faithful we could assume that the attribute function $@ID$ is injective.

The semantic functions $\llbracket \cdot \rrbracket_{NodeSet} : Node \rightarrow NodeSet$ and $\llbracket \cdot \rrbracket_{Boolean} : Node \rightarrow Boolean$ of **NavXPath** are extended as follows to handle the additional constructs:

$$\begin{aligned} (P5) \quad &\llbracket \text{id}(p/@A) \rrbracket_{NodeSet}(n) := \{n' : \exists n'' \in \llbracket p \rrbracket_{NodeSet}(n) \ @ID(n') = A(n'')\}, \\ (Q6) \quad &\llbracket i \text{ RelOp } i' \rrbracket_{Boolean}(n) :\Leftrightarrow \llbracket i \rrbracket_{Int}(n) \text{ RelOp } \llbracket i' \rrbracket_{Int}(n), \\ (Q7) \quad &\llbracket p/@A \text{ RelOp } i \rrbracket_{Boolean}(n) :\Leftrightarrow \exists n' \in \llbracket p \rrbracket_{NodeSet}(n) \ A(n') \text{ RelOp } \llbracket i \rrbracket_{Int}(n), \text{ and} \\ (Q8) \quad &\llbracket p/@A \text{ RelOp } p'/@B \rrbracket_{Boolean}(n) :\Leftrightarrow \exists n' \in \llbracket p \rrbracket_{NodeSet}(n) \ \exists n'' \in \llbracket p' \rrbracket_{NodeSet}(n) \\ &\quad A(n') \text{ RelOp } B(n''), \end{aligned}$$

where $\llbracket c \rrbracket_{Int}(n) = c$ for constant c .

Aggregate XPath (AggXPath). Next, we add on expressions to FOXPath that manipulate integers and compute aggregates.

The syntax of AggXPath is obtained from FOXPath by extending number-typed qualifiers i (from exclusively integer constants in FOXPath) to

$$i ::= 'c' \mid i + i \mid i * i \mid \text{count}(p) \mid \text{sum}(p/@A)$$

where p ranges over path expressions and $@A$ is an attribute function. We call “+” and “*” *arithmetic operators* and “count” and “sum” *aggregate operators*.

The semantic function $\llbracket i \rrbracket_{Int} : Node \rightarrow Int$ for numerical expressions of FOXPath is extended to

- (I1) $\llbracket c \rrbracket_{Int}(n) := c$
- (I2) $\llbracket i \circ i' \rrbracket_{Int}(n) := \llbracket i \rrbracket_{Int}(n) \circ \llbracket i' \rrbracket_{Int}(n) \quad (\circ \in \{+, *\})$
- (I3) $\llbracket \text{count}(p) \rrbracket_{Int}(n) := |\llbracket p \rrbracket_{NodeSet}(n)|$
- (I4) $\llbracket \text{sum}(p/@A) \rrbracket_{Int}(n) := \Sigma\{\@A(n') \mid n' \in \llbracket p \rrbracket_{NodeSet}(n)\}$

Aggregate XPath with position arithmetic (OrdXPath). Finally, we add the numerical operations “position()” and “last()” to AggXPath; these are called *positional operators*.

If we look at the semantic functions $\llbracket \cdot \rrbracket_{NodeSet}$ and $\llbracket \cdot \rrbracket_{Boolean}$ of AggXPath, we say that they map from a context node (e.g., the root node of the document tree) to either a node set, a Boolean, or an integer value. In OrdXPath, qualifiers and numerical expressions are defined with respect to a more extensive “context” consisting of a node and two additional integers, which can be accessed by the positional operators.

- (1) $\llbracket \cdot \rrbracket_{NodeSet} : Node \rightarrow NodeSet$ is as in AggXPath except for

$$(P2') \quad \llbracket \text{step}[q] \rrbracket_{NodeSet}(n) := \{n_j \mid \llbracket \text{step} \rrbracket_{NodeSet}(n) = \{n_1, \dots, n_k\} \wedge n_1 \prec n_2 \prec \dots \prec n_k \wedge 1 \leq j \leq k \wedge \llbracket q \rrbracket_{Boolean}(n_j, j, k)\},$$

where \prec denotes the *document order*, i.e. the total order

$$n \prec n' \Leftrightarrow R_{\text{descendant}}(n, n') \vee R_{\text{following}}(n, n');$$

- (2) $\llbracket \cdot \rrbracket_{Boolean} : Node \times Int \times Int \rightarrow Boolean$ is defined analogously to $\llbracket \cdot \rrbracket_{Boolean}$ of AggXPath, however taking a context consisting of a triple (n, j, k) and passing it on to all qualifier and numerical subexpressions (for instance, $\llbracket q_1 \wedge q_2 \rrbracket_{Boolean}(n, j, k) \Leftrightarrow \llbracket q_1 \rrbracket_{Boolean}(n, j, k) \wedge \llbracket q_2 \rrbracket_{Boolean}(n, j, k)$), and
- (3) $\llbracket \cdot \rrbracket_{Int} : Node \times Int \times Int \rightarrow Int$ is defined analogously to $\llbracket \cdot \rrbracket_{Int}$ of AggXPath, however passing on the full context triple (n, j, k) to its numerical subexpressions (for instance, $\llbracket i + i' \rrbracket_{Int}(n, j, k) := \llbracket i \rrbracket_{Int}(n, j, k) + \llbracket i' \rrbracket_{Int}(n, j, k)$). For the new operators of OrdXPath, we have:
 - (I5) $\llbracket \text{position}() \rrbracket_{Int}(n, j, k) := j$
 - (I6) $\llbracket \text{last}() \rrbracket_{Int}(n, j, k) := k$

By *positive* FOXPath, denoted PFOXPath, (resp., NavXPath, denoted PNavXPath), we will refer to FOXPath (resp., NavXPath) without negation and inequalities (i.e., expressions $p \text{ RelOp } p'$ with $\text{RelOp} \neq "="$). We say that a FOXPath query (resp., NavXPath query) is *conjunctive* (and *connected*) if it does not use disjunction, union, negation, or inequalities.

REMARK 2.1. The XPath fragments just presented – just like XPath 1.0 – allow for multiple qualifier brackets as part of a step expression. Steps containing multiple qualifier brackets $axis[\dots]$ can be simplified to $axis[\dots]$ in all our XPath

languages except for `OrdXPath`. In the proofs of our survey, we will sometimes assume this simplified syntax for convenience.

In `OrdXPath` this simplification is not applicable in general, and hence for this fragment the ability to use multiple qualifiers allowed in XPath 1.0 does add expressiveness.

EXAMPLE 2.2. On a context node n with three children n_1, n_2, n_3 , of which the first is labeled B and the second and third are labeled A ,

$$\llbracket \text{child}[\text{lab}() = A][\text{position}() = 1] \rrbracket_{NodeSet}(n) = \{n_2\},$$

since n_2 is the first child of n in document order that is labeled A . This query cannot be phrased with a single qualifier bracket in each step. For instance,

$$\begin{aligned} \llbracket \text{child}[\text{lab}() = A \wedge \text{position}() = 1] \rrbracket_{NodeSet}(n) = \\ \{n_j \mid 1 \leq j \leq 3 \wedge \llbracket \text{lab}() = A \wedge \text{position}() = 1 \rrbracket_{Boolean}(n_j, j, 3)\} = \emptyset, \end{aligned}$$

while

$$\begin{aligned} \llbracket \text{child}[\text{lab}() = A]/\text{self}[\text{position}()=1] \rrbracket_{NodeSet}(n) = \\ \bigcup \{ \llbracket \text{self}[\text{position}()=1] \rrbracket_{NodeSet}(n_i) \mid n_i \in \llbracket \text{child}[\text{lab}() = A] \rrbracket_{NodeSet}(n) \} = \\ \llbracket \text{self}[\text{position}()=1] \rrbracket_{NodeSet}(n_2) \cup \llbracket \text{self}[\text{position}()=1] \rrbracket_{NodeSet}(n_3) = \{n_2, n_3\}. \end{aligned}$$

□

Similarly, while $p/\text{following}[q_1] \dots [q_k]/p'$ is equivalent to

$$p/\text{ancestor-or-self}/\text{following-sibling}/\text{descendant-or-self}[q_1] \dots [q_k]/p'$$

if q_1, \dots, q_k are `AggXPath`, the following axis is not redundant in `OrdXPath`.

2.3 Query Equivalence

By a *query*, we mean any expression from one of the XPath fragments introduced above. Two queries with domain *Node* p and p' are *fully equivalent* (or simply *equivalent* when it is clear from the context), denoted by $p \equiv p'$, iff for any XML document \mathcal{D} and all nodes $n \in \mathcal{D}$, $\llbracket p \rrbracket_{NodeSet}(n) = \llbracket p' \rrbracket_{NodeSet}(n)$, and similarly for `OrdXPath` queries with context $Node \times Int \times Int$.

Let `true` be a shortcut for the qualifier $(\text{lab}() = A) \vee \neg(\text{lab}() = A)$. We say two queries are *equivalent over* Σ_0 (denoted by \equiv_{Σ_0}) where Σ_0 is a fixed finite label alphabet, if the above holds for any document \mathcal{D} whose labels are in Σ_0 . For example, `true` is equivalent to $\text{lab}() = A \vee \text{lab}() = B$ over the alphabet $\{A, B\}$, but not in general. We will usually work with the stronger notion of general equivalence \equiv , and specify when results also hold for restricted equivalence – equivalence w.r.t. some finite alphabet Σ_0 .

For queries with domain *Node* (which include all `NavXPath` expressions), a weaker equivalence relation is defined as follows: p and p' are called *root equivalent*, denoted by $p \equiv_r p'$, iff for any XML document \mathcal{D} , $\llbracket p \rrbracket_{NodeSet}(rt) = \llbracket p' \rrbracket_{NodeSet}(rt)$, where rt is the root of \mathcal{D} . For `NavXPath` queries defined using upward axes, root equivalence can be weaker than general equivalence: for example $\text{self}[\text{parent}] \equiv_r \text{self}[\neg \text{true}]$, since the root node has no parent, but clearly these two expressions are not fully equivalent.

2.4 Discussion: Faithfulness to the XPath Standard

We now discuss the distinctions between the model above and standard XPath 1.0 syntax.

§1. *Node tests*:. In XPath 1.0, label tests $\text{lab}() = L$ can be carried out in qualifiers using the “name” function or via paths $\text{self}::L$. The standard syntax can clearly be mimicked in our language, and vice versa. We will often write $\text{axis}::L[q]$, corresponding to XPath standard syntax for $\text{axis}[\text{lab}() = L][q]$ and $\text{axis}::*[q]$ for $\text{axis}[q]$.

§2. *Root slash*:. XPath 1.0 supports path expressions of the form $/p$ (even as qualifiers); For instance, $/\text{child}::A[\text{child}::B]$ selects the A -labeled children of the root node iff the root node has at least one child labeled B . This root slash can be considered a special axis **root** definable in NavXPath as $\text{ancestor-or-self}[-\text{parent}]$. The omission of the **root** axis thus has no impact on results about NavXPath, although it has some consequences for characterization theorems within NavXPath, as will be noted below.

§3. *Axes*:. XPath 1.0 does not have the **next-sibling** or **previous-sibling** axes. They can be defined from **following-sibling** and **preceding-sibling** using $\text{position}()$ within **AggXPath**. It will follow from the characterization of NavXPath and known results on two-variable logic that these axes do add expressiveness to NavXPath. We found it more natural to include these axes as primitives: their inclusion does not impact complexity bounds, and the characterizations of expressiveness have variations both with these axes and without them. On the other hand, XPath 1.0 has an **attribute** axis. In order to distinguish the navigational fragment of XPath more clearly, we treat attribute access as a separate feature rather than as an axis, but this does not affect expressiveness.

§4. *Document order and position arithmetic*:. The interpretation steps $\text{axis}[q]$ with qualifier q employing “ $\text{position}()$ ” depends on whether *axis* is a so-called *forward axis* (**child**, **descendant**, **following**, ...) or a *backward axis* (**parent**, **ancestor**, **preceding-sibling**, ...). In the former case, just as in our definition, the position of a node in a set is determined in document order \prec ; however, for backward axes, positions of nodes are determined with respect to the inverse of \prec . This can be dealt with by replacing “ $\text{position}()$ ”, for qualifiers of backward axis steps, by “ $\text{last}() + 1 - \text{position}()$ ”.

§5. *Union*:. A significant distinction concerns the union operator \cup ; in XPath 1.0 this is allowed only at top-level, while we permit it as a binary operator on arbitrary *NodeSet* expressions. Let us call *unnested OrdXPath* the language **OrdXPath** from above, but with union only permitted at top-level, and similarly refer to **unnested NavXPath** and **unnested FOXPath**. Every **FOXPath** expression can be translated into an **unnested** one, and similarly for **NavXPath**. One simply pushes unions upward – e.g. $p/(p_1 \cup p_2)/p' = p/p_1/p' \cup p/p_2/p'$. There is an exponential blow-up in passing from one to the other; consider paths

$$(\text{child}::A \cup \text{child}::B)/(\text{child}::A \cup \text{child}::B)/\dots/(\text{child}::A \cup \text{child}::B).$$

The inclusion of unrestricted nesting can thus have an impact on complexity and succinctness. Prior theoretical studies [Mar04a; BFK03] allow arbitrary nesting. Arbitrary nesting gives a cleaner language, and many of the most important results about XPath hold for the more general language (e.g. expressive equivalences, upper and lower bounds on evaluation complexity). We do the same here, but we will distinguish some results that hold only for the **unnested** variant.

§6. *Data Model*:. Clearly, any aspects of the XPath data model for documents are not present in our formalization of XML documents. We do not, for example, concern ourselves with the distinction between types of nodes (**comment**, **processing-instruction**, etc.). These features could be modeled on top of the data model we present here (e.g. using distinct sets of element tags to model the different types).

In the XPath standard [Wor99], the document root node has precisely one child node, which is the root of the visible XML tree (the topmost XML *element node*). Without loss of generality, we may ignore this subtlety and may think of any query $/p$ as $/\text{child}/p$.

2.5 Historical and Bibliographic Remarks

XPath was initially developed by James Clark and formalized and promulgated as an independent standard by the W3C starting in 1999, as XPath 1.0 [Wor99]. The standard defines the syntax of the language, along with use cases, but gives the semantics only informally. An early attempt to give a formal semantics is found in [Wad00; Wad99]. A complete and yet very concise formal semantics of XPath 1.0 can be found in [GKP02].

In the process of the development of XQuery, a significant extension of XPath 1.0 was developed, released as XPath 2.0 [Wor02]. XPath 2.0 is the result of the integration of XPath and XQuery into a common syntax and semantics definition, and its semantics is presented as part of the XQuery 1.0 Formal Semantics [Wor02]. XPath 2.0 is a radically different language from XPath 1.0, including variables and explicit quantification. From a theoretical perspective, no polynomial time bounds can be given on basic problems like XPath 2.0 evaluation (while this is possible for XPath 1.0, see Section 4). From a practical point of view the breadth of XPath 2.0 and XQuery would require discussion to subsume nearly every aspect of general-purpose program optimization and analysis.

The extensions of XPath 2.0 over XPath 1.0 are mostly by programming language constructs that do not preserve the theoretical properties of XPath pointed out in the introduction. The largest language studied in this article, **OrdXPath**, is in the intersection of XPath 1.0 and 2.0 and yet subsumes all the XPath fragments for which fundamental results have been presented in the literature.

3. EXPRESSIVENESS

We now investigate where XPath “fits” in terms of other formalisms for querying trees and tree-structured data. One natural benchmark is first-order logic (FO), but we will also consider Monadic Second Order logic (MSO), the existential fragment of FO ($\exists FO$), the positive existential fragment of FO ($\exists^+ FO$) and the fragment FO^k of FO formulas that use at most k distinct variables. The semantics of these languages is standard [Lib04]. For a logical language \mathcal{L} , we will use $\mathcal{L}[\sigma]$ to denote the formulas of \mathcal{L} over vocabulary σ .

3.1 Expressiveness of NavXPath

We start by investigating how NavXPath compares to first-order logic over the navigational structure of XML documents. Note that a formula of first-order logic with two free variables can be thought of as defining a mapping from *Node* to *NodeSet*, while a formula with one free variable defines a mapping from *Node* to Boolean. The semantics of NavXPath presented in Section 2.2 gives already a translation into these first-order languages.

Let σ_{transnav} be the vocabulary extending σ_{nav} with $R_{\text{descendant}}$ and $R_{\text{following-sibling}}$. Then,

PROPOSITION 3.1. *For every NavXPath expression e , one can find (in linear time) a corresponding formula ϕ in $FO[\sigma_{\text{transnav}}]$ fully equivalent to e . Furthermore,*

- $\phi \in FO[(\text{Lab}_L)_{L \in \Sigma}, R_{\text{child}}]$ if e uses only child and parent axes,
- $\phi \in FO[(\text{Lab}_L)_{L \in \Sigma}, R_{\text{descendant}}]$ if e uses only upward and downward axes, and
- $\phi \in FO[\sigma_{\text{nav}}]$ if e uses only child, parent, next-sibling, previous-sibling.

Note that this proposition holds both for path expressions *returning* nodesets (in this case ϕ has two free variables) and for those returning Boolean expressions (here ϕ has one free variable).

However, this is not an *exact* characterization of the expressiveness of NavXPath. It is easy to find first-order queries over trees that are not expressible in NavXPath: for example, the query that asks whether the tree has two nodes labeled C that are in an ancestor relationship, and such that all nodes between them are labeled B . We now show that NavXPath corresponds precisely to two-variable logic.

We introduce a normal form for FO^2 queries over vocabulary $\sigma_{transnav}$. $XPNF$ is the set of queries that are disjunctions of formulas $\gamma(z_1, z_n)$ of the form:

$$\exists z_2 \dots \exists z_{n-1} \rho_1(z_1) \wedge \chi_1(z_1, z_2) \wedge \rho_2(z_2) \wedge \dots \wedge \chi_{n-1}(z_{n-1}, z_n) \wedge \rho_n(z_n)$$

where the z_i here are distinct variables, the ρ_i are FO^2 formulae, and the $\chi_i(z_i, z_{i+1})$ are unions of binary atomic formulas over predicates from $\sigma_{transnav}$.

THEOREM 3.2 [MDR04]. *NavXPath corresponds to FO^2 in expressiveness, in the following sense.*

- *For every NavXPath expression returning a Boolean there is a corresponding fully equivalent expression in FO^2 , and for every FO^2 expression there is a corresponding fully equivalent NavXPath expression.*
- *For every NavXPath expression returning a NodeSet, there is a corresponding expression in $XPNF$ and vice versa.*

Proof (Sketch). We first show the direction from NavXPath *NodeSet* expressions to $XPNF$ and from NavXPath Boolean expressions to FO^2 . We will restrict to *unnested* NavXPath expressions, that is, NavXPath expressions that have union only at top-level. These have the same expressiveness as general NavXPath expressions. Since the target classes FO^2 and $XPNF$ are closed under disjunction, it suffices to translate expressions that have no occurrence of the union operator. So it suffices to show that all NavXPath *NodeSet* expressions that do not use the union operator translate to $XPNF$ expressions without top-level disjunction, and every NavXPath Boolean expression that does not use the union operator translates to an FO^2 expression. We show this pair of statements by simultaneous induction. The base case for $\text{lab}() = A$ is simple, as is the case for Boolean operations in Boolean expressions (since FO^2 is closed under Boolean operators). The case *step*[q] can be translated into $XPNF$ formula $\chi(x, y) \wedge \phi(y)$, where χ is a $XPNF$ formula without top-level disjunction formed inductively for *step*, and ϕ is an FO^2 formula formed for q . We now do the inductive proof for $p = p_1/p_2$. By induction, we assume we have $XPNF$ formulas (without top-level disjunction) γ_1 equivalent to p_1 and γ_2 equivalent to p_2 . If we have

$$\gamma_1 = \exists z_2 \dots \exists z_{m-1} \left(\bigwedge_{i=1}^{m-1} \rho'_i(z_i) \wedge \chi_i(z_i, z_{i+1}) \right) \wedge \rho'_m(z_m)$$

and

$$\gamma_2 = \exists z_m \dots \exists z_{n-1} \left(\bigwedge_{i=m}^{n-1} \rho''_i(z_i) \wedge \chi_i(z_i, z_{i+1}) \right) \wedge \rho''_n(z_n)$$

then we can write γ_1/γ_2 as

$$\exists z_2 \dots \exists z_{n-1} \left(\bigwedge_{i=1}^{n-1} \rho_i(z_i) \wedge \chi_i(z_i, z_{i+1}) \right) \wedge \rho_n(z_n) \quad (1)$$

where $\rho_i(z_i)$ is $\rho'_i(z_i)$ for $i < m$, $\rho'_i(z_i) \wedge \rho''_i(z_i)$ for $i = m$, and $\rho''_i(z_i)$ for $i > m$.

The other interesting inductive case is that of qualifiers of the form p . By induction we have a *XPNF* formula γ representing p . We will assume $\gamma(z_1, z_n)$ to be as shown in equation (1).

We need to show that the formula $\exists z_n \gamma(z_1, z_n)$ is in FO^2 . Suppose that n is odd (the case where n is even is similar). Let $var(i) = z_1$ for i odd and z_2 for i even. Let $\phi([x \mapsto y])$ denote the formula obtained by substituting all occurrences of variable x by y in ϕ . Define $\psi_n = \rho_n([z_n \mapsto var(n)])$ and $\psi_{i-1} = \rho_{i-1}([z_{i-1} \mapsto var(i-1)]) \wedge \exists var(i) \chi_i(var(i-1), var(i)) \wedge \psi_i$. Then ψ_i is an FO^2 sentence with $var(i)$ free. We can verify that ψ_1 is equivalent to $\exists z_n \gamma(z_1, z_n)$.

The converse direction is to show by induction that formulas in *XPNF* can be translated to *NavXPath NodeSet* expressions, while FO^2 formulas with one free variable can be translated to *NavXPath* Boolean expressions. Since the first statement follows easily from the second, we focus on the proof of the second. The translation function T is formed by induction on the structure of an FO^2 formula. The atomic cases are straightforward, as are the Boolean operations. The interesting case is $\exists y \beta(x, y)$, where β is in FO^2 . Formula β can be assumed to be a Boolean combination of atomic binary formulas and FO^2 formulas in one free variable of lower quantifier rank. Let β' be a formula equivalent to β obtained by turning β into Disjunctive Normal Form (DNF) and replacing each disjunct $\phi(x, y)$ that does not contain a binary atom by $(\phi(x, y) \wedge x = y) \vee (\phi(x, y) \wedge x \neq y)$. This preserves the DNF.

The atomic binary predicates in β' are either equality, inequality, or axis relations; however, equality $x = y$ can be replaced by $\text{self}(x, y)$, and an inequality $x \neq y$ can be replaced by a disjunction of four axis relations (y is either and ancestor or descendant of x or follows or precedes x). Let β'' be obtained by applying these substitutions to β' and again turning the formula into DNF.

Since two axis predicates are either inconsistent with one another (i.e., the axis relations have an empty intersection) or subsume each other, we can assume $\beta''(x, y)$ to be of the form

$$\bigvee_i \phi_i(x) \wedge R_{\chi_i}(x, y) \wedge \psi_i(y),$$

that is, each disjunct contains precisely one binary atom.

We can easily translate $\beta''(x, y)$ into *NavXPath* as

$$T(\beta'') := \bigcup_i \text{self}[T(\phi_i)] / \chi_i[T(\psi_i)].$$

□

We note that the argument from *NavXPath* to FO^2 shows that there is a polynomial time translation from unnested *NavXPath* to FO^2 ; for general *NavXPath* expressions the best translation we know of is in exponential time. This mapping introduces atomic predicates in the output corresponding only to axes mentioned in the input; hence *NavXPath* filters without the *next-sibling* or *previous-sibling* axes map to FO^2 formulas that do not use (atomic relations for) these axes.

In the direction from FO^2 to *NavXPath*, the translation also yields an output that is exponential in the input in the worst case, and this has been shown to be unavoidable. See [Mdr04] for discussion and proof of this; we will give a further argument that there is no polynomial translation in Section 5.¹ This direction does introduce new axes. The sibling axes may appear in the output even when the original formula mentions only the *child* axis; the *XPNF* formula $x \neq y$ cannot be translated into *NavXPath* unless the sibling axis is present. Similarly, transitive axes are introduced in the translation. On the other hand, *next-sibling* and *previous-sibling*

¹Although the argument there is relative to a complexity-theoretic assumption.

are not introduced in this translation unless the corresponding atomic predicates occur in the input. Hence `NavXPath` filters without these axes correspond exactly to FO^2 formulas that do not have atomic relations for these axes. Since it is known that a successor relation of a linear order cannot be expressed in FO^2 over the signature whose only binary predicate is for the linear order (see e.g. [EVW02]), it follows from these translations that `NavXPath` cannot express `next-sibling` or `previous-sibling`.

We now turn to the consequences of this characterization for closure properties of `NavXPath`. It is clear that `NavXPath` qualifiers are closed under Boolean operations, since we have explicit operators for these; it can also be seen to follow from Theorem 3.2, since FO^2 is obviously Boolean closed. What about the closure properties of `NavXPath` expressions? In [Mar05], the following is shown:

THEOREM 3.3 [MAR05]. *NavXPath expressions returning nodesets are closed under intersection and union, but not under complement.*

Closure under union is obvious, since `NavXPath` has a built-in union operator. Closure under intersection follows from the fact that the conjunction of acyclic conjunctive queries is still a conjunctive query, and every conjunctive query on trees can be transformed into an equivalent union of acyclic conjunctive queries [BFK03; GKS04] (cf. Theorem 3.7), and unions of acyclic conjunctive queries can be easily translated into `NavXPath`.

The lack of closure under complementation may seem surprising. In fact, [Mar05] shows a stronger result: any extension of `NavXPath` closed under complementation can express all first-order properties. The proof is by showing that an “until” operator can be defined by complementing `NavXPath` expressions. The following example is taken from page 7 of [Mar05]: Let $\phi(x, y)$ hold iff y is an A -labeled descendant of x and every descendant of x that is an ancestor of y is labeled B . Then ϕ is expressible in `NavXPath` extended with a complement operator $(\cdot)^c$ as:

$$\text{descendant}[\text{lab}() = A] \cap (\text{descendant}[\text{lab}() \neq B] / \text{descendant})^c$$

Above, we use also the intersection operator \cap , but this can easily be defined using complementation and union.

The translation of unnested `NavXPath` to FO^2 can be extended as follows: let NavXPath^\cap be the extension of `NavXPath` with the intersection operator \cap , and let *unnested* NavXPath^\cap be the same but with union allowed only at top-level. By Theorem 3.3 above, we have NavXPath^\cap has the same expressiveness as `NavXPath` (for both expressions and qualifiers). Hence NavXPath^\cap qualifiers have the same expressiveness as FO^2 formulas. Using the argument of [OMFB02], one can show that even unnested NavXPath^\cap formulas can be exponentially more succinct than `NavXPath` formulas. However, unnested NavXPath^\cap formulas can still be translated into FO^2 efficiently:

PROPOSITION 3.4. *There is a polynomial time function producing for each unnested NavXPath^\cap filter an equivalent FO^2 formula $\phi(x)$.*

Proof. We extend the dual translations from the proof of Theorem 3.2 to go from NavXPath^\cap *NodeSet* expressions without union to *XPNF* queries and from `NavXPath` Boolean expressions without union to FO^2 queries. We use exactly the same construction of a translation function, let us call it f , as for `NavXPath`, but for the inductive step for $f(E_1 \cap E_2)$ we translate into $f(E_1) \wedge f(E_2)$. \square

We now provide an example of a navigational FO query that we prove not to be expressible in `NavXPath`. Our example, a new `immediately-following` axis, has a practical motivation. Computational linguists have proposed the addition of such an axis to `XPath` to ask practical queries on linguistic trees [BCD⁺05]. We can give a semantics to this axis using a corresponding binary relation $R_{\text{immediately-following}}$,

which holds of (x, y) iff

$$R_{\text{following}}(x, y) \wedge \neg \exists z (R_{\text{following}}(x, z) \wedge R_{\text{following}}(z, y)).$$

In [BCD⁺05] an extension of XPath with immediately-following is proposed. We show here the following:

PROPOSITION 3.5. *There is no NavXPath expression E equivalent as a nodeset query to immediately-following.*

Proof. Consider instances that obey the DTD D_0 given as

$$A \Rightarrow A(B|C|\epsilon) \mid \epsilon \quad B \Rightarrow \epsilon \quad C \Rightarrow \epsilon$$

with A being the root.

An instance of this DTD consists of a chain of A elements, with one of the following holding for each element x :

- (1) x has a single A child (the next element of the chain), and no other children,
- (2) x has no children (i.e. it is the lowest element of the chain),
- (3) x has a single A child and a single B child, or
- (4) x has a single A child and a single C child.

There is a NavXPath qualifier Q_0 that holds of the root of a document iff the document is valid with respect to D_0 . Consider the qualifier Q_1

$$\text{lab}() = A \wedge \text{immediately-following}[\text{lab}() = B]$$

in NavXPath extended with immediately-following.

That is, Q_1 holds of an A node iff it has an immediately-following node that is a B . For a node n in an instance satisfying D_0 , Q holds at n iff the first ancestor of n which has a non- A child has a B child. We claim that there is no NavXPath qualifier equivalent to $Q_1 \wedge Q_0$. From this, the proposition follows. From Theorem 3.2, it suffices to show that no two-variable logic formula can express $Q_1 \wedge Q_0$.

We will reduce expressibility of $Q_1 \wedge Q_0$ over trees to a statement about expressibility of a certain property in two-variable logic over strings. Let FO^- be the logic built up using quantification only over A nodes, where the vocabulary includes the binary predicates $R_{\text{descendant}}$ and R_{child} and unary predicates P_1, P_2, P_3, P_4 , where P_i holds of x iff case i holds above.

CLAIM 3.6. *For every $FO[\sigma_{\text{transnav}}]$ sentence $\phi(x)$ there is an FO^- sentence $\phi^-(x)$ with the same number of variables as ϕ which is equivalent to ϕ over all A -nodes within all documents satisfying D_0 .*

Informally, ϕ^- is obtained inductively by replacing variables over B, C nodes by variables over their A parents. A sentence $\phi = \exists x B(x)$ would map to $\phi^- = \exists x \in A P_3(x)$. Formally, we proceed as follows. Let $\text{SecChild}(D, x)$ be the partial function on nodes of D that maps a node labeled A to its second child, if such a child exists, and $\text{Self}(D, x)$ be the identity function on nodes labeled A . We create a function $T(\phi, b)$ for $\phi \in FO[\sigma_{\text{transnav}}]$, b a function from the free variables of ϕ to either SecChild or Self , returning a formula $\phi' \in FO^-$ with the same free variables as ϕ , and such that: for all documents D , $T(\phi(x, y), b)$ holds of A nodes m, n iff $\phi(x, y)$ holds when applied to $b(D, m), b(D, n)$, and similarly for $\phi(x), \phi(y)$.

The main atomic cases for T are:

- $T(R_{\text{next-sibling}}(x, y), b)$ is $(P_3(y) \vee P_4(y)) \wedge R_{\text{child}}(y, x)$ if $b(x) = \text{Self}$ and $b(y) = \text{SecChild}$, and is *false* otherwise.
- $T(R_{\text{child}}(x, y), b)$ is $R_{\text{child}}(x, y)$ if $b(x) = \text{Self}$ and $b(y) = \text{Self}$, is $(P_3(x) \vee P_4(x)) \wedge x = y$ if $b(x) = \text{Self}$ and $b(y) = \text{SecChild}$, and is *false* otherwise.

- $T(R_{\text{descendant}}(x, y), b)$ is $R_{\text{descendant}}(x, y)$ if $b(x) = \text{Self}$ and $b(y) = \text{Self}$, is $P_3(y) \vee P_4(y)$ if $b(x) = \text{Self}$ and $b(y) = \text{SecChild}$, and is *false* otherwise.
- $T(B(x), b)$ is $P_3(x)$ if $b(x) = \text{SecChild}$, and is *false* otherwise.
- $T(C(x), b)$ is $P_4(x)$ if $b(x) = \text{SecChild}$ and is *false* otherwise.
- $T(A(x), b)$ is *true* if $b(x) = \text{Self}$, and is *false* otherwise.

The other atomic cases are similar. The inductive cases are:

- $T(\exists x \rho(x, y), b) = \bigvee_{b': b'|\{y\}=b} \exists x \in A T(\rho(x, y), b')$
- $T(\forall x \rho(x, y), b) = \bigwedge_{b': b'|\{y\}=b} \exists x \in A T(\rho(x, y), b')$
- $T(\phi_1 \wedge \phi_2, b) = T(\phi_1, b) \wedge T(\phi_2, b)$
- $T(\phi_1 \vee \phi_2, b) = T(\phi_1, b) \vee T(\phi_2, b)$
- $T(\neg \phi, b) = \neg T(\phi, b)$

Finally, for a sentence we let $\phi^-(x)$ be $\bigvee_b T(\phi(x), b)$, where in the disjunction b ranges over all the bindings for x . One can verify inductively that T , and hence ϕ^- has the required properties.

From this construction, we see that if $\phi(x) \in \text{NavXPath}$ expresses $Q_0 \wedge Q_1$, then $\phi^-(x)$ must hold of an A -node n iff the first ancestor of n which satisfies $P_3 \vee P_4$ satisfies P_3 . Let S_0 be the set of strings from alphabet $\Sigma = \{P_1, P_2, P_3, P_4\}$, ending with the symbol P_1 . There is an obvious bijection F from documents satisfying D_0 to strings in S_0 . Using this function, we can see that $\phi^-(x)$, considered as a predicate on strings in S_0 , holds at node n iff the first ancestor of n which satisfies $P_3 \vee P_4$ satisfies P_3 . But then by flipping the variables in every predicate $R_{\text{descendant}}$ or R_{child} in ϕ^- we obtain a two-variable formula $\phi^-(x)$ that holds at node n of string s iff the first descendant of n satisfying $P_3 \vee P_4$ satisfies P_3 . From this we easily get a contradiction of prior results about the inexpressibility of the Until operator in two variable logic (for strings, those of [EW00; EVW02], or for trees those of [Mar04b]). Consider the query Q that holds of a string s iff s has a substring that contains two nodes satisfying P_3 but none satisfying P_4 . If $\phi^-(x)$ were expressible in two-variable logic, then Q would be expressible over strings in two-variable logic over the vocabulary consisting of the labels, the descendant predicates, and the child predicate. But in [EW00] it is shown that Q (denoted there by $FAIR_2$) is not expressible in Unary Temporal Logic, and by [EVW02] Unary Temporal Logic is the same as two-variable logic over strings. Hence Q is not expressible in two-variable logic, and we have a contradiction. \square

Note that the problem of deciding whether a given FO sentence over trees is in NavXPath (i.e. is a two-variable sentence) is still open. The analogous problem for strings is known to be decidable [BP89].

3.2 Expressiveness of Fragments of NavXPath

NavXPath is still a large language, and many applications make use only of the positive fragment.

Following [BFK03], we characterize NavXPath both using logic and a visual query formalism, *tree patterns*.

A tree pattern (over label alphabet Σ) is a node and edge-labeled tree. Edges are labeled with a forward axis (*child*, *descendant*, *following-sibling*). In a *Boolean tree pattern* node labels have one component that is either a label from Σ or wildcard and another component that identifies whether a node is the distinguished *context node* or not. In a *unary tree pattern* the additional component identifies a node as either the context node, the selected node, or neither. Figure 1 shows a unary tree pattern. Following the standard convention for drawing patterns, double lines are used for a descendant edge and single lines for a child edge. A star is used

to denote the selected node, and the context node is implicitly the root node. A Boolean pattern corresponds to a Boolean query, returning true at context node n in a document iff there is a homomorphism from the pattern to the document mapping the context to n . A unary tree pattern corresponds to a *NodeSet* query, which returns node n' on input n iff there is a homomorphism from the pattern to the document which maps a node labeled context to n and the selected node to n' . The pattern in the figure is equivalent to the XPath expression

$$\text{self}[\text{lab}() = A \wedge \text{child}[\text{lab}() = B \wedge \text{descendant}[\text{lab}() = D]]]/\text{child}[\text{lab}() = C].$$

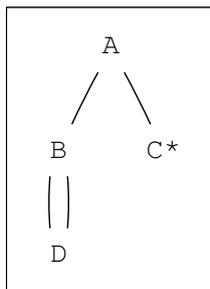


Fig. 1. Tree pattern

A finite set of tree patterns can be considered as a query, returning the union of the results of the individual patterns in the case of unary tree patterns, and returning the disjunction of the results in the case of Boolean tree patterns.

THEOREM 3.7. *The following have equal expressiveness (up to full equivalence)*

- PNavXPath *NodeSet* queries,
- \exists^+ FO formulas $\phi(c, s)$ in the signature σ_{transnav} , and
- sets of unary tree patterns.

We give a sketch of why the above holds: further details (for the case where there are only upward or downward axes, but no sideways axes such as *following* or *following-sibling*) can be found in [BFK03]; the general case is proved in [GKS04]. For every PNavXPath *NodeSet* query, and unary tree patterns, the corresponding equivalent \exists^+ FO formula can be found in linear time, simply by translating the semantics of PNavXPath or of tree patterns into logic. Translating from unary tree pattern queries to PNavXPath queries is likewise straightforward: path steps are used to traverse the path from the context node upward to the least common ancestor of the context and selected node, then downwards from this ancestor to the selected node. The existence of subtrees sprouting off from this path is asserted using filters. Translation of \exists^+ FO formulas into tree patterns is done by forming the “graph pattern” whose nodes are the variables and whose edges represent relationships between variables implied by the formula. This pattern has the same structure as a tree pattern, but does not satisfy the requirement that the underlying graph is a tree. This initial pattern is modified by recursively removing cycles in the dependency graph by identifying variables that participate in a cycle. Sibling axes are normalized in a similar fashion.

EXAMPLE 3.8. Figure 2 illustrates the query rewriting technique for transforming conjunctive queries over trees into equivalent acyclic positive queries. Consider the conjunctive query

$$Q \leftarrow R_{\text{descendant}}(x, y), R_{\text{descendant}}(x, z), R_{\text{following}}(y, z), \Phi(x, y, z)$$

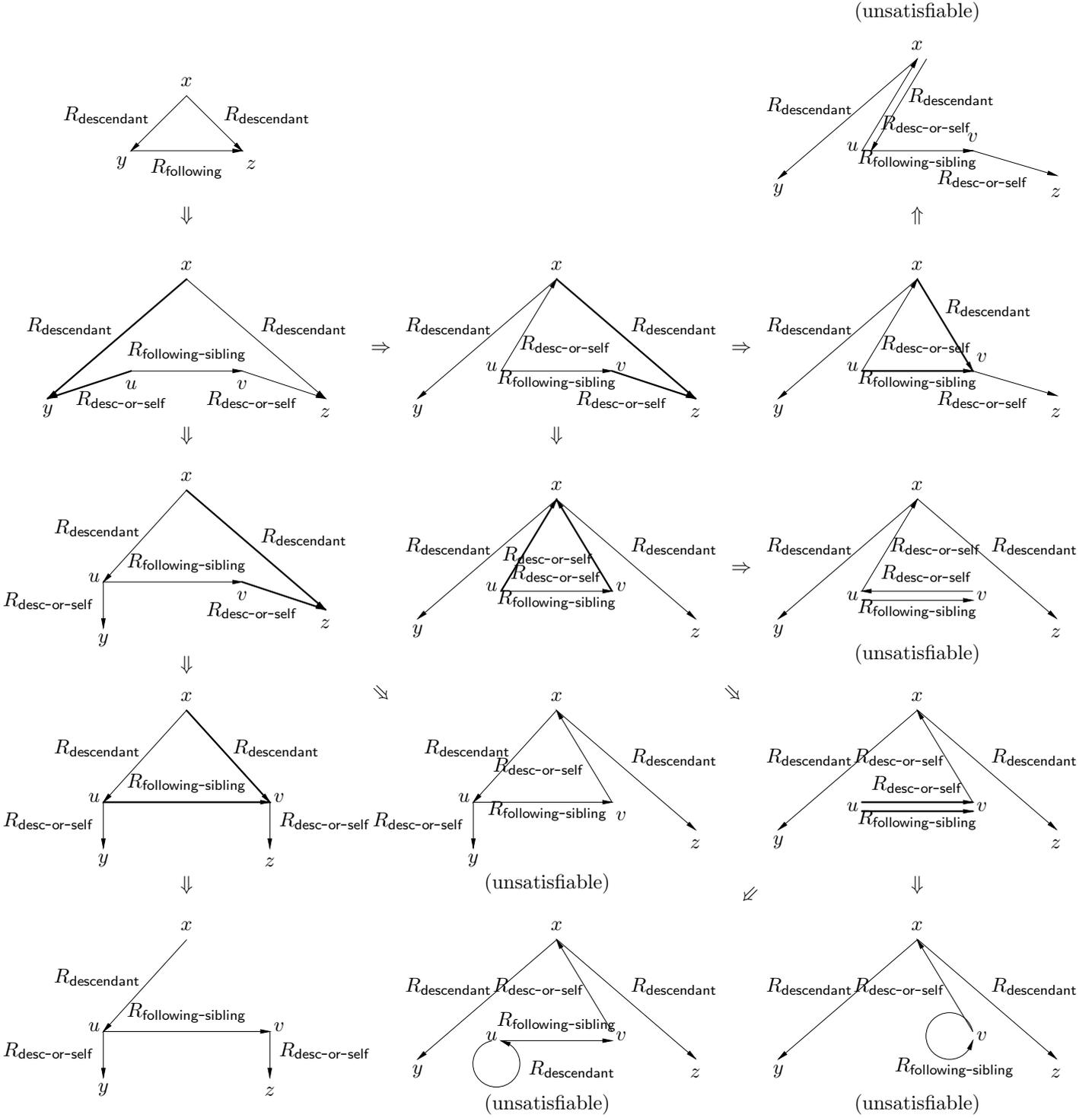


Fig. 2. Translation of a conjunctive query into an APQ.

where Φ stands for a conjunction of further atoms, such as unary label atoms, that we do not rewrite. We first rewrite the $R_{\text{following}}$ atom using $R_{\text{desc-or-self}}$ and $R_{\text{following-sibling}}$. Then we iteratively simplify cycles using facts such as that for Q to be true on a tree, given atoms $R_{\text{descendant}}(x, z)$ and $R_{\text{descendant}}(y, z)$, x must either map to the same node as y , to an ancestor of that node, or y must map to an ancestor of the node to which x maps. But then we can replace these two atoms either by $R_{\text{descendant}}(x, y)$, $R_{\text{descendant}}(y, z)$, by $R_{\text{descendant}}(y, x)$, $R_{\text{descendant}}(x, z)$, or by

$R_{\text{descendant}}(x, z)$ and replace all occurrences of variable y in the query by x .

Whenever there are several choices, we compute one copy of the query for each choice, make the appropriate replacement and then continue rewriting each conjunctive query that still has a cycle. Given an appropriate set of rewrite rules (cf. [GKS04]), we are guaranteed to either obtain a cycle that asserts unsatisfiability (for instance, a node cannot be its own ancestor), or obtain an acyclic query.

In this example, all conjunctive queries that we obtain are unsatisfiable, except for one, shown at the bottom left corner of Figure 2. Thus, for Q there exists an equivalent acyclic *conjunctive* query. \square

The translations into FO^2 from PNavXPath and from tree pattern queries are linear, but every other translation in the above theorem is exponential in the worst case; from \exists^+FO to PNavXPath and from \exists^+FO to tree patterns, this is shown in [GKS04]. For the translation from PNavXPath to tree patterns, note that PNavXPath can encode a Conjunctive Normal Form of a propositional formula (e.g. proposition p_i encoded by $[R_{\text{child}}/\text{lab}() = A_i]$). A set of tree patterns would correspond to a Disjunctive Normal Form representation of the same formula. Since it is known that there is an exponential blow-up in going from CNF to DNF, the exponential blow-up of this translation follows.

A similar argument gives:

THEOREM 3.9. *The following have equal expressiveness (up to full equivalence)*

- Boolean PNavXPath queries,
- \exists^+FO formulas $\phi(c)$ in the signature σ_{transnav} ,
- \exists^+FO formulas $\phi(c)$ in the signature σ_{transnav} with at most two variables, and
- finite sets of Boolean tree patterns.

It is easy to show that $\exists^+FO[\sigma_{\text{transnav}}]$ is closed under intersection and union, but not complement. From this and the theorem above, one has:

COROLLARY 3.10. *Boolean PNavXPath queries are closed under intersection and union, but not under complementation.*

Another consequence of the above is:

COROLLARY 3.11 [OMFB02]. *For every PNavXPath query p , there is a query p' that contains none of the axes preceding-sibling, previous-sibling, and is equivalent to p . In addition there is a query p' containing none of the “backward axes” (parent, ancestor, ancestor-or-self, preceding-sibling, previous-sibling) such that $p \equiv_r p'$.*

To see this, consider the translation of a tree pattern into PNavXPath. This translation can be done in such a way as to never introduce preceding-sibling or previous-sibling. The upward axes parent and ancestor are introduced only when the context node in the pattern is not the root. But under root equivalence, a tree pattern can always be taken to have the context node to the root (since otherwise the pattern is root equivalent to true).

[OMFB02] gives a rewrite system that removes the backward axes (parent, ancestor, ancestor-or-self, preceding-sibling), assuming root equivalence.

It is known that upward axes and backward axes cannot be removed in the presence of negation or data values: for negation, one can consider the query $p = \text{descendant}[\text{lab}() = B \wedge \neg\text{ancestor}[\text{lab}() = A]]$. One can show by an analysis of NavXPath queries without upward axes that this cannot be expressed without the use of ancestor.

Marx [Mar04a] proposes two extensions of NavXPath to capture FO^3 , and thus be first-order complete. One is by adding a path complementation feature to NavXPath and the other is by introducing conditional axes in the spirit of the *until* operator of

CTL. These results can be seen as extensions of Kamp’s Theorem [Kam68], which states that propositional temporal logic (with “until”) captures first-order logic over infinite words, to the setting of unranked trees.

3.3 Expressiveness of FOXPath

Much less is known about the expressiveness of FOXPath and AggXPath than for NavXPath. It is easy to see that FOXPath expressions can be translated into first-order logic over the signature

$$\sigma_{val}^+ = \sigma_{nav} \cup \{\text{RelOp}_{A_i, A_j} \mid i, j \in \{1, \dots, n\}, \text{RelOp} \in \{=, \neq, <, \leq, >, \geq\}\} \\ \cup \{R_{\text{descendant}}, R_{\text{following-sibling}}\},$$

where $\text{RelOp}_{A_i, A_j}(x, y)$ holds of nodes x and y iff $x.A_i \text{ RelOp } y.A_j$. An important observation is the following, analogous to one direction of Theorem 3.2:

PROPOSITION 3.12. *Every FOXPath expression p can be translated (in linear time) to a fully equivalent formula ϕ_p over vocabulary σ_{val}^+ such that ϕ_p uses at most three variables. In case p is a Boolean expression, p will have one free variable, and in case p is a NodeSet expression it will have two free variables.*

Proof. The translation is inductive; the only new case over NavXPath is the case of a qualifier $F = E \text{ RelOp } E'$. Letting $\phi_E(x, y), \phi_{E'}(x, y)$ be the translations formed inductively from E, E' respectively. Then we can set

$$\phi_F = \exists y \exists y' \phi_E(x, y) \wedge \phi_{E'}(x, y') \wedge \text{RelOp}(y, y'),$$

and note that ϕ_F has at most 3 variables. □

However, it is clear that the converse does not hold: there are first-order logic formulas using only three variables that have no equivalent in FOXPath. This is because FOXPath gives no added expressiveness on the navigational structure of a document. Formally, say that a Boolean query Q over XML documents is *navigational* if Q cannot distinguish two documents that are isomorphic as unranked ordered trees (that is, the two documents have isomorphic interpretations for σ_{nav}). Then we have

PROPOSITION 3.13 [BK05]. *Any navigational Boolean query expressible in FOXPath is expressible in NavXPath, and hence is expressible in FO². In particular (by [EVW02]), there are FO[$\sigma_{nav}, R_{\text{descendant}}$] queries not expressible in FOXPath.*

In the case of AggXPath, in contrast, it is known that all navigational first-order queries are expressible:

PROPOSITION 3.14 [BK05]. *Any FO[$\sigma_{transnav}$] query is expressible in AggXPath.*

In particular, the axis immediately-following is expressible in AggXPath.

Proof Sketch. We use Proposition 6 of [Mar04a], which states that it is sufficient to show closure under the following variant of the modal until operators. For an axis $\alpha \in \{\text{child}, \text{parent}, \text{next-sibling}, \text{previous-sibling}\}$, we write α^+ for the corresponding transitive axis ($\text{child}^+ = \text{descendant}$, etc.) and α^* for the union of α^+ with the self axis ($\text{child}^+ = \text{descendant-or-self}$, etc.). For axis $\alpha \in \{\text{child}, \text{parent}, \text{next-sibling}, \text{previous-sibling}\}$ and queries $Q_1(x), Q_2(x)$, the query $\text{Until}_\alpha(Q_2, Q_1)(x)$ (“property Q_1 until property Q_2 ”) holds at a node n iff there is n' such that $R_{\alpha^+}(n, n')$ holds, $Q_2(n')$ holds, and for all n'' such that $R_{\alpha^+}(n, n'')$ and $R_{\alpha^+}(n'', n')$ we have $Q_1(n'')$. But if E_1 and E_2 are AggXPath expressions returning Booleans, then $\text{Until}_\alpha(E_2, E_1)$ can be expressed as $\alpha^+[E_2] \wedge \neg(\text{count}(\alpha^+[\neg E_1]/\alpha^+[E_2]) = \text{count}(\alpha^+[E_2]))$. □

3.4 Further Bibliographic Remarks

In this section, we have discussed exact characterizations of `NavXPath` and its sub-languages via logic, tree patterns, and automata. There are other formalisms in which `NavXPath` can be embedded, as a strict subset.

[NS02] deals with *query automata*, an automata model that defines *NodeSet* queries. Query automata have the expressiveness of Monadic Second Order Logic, hence they are strictly more powerful than `NavXPath`. [FGK03; Koc03] deal with a variant of non-deterministic tree automata that can define unary rather than Boolean queries. [CNT04] define queries on unranked trees via automata that work on binary encodings. As with query automata, both these formalisms strictly subsume `NavXPath` in expressiveness. One starting point in looking for an automata characterization of XPath is [STV01], which gives a characterization of two-variable logic over strings in terms of partially-ordered two-way deterministic automata. We do not know of a similar characterization for two-variable logic on trees.

As mentioned in the introduction, there is a natural connection between navigational XPath and modal logics, which was first observed in [MS02] and [GK02] and subsequently revisited in several works (e.g. [Mar04b; Mar04a; AFMDR04]). The relationship between `PNavXPath` queries and acyclic first-order queries is also explored in [GKS04].

A natural question is what should be added to `NavXPath` to capture all of first-order logic. It is known that first-order logic with 3 variables captures *FO* (established in [Mar04a] for ordered unranked trees).

4. COMPLEXITY AND EFFICIENT EVALUATION

This section studies the complexity of XPath queries. XPath is a variable-free query language in which many queries – in particular, all `NavXPath` queries – have a natural tree shape when converted into first-order logic. At the same time the navigational structure of XML documents is tree-shaped. We first look at some of the classical results about tree-like queries and queries on tree-like structures. Then we explore the connections between the powerful notion of hypertree-width and XPath and show the new result that conjunctive `FOXPath` queries have hypertree-width 2. After that, we generalize from XPath evaluation based on hypertree decompositions and illustrate the dynamic programming technique that has yielded a polynomial time algorithm for full XPath 1.0. Then we survey the parallel complexity of XPath and give a new simplified proof that XPath is hard for polynomial time. Finally, we study XPath processing on data streams and give an overview over further work on efficient XPath processing.

4.1 Complexity Background

Throughout this section, we will consider logics and query languages as problem classes and will simply identify the languages with their evaluation problems. Two kinds of complexity of query evaluation will be considered, *data complexity* (where queries are assumed to be fixed and data variable) and *combined complexity* (where both data and query are considered variable) [Var82].

We briefly discuss the complexity classes and some of their characterizations used throughout the remainder of this survey. For more thorough surveys of complexity classes and the related theory see [Joh90; Pap94; GHR95].

By `P`TIME, `EXP`TIME, `NEXP`TIME, `LOG`SPACE, `NLOG`SPACE, and `PSPACE` we denote the well-known complexity classes of problems solvable on Turing machines in deterministic polynomial time, deterministic exponential time, nondeterministic exponential time, deterministic logarithmic space, nondeterministic logarithmic space, and (deterministic) polynomial space, respectively. By `NP`, we denote the decision problems solvable in nondeterministic polynomial time and `co-NP` denotes

the class of their complements.

It is a widely-held conjecture that problems complete for PTIME are inherently sequential and cannot profit from parallel computation (cf. e.g. [GHR95]). Instead, a problem is called *highly parallelizable* if it can be solved within the complexity class NC of all problems solvable in polylogarithmic time on a polynomial number of processors working in parallel [GHR95].

A simple model of parallel computation is that of Boolean circuits. By a monotone circuit, we denote a circuit in which only the input gates may possibly be negated. All other gates are either \wedge -gates or \vee -gates (but no \neg -gates). A family of circuits is a sequence $\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_2, \dots$, where the n -th circuit \mathcal{G}_n has n inputs. Such a family is called *LOGSPACE-uniform* if there exists a LOGSPACE-bounded deterministic Turing machine which, on the input of n bits 1 (the string 1^n), outputs the circuit \mathcal{G}_n . A family of circuits has *bounded fan-in* if all of the gates in these circuits have fan-in bounded by some constant. On the other hand, a family of monotone circuits is called *semi-unbounded* if all \wedge -gates are of bounded fan-in (without loss of generality, we may restrict the fan-in to two) but the \vee -gates may have unbounded fan-in.

NC^i denotes the class of languages recognizable using LOGSPACE-uniform Boolean circuit families of polynomial size and depth $O(\log^i n)$ (in terms of the size n of the input). SAC^1 is the class of languages recognizable by LOGSPACE-uniform families of semi-unbounded circuits of depth $O(\log n)$ (SAC^1 circuits).

A nondeterministic auxiliary pushdown automaton (NAuxPDA) is a nondeterministic Turing machine with a distinguished input tape, a worktape, and a stack (of which strictly only the topmost element can be accessed at any time).

LOGCFL is usually defined as the complexity class consisting of all problems LOGSPACE-reducible to a context-free language. There are two important alternative characterizations of LOGCFL that we are going to use. They are recalled in Proposition 4.1 and 4.2, respectively.

PROPOSITION 4.1 [VEN91]. $\text{LOGCFL} = \text{SAC}^1$. SAC^1 *Circuit Value* is LOGCFL-complete.

PROPOSITION 4.2 [SUD77]. LOGCFL is the class of all decision problems solvable by a NAuxPDA with a logarithmic space-bounded worktape in polynomial time.

We have $\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{LOGCFL} \subseteq \text{NC}^2 \subseteq \text{NC} \subseteq \text{PTIME} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME}$. All inclusions \subseteq are suspected to be strict, and all these complexity classes are closed under LOGSPACE-reductions.

Unless stated otherwise, we assume the input represented as a σ_{dom} -structure encoded in the usual way.

4.2 Tree-like Data and Tree-like Queries

As a warm-up, we use the well-studied graph-theoretical notion of *tree-width* to derive a few results about the complexity of XPath that follow immediately from the literature.

Let $G = (V^G, E^G)$ be a graph. A *tree decomposition* of G is a pair (T, χ) such that T is a rooted tree with nodes V^T , χ is a function $\chi : V^T \rightarrow 2^{V^G}$ that maps each node of tree T to a subset of V^G , for each edge $(u, v) \in E^G$ there exists a node $w \in V^T$ such that $u, v \in \chi(w)$, and for each node $u \in V^G$, the set $\{v \in V^T \mid u \in \chi(v)\}$ induces a connected subtree of T . The *width* of tree decomposition (T, χ) is defined as $(\max\{|\chi(v)| \mid v \in V^T\}) - 1$. The *tree-width* of a graph G is the smallest width over all tree decompositions of G . Intuitively, graphs of low tree-width are very tree-like. As a special case, the connected graphs of tree-width one are precisely the trees. An example of a graph and a tree decomposition (of width 2) for it is given in Figures 3 (a) and (b), respectively.

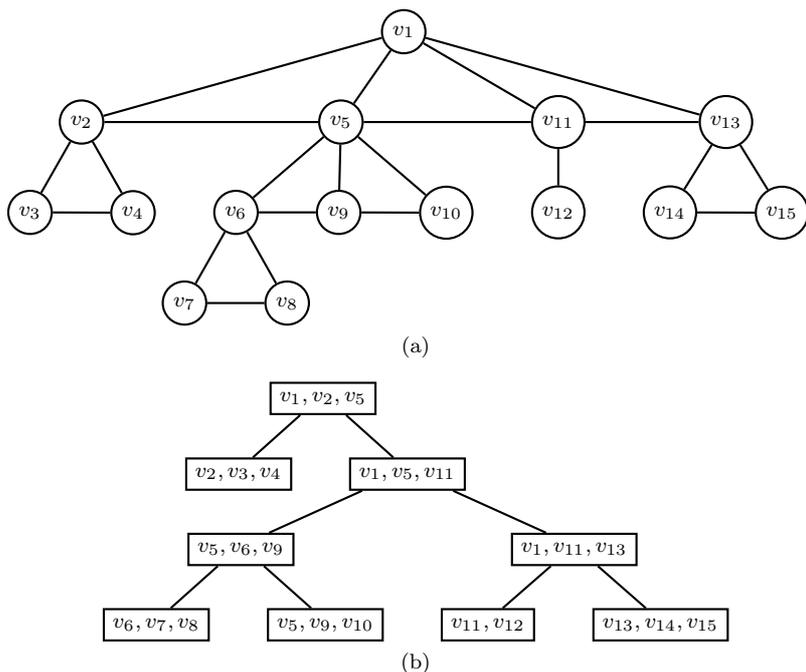


Fig. 3. A σ_{nav} -tree is a graph of tree-width two.

We say that a structure consisting only of unary and binary relations has tree-width k if the union of (the symmetric closure of) its binary relations has tree-width k . We do not give a formal definition of the general case of *queries of bounded tree-width* here; however, for conjunctive queries Q over a vocabulary of at most binary relation symbols, the *tree-width of Q* is defined as the tree-width of the graph $G = (V, E)$ where V consists of the variables of Q and $(x, y), (y, x) \in E$ if there is an atom $a(x, y)$ in Q .

§1: *Tree-like data lead to linear-time data complexity.* The Boolean MSO queries on trees labeled with a finite alphabet (e.g. σ_{nav} -trees) define precisely the *regular tree languages*, which correspond to the *deterministic bottom-up tree automata* [TW68; Don70; BKMW01]. Each Boolean MSO query can be mapped to such an automaton, whose acceptance of a given input tree can be checked in linear time in the size of the tree (traversing it once bottom-up). Thus, Boolean MSO queries on trees have linear-time data complexity. A slightly more general version of this fact for bounded tree-width structures is known as Courcelle’s Theorem [Cou90], which can be further generalized to

THEOREM 4.3 [FFG02]. *Let \mathbf{C} be a class of structures of bounded tree-width. For a fixed MSO formula ϕ , there is an algorithm that evaluates ϕ on each structure $\mathcal{A} \in \mathbf{C}$ in time $O(|\mathcal{A}| + |\phi(\mathcal{A})|)$.*

That is, this algorithm runs in time linear in the size of the input and the output, and in particular in linear time *in the size of the input* on MSO formulas with at most one free variable.

It can be verified that unranked ordered trees represented by σ_{nav} -structures, that is, the union of their binary relations R_{child} and $R_{\text{next-sibling}}$, have tree-width two²

²Note, however, that in the context of MSO, it is more wide-spread [Nev02; GK04] to use a signature σ'_{nav} obtained from σ_{nav} by replacing R_{child} by a relation FirstChild such that $\text{FirstChild}(x, y)$ iff y is the *leftmost* child of x . Then, MSO on σ_{nav} and σ'_{nav} are equivalent and all σ'_{nav} -structures have tree-width 1.

(see Figure 3, where each node v is labeled with $\chi(v)$). Transitive axis relations such as $R_{\text{descendant}}$ or $R_{\text{following-sibling}}$ (cf. Section 2.1) do not have bounded tree-width in general, but it is not difficult to map NavXPath queries with transitive axes to MSO over signature σ_{nav} [GK02]. The construction is similar to the one of Theorem 3.2 mapping NavXPath to FO^2 , defining $R^*(x, y)$, where R^* is the reflexive and transitive closure of relation R , in MSO as $\forall S (S(x) \wedge \forall u \forall v S(u) \wedge R(u, v) \rightarrow S(v)) \rightarrow S(y)$. From this we can conclude the following bound.

COROLLARY 4.4. *Unary NavXPath is in linear time w.r.t. data complexity.*

§2: *Tree-like data do not yield low combined complexity.* The usual technique for proving linear-time data complexity of MSO is by reduction to automata. For unary MSO formulas, somewhat sophisticated automata with a capability for selecting nodes are required. It has been observed that such automata with the power of unary MSO can be designed to traverse the data tree only twice [Nv02; FGK03]. Reductions from MSO to automata do not yield good upper bounds on the combined complexity of NavXPath, however. Indeed, they are necessarily nonelementary [Mey75; Rei02] (i.e., their cost cannot be bounded by any tower of exponentials $2^{2^{\dots^{2^n}}}$ of fixed height). For NavXPath, a doubly exponential translation to *selecting tree automata* [FGK03] is implicit in [Koc03].

§3: *Tree-like queries yield polynomial-time combined complexity.* While MSO over trees is known to be PSPACE-complete with respect to combined complexity, FO^k (even over arbitrary relational structures) is known to be in time $O(n^k * |Q|)$:³

PROPOSITION 4.5 [KV00]. *Conjunctive FO^{k+1} queries have tree-width $\leq k$.*

THEOREM 4.6 [CR97]. *Given a Boolean conjunctive query Q of tree-width k and a database \mathcal{A} with domain size n , Q can be evaluated on the database in time $O((n^{k+1} + |\mathcal{A}|) * |Q|)$.*

Both results generalize from conjunctive to FO queries [FFG02].

Since NavXPath queries can be translated efficiently, in linear time, into equivalent FO^2 queries (Theorem 3.2) and FOXPath queries can be translated in linear time into FO^3 (Proposition 3.12),

COROLLARY 4.7. *Boolean NavXPath and FOXPath are in time $O(n^2 \cdot |Q|)$ and $O(n^3 \cdot |Q|)$, respectively.*

As we will see next, these combined complexity bounds can be improved upon.

4.3 Hypertree-width and Conjunctive XPath

Let Q be a conjunctive query over a relational database, and let $\text{vars}(Q)$, $\text{free}(Q)$, and $\text{atoms}(Q)$ denote the set of variables, free/head variables, and atoms occurring in Q , respectively.

A (complete) *hypertree decomposition* of Q is a triple (T, χ, λ) such that T is a rooted tree with nodes $V(T)$ and root node r , $\chi : V(T) \rightarrow 2^{\text{vars}(Q)}$ maps each node of tree T to a set of variables from Q , $\lambda : V(T) \rightarrow 2^{\text{atoms}(Q)}$ maps each node of T to a set of body atoms of Q ,

- (1) $\text{free}(Q) \subseteq \chi(r)$,
- (2) for each atom $A \in \text{atoms}(Q)$, there exists a node $v \in V(T)$ such that $A \in \lambda(v)$ and $\text{vars}(A) \subseteq \chi(v)$,
- (3) for each variable $x \in \text{vars}(Q)$, the set $\{v \in V(T) \mid x \in \chi(v)\}$ induces a connected subtree of T , and

³This can be shown directly without tree-width as well, however.

(4) for each node $v \in V(T)$, $\chi(v) \subseteq \text{vars}(\lambda(v))$ and

$$\text{vars}(\lambda(v)) \cap \bigcup \{\chi(v') \mid v = v' \text{ or } v' \text{ is a descendant of } v \text{ in } T\} \subseteq \chi(v).$$

The *width* of a hypertree decomposition (T, χ, λ) is the maximum number of atoms occurring in any single node of T , i.e. $\max\{|\lambda(v)| \mid v \in V(T)\}$. The *hypertree-width* of a conjunctive query Q is the smallest width over all hypertree decompositions of Q . The conjunctive queries of hypertree-width 1 coincide with the so-called *acyclic* conjunctive queries (cf. e.g. [AHV95]). As shown in [Yan81], the acyclic conjunctive queries can be evaluated in time $O(n \cdot |Q|)$. Yannakakis' result was generalized to hypertree-width k , for arbitrary k :

THEOREM 4.8 [GLS02]. *Let Q be a conjunctive query and \mathcal{H} a hypertree decomposition of width k of Q . Then Q can be evaluated on a database \mathcal{A} in time $O((|\mathcal{H}| + |\mathcal{A}|)^k)$.*

Let σ'_{dom} be the signature obtained from σ_{dom} by replacing each attribute function $@A$ by its graph (i.e., the binary relation $\{(n, @A(n)) \mid n \in \text{Node}\}$) and adding the relations $R_{\text{descendant}}$ and $R_{\text{following-sibling}}$.

A considerable fragment of **FOXPath** can be modeled by conjunctive queries over a structure of relational signature σ'_{dom} . We say that a **FOXPath** query (resp., **NavXPath** query) is *conjunctive* (and *connected*) if it does not use disjunction, negation, inequalities (i.e., expressions $p \text{ RelOp } p'$ with $\text{RelOp} \neq "="$), or the root slash $/$. The notions of hypertree decomposition and hypertree-width can be readily applied to conjunctive **FOXPath** (and thus **NavXPath**) queries. A conjunctive **FOXPath** query maps to a conjunctive query over σ'_{dom} , and we can speak of its hypertreewidth using this mapping.

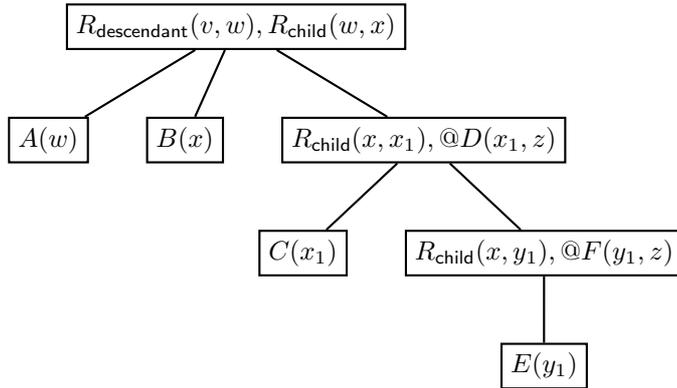
EXAMPLE 4.9. The conjunctive **FOXPath** query

$$\text{descendant}::A/\text{child}::B[\text{child}::C/@D = \text{child}::E/@F]$$

can be phrased as a conjunctive query over signature σ'_{dom}

$$Q(v, x) \leftarrow R_{\text{descendant}}(v, w), A(w), R_{\text{child}}(w, x), B(x), R_{\text{child}}(x, x_1), C(x_1), @D(x_1, z), \\ R_{\text{child}}(x, y_1), E(y_1), @F(y_1, z).$$

Consider the following hypertree decomposition, \mathcal{H} , of Q , where the nodes v have been labeled with $\lambda(v)$ and $\chi(v) = \text{vars}(\lambda(v))$:



Note that \mathcal{H} is of width 2. There exists obviously no hypertree decomposition of width 1: the atoms $\{R_{\text{child}}(x, x_1), @D(x_1, z), R_{\text{child}}(x, y_1), @F(y_1, z)\}$ of Q induce a cycle. Thus Q is of hypertree-width 2. \square

By Propositions 4.5 and 3.12, conjunctive **FOXPath** queries have tree-width ≤ 2 . It is known that conjunctive queries of tree-width k have hypertree-width \leq

$k + 1$ [GLS02], so we can obtain the $O(n^3)$ data complexity bound observed in Corollary 4.7 also from Theorem 4.8. However, fortunately,

THEOREM 4.10. *The conjunctive FOXPath queries have hypertree-width ≤ 2 .*

Proof. Given a hypertree decomposition of a conjunctive query of width k , there is an efficient algorithm for turning it into a relational algebra query plan that can be naturally evaluated in time $O(n^k)$ [GLS99].

In this proof, however, we will go the other way round and will first compute a first-order query (using just \exists and \wedge) over σ'_{dom} for a given conjunctive FOXPath query and will then show that it yields a hypertree decomposition of width ≤ 2 . From this first-order formula an equivalent relational algebra plan can be obtained immediately by rewriting \wedge into join \bowtie and \exists into projection π .

Without loss of generality, we will assume that our query is a path expression p . The proof works analogously for qualifiers. We translate p into a first-order formula $FO(p)_2$ as follows:

$$\begin{aligned}
FO(axis)_2(x, y) &:= R_{axis}(x, y) \\
FO(step[q])_2(x, y) &:= FO(step)_2(x, y) \wedge FO(q)_1(y) \\
FO(p/step)_2(x, z) &:= \exists y \llbracket p \rrbracket_2(x, y) \wedge FO(step)_2(y, z) \\
FO(\text{lab}() = L)_1(x) &:= L(x) \\
FO(p)_1(x) &:= \exists y FO(p)_2(x, y) \\
FO(q \wedge q')_1(x) &:= FO(q)_1(x) \wedge FO(q')_1(x) \\
FO(p/@A = p'/@B)_1(x) &:= \exists z (\exists y_1 FO(p)_2(x, y_1) \wedge @A(y_1, z)) \wedge \\
&\quad (\exists y_2 FO(p')_2(x, y_2) \wedge @B(y_2, z))
\end{aligned}$$

Without loss of generality, we will assume that there are no two distinct occurrences of existential quantification over the same variable in $FO(p)_2$; thus, any two occurrences of the same variable name in formula $FO(p)_2$ indeed refer to the same variable.

It is easy to verify – $FO(\cdot)_2$ is only a minor variation of $\llbracket \cdot \rrbracket_{NodeSet}$ – that $FO(p)_2$ defines a binary relation $\{(n, n') \mid n' \in \llbracket p \rrbracket_{NodeSet}(n)\}$.

We now construct a hypertree decomposition of $FO(p)_2$. Consider the parse tree T of formula $FO(p)_2$. This parse tree has relation atoms as its leaves and $\exists x$ - and \wedge -labels on its internal nodes. Each node of the tree corresponds to a subformula ϕ of $FO(p)_2$. We will identify each tree node with the subformula ϕ it denotes.

We define a function λ that maps each node ϕ of T to a set of leaf nodes (and thus relational atoms). We do this inductively, bottom-up:

- (i) for each leaf node ϕ , $\lambda(\phi) := \{\phi\}$;
- (ii) for each node ϕ of the form $\psi_1(x) \wedge \psi_2(x)$, $\psi_1(x, y) \wedge \psi_2(y)$, or $\psi_1(x, y) \wedge \psi_2(x, y)$, let $\lambda(\phi) := \lambda(\psi_1)$;
- (iii) for each node $\phi = \psi_1(x, y) \wedge \psi_2(y, z)$, let $\lambda(\phi) := \{\psi'\} \cup \lambda(\psi_2)$, where ψ' is any atom over x from $\lambda(\psi_1)$; finally,
- (iv) for each node $\phi = \exists x \psi$, $\lambda(\phi) := \lambda(\psi)$.

Note, in particular, that each free variable of ϕ occurs in at least one atom of $\lambda(\phi)$. Now let function χ map each node ϕ of T to $vars(\lambda(\phi))$.

To verify that (T, χ, λ) is indeed a hypertree decomposition of p , we have to check points (1) to (4) of the definition. (1) and (4) are due to the definition of χ as $\phi \mapsto vars(\lambda(\phi))$. (2) is immediate from (i). The connectedness condition (3) follows from the fact that in a first-order query without any two distinct occurrences of existential quantification over the same variable, the nodes of parse tree T that have x as a free variable plus the node $\exists x \psi$ if x is not free in the query induce a connected subtree of T .

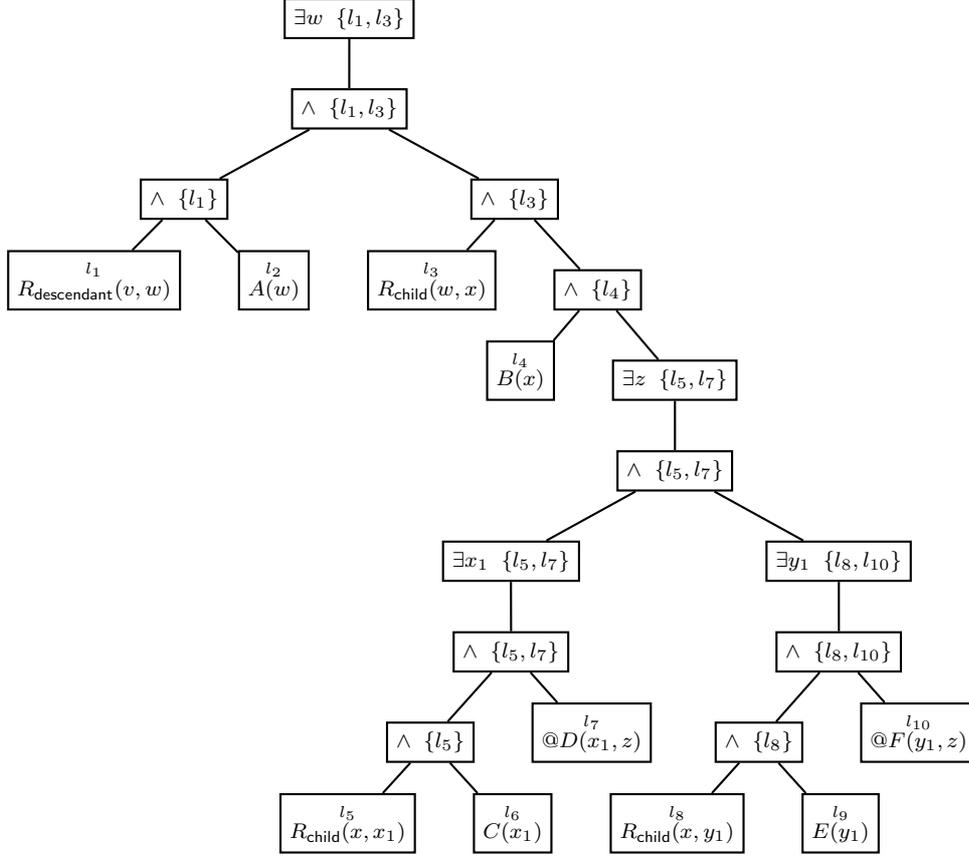


Fig. 4. Hypertree decomposition of the query of Example 4.9 as constructed in the proof of Theorem 4.10.

Let us now consider the sizes $|\lambda(\phi)|$ for all nodes ϕ of T . The most interesting case is $\phi = \psi_1(x, y) \wedge \psi_2(y, z)$. Observe that in this case ψ_2 is either a step expression or a leaf, and thus $|\lambda(\psi_2)| = 1$, so $|\lambda(\phi)| = 2$. It can be shown by a straightforward induction that for all nodes ϕ , $|\lambda(\phi)| \leq 2$, so our query has hypertree-width ≤ 2 . \square

EXAMPLE 4.11. For the query of Example 4.9, the proof of Theorem 4.10 constructs the first-order formula given by the parse tree of Figure 4. Here, the leaf nodes have been labeled l_1, l_2, l_3, \dots from left to right and the interior nodes ϕ of the parse tree of the formula have been annotated with $\lambda(\phi)$. Again, $\chi(\phi) = \text{vars}(\lambda(\phi))$. This identifies the hypertree decomposition constructed in the proof. \square

The transformation of the previous proof can be implemented so as to compute both first-order query and hypertree decomposition in linear time. By the latter observation and Theorem 4.8, conjunctive FOXPath can thus be evaluated in time $O((|Q| + |\mathcal{A}|)^2)$. We give a direct proof of the following (close but incomparable) bound.

PROPOSITION 4.12. *Conjunctive FOXPath on σ'_{dom} -structures \mathcal{A} is in time $O(|Q| \cdot |\mathcal{A}|^2)$.*

Proof. Let us now consider relational algebra queries $ALG(p)$ and $ALG(q)$ corresponding to the first-order (calculus) queries $FO(p)_2$ and $FO(q)_1$ of the previous proof. The translation is standard [AHV95] and just requires rewriting existential quantification by projection and conjunction by join.

Now observe that, as with the subformulas of ϕ in $FO(p)_2$, each subexpression

of $ALG(p)$ defines a relation that is a subset of the product of at most two base relations $\lambda(\phi)$, and is thus of size at most $O(|\mathcal{A}|^2)$.

Query evaluation requires no more than $|Q|$ relational algebra operations (projections or joins). The projections $\pi_{\bar{A}}R$ are obviously operations that run in time linear in $|R|$. Joins *guarded* by one of the input relations (corresponding to formulae $\psi_1(x, y) \wedge \psi_2(x, y)$, $\psi_1(x, y) \wedge \psi_2(y)$, and $\psi_1(y) \wedge \psi_2(y)$) can be evaluated in time linear in the sum of the sizes of the two relations joined by first building a bitfield for testing whether tuples are true in ψ_2 and then using it to filter the tuples of ψ_1 .

The most interesting case is a join corresponding to formula $\psi_1(x, y) \wedge \psi_2(y, z)$. Let $\llbracket \phi \rrbracket$ be the relation defined by first-order formula ϕ . We first compute the relations $R_1^y = \{x \mid \psi_1(x, y)\}$, for each y , in total time $O(\llbracket \psi_1 \rrbracket + \llbracket \psi_2 \rrbracket)$. Then we compute our join as the union of the sets $\{(x, y, z) \mid R_1^y(x)\}$, for each tuple $\psi_2(y, z)$. As mentioned in the previous proof, ψ_2 always defines a subset of an input relation, so this union can be formed in time $O(|A| \cdot \llbracket \psi_2 \rrbracket) = O(|\mathcal{A}|^2)$. \square

Conjunctive NavXPath queries are acyclic (see [GKP02]) and can therefore be evaluated using Yannakakis' algorithm (or by precisely the techniques from the previous two proofs) both in linear time in the data *and* efficiently in the size of the query.

PROPOSITION 4.13. *Unary conjunctive NavXPath queries can be evaluated in time $O(|\mathcal{A}| \cdot |Q|)$ on $(\sigma_{nav}, R_{\text{descendant}}, R_{\text{following-sibling}})$ -structures \mathcal{A} .*

4.4 Beyond Conjunctive Queries

The conjunctive query processing techniques based on hypertree decompositions of the previous section leave three features of FOXPath unaddressed:

- (1) Conjunctive FOXPath excludes disjunction, union, negation, inequalities, and disconnected queries (via the root / in conditions).
- (2) We assumed that the data tree is given by σ_{val}^+ -structures, which include binary relations for transitive axes such as **descendant**. If we assume transitive axis relations present in the structure \mathcal{A} representing a tree with domain A and therefore $|\mathcal{A}| = O(|A|^2)$, our upper time bound of $O(|\mathcal{A}|^2 \cdot |Q|)$ from Proposition 4.12 deteriorates to time $O(|A|^4 \cdot |Q|)$.
- (3) Finally, we did not deal with inequalities $\text{RelOp} \in \{\neq, <, \leq\}$ in expressions $e \text{RelOp} e'$.

THEOREM 4.14. *A FOXPath query Q can be evaluated on σ_{dom} -structures with domain A in time $O(|A|^2 * |Q|)$.*

Proof.

- (1) Now we complete the mapping ALG of the previous proof by the operations of FOXPath missing from conjunctive FOXPath:

$$\begin{aligned} -ALG(p \mid p') &:= ALG(p) \cup ALG(p') \\ -ALG(q \vee q') &:= ALG(q) \cup ALG(q') \\ -ALG(\neg q) &:= A - ALG(q) \end{aligned}$$

- (2) Next we would like to eliminate transitive axis relations such as **descendant** from the signature.

[GKP02] gives algorithms for computing, given a set S of tree nodes and any XPath axis α , the set of nodes

$$\alpha(S) = \{y \mid x \in S \wedge R_\alpha(x, y)\}$$

in time $O(|Node|)$. Consider the unary operations

$$\bowtie_{\alpha[q]}: R \mapsto \{(x, z) \mid \exists y R(x, y) \wedge R_\alpha(y, z) \wedge \llbracket q \rrbracket_{\text{Boolean}}(z)\},$$

which can be evaluated in quadratic time by first partitioning R into sets $S_x = \{y \mid R(x, y)\}$, for each x , and then computing the union over x of the sets $\{(x, y) \mid y \in \alpha(S_x) \wedge \llbracket q \rrbracket_{Boolean}(y)\}$.

Now we can evaluate $\llbracket p/\alpha[q_1] \dots [q_n] \rrbracket$ as $\alpha[q_1 \wedge \dots \wedge q_n](\llbracket p \rrbracket)$ in quadratic time, for any axis α , even if our structure is just of signature σ_{dom} .

- (3) Let α^{-1} denote the inverse of axis α (i.e., $R_{\alpha^{-1}}$ is the inverse of R_α). To compute a query plan for an inequality

$$\alpha_1[q_1]/\alpha_2[q_2]/\dots/\alpha_n[q_n]/@A \text{ RelOp } \beta_1[q'_1]/\beta_2[q'_2]/\dots/\beta_n[q'_n]/@B$$

with $\text{RelOp} \neq "="$, we first compute the binary relation $\text{RelOp}_{A,B}$ (see the definition of σ_{val}^+ in Section 3.3) in time $O(|A|^2)$. Then we compute

$$S := \bowtie_{\beta_1^{-1}} (\bowtie_{\beta_2^{-1}[q'_1]} (\bowtie_{\beta_3^{-1}[q'_2]} (\dots \bowtie_{\beta_n^{-1}[q'_{n-1}]} (\bowtie_{\text{self}[q'_n]} (\text{RelOp}_{A,B})) \dots)))$$

Finally,

$$(\bowtie_{\alpha_1^{-1}} (\bowtie_{\alpha_2^{-1}[q_1]} (\bowtie_{\alpha_3^{-1}[q_2]} (\dots \bowtie_{\alpha_n^{-1}[q_{n-1}]} (\bowtie_{\text{self}[q_n]} (S^{-1})) \dots))))^{-1}$$

is the desired relation. \square

Applying the first two parts of the previous proof to NavXPath yields

PROPOSITION 4.15 [GKP02]. *A NavXPath query Q can be evaluated on σ_{nav} -structures \mathcal{A} in time $O(|\mathcal{A}| \cdot |Q|)$ and space $O(|\mathcal{A}|)$.*

Beyond FOXPath , we are faced with queries containing possibly nested numeric expressions involving the arithmetic operations $+$ and $*$ (whose graphs are infinite) and aggregations. For that reason, it is helpful to digress from the framework used above (i.e., relations $\subseteq A^2$ or $\subseteq A$) and view every expression e of *type* t (either NodeSet , Boolean , or Int) as defining a table $\{(n, \llbracket e \rrbracket_t(n)) \mid n \in A\}$. Each node n denotes a context in which expression e evaluates to value $\llbracket e \rrbracket_t(n)$. Thus such tables were called *context-value tables* in [GKP05]. The context-value table of an expression e can be efficiently computed from the context-value table of the direct subexpressions of e . For FOXPath , the method for doing so was given in the previous proof, up to the notational subtleties that now for NodeSet -typed expressions, the value column may hold sets (nodes grouped by their context) while in the proof the relations defined were flat, and that context-value tables for Boolean -valued expressions are binary, with either “true” or “false” in the value column.

This method can be adapted to AggXPath without a runtime penalty, since on a binary relation $\llbracket p \rrbracket$ over the domain of nodes – and thus of quadratic size – the relations $\{(n, i) \mid \llbracket \text{count}(p) \rrbracket_{Int}(n) = i\}$ and $\{(n, i) \mid \llbracket \text{sum}(p/@A) \rrbracket_{Int}(n) = i\}$ can be computed in quadratic time without difficulty. For the arithmetic operation $*$ (multiplication), numbers can grow linearly with the query, thus a binary relation representing the result of a numeric relation may be of size $O(|A| \cdot |Q|)$. Thus,

PROPOSITION 4.16. *The AggXPath queries Q can be evaluated on σ_{dom} -structures with domain A in time $O(|A| \cdot (|A| + |Q|) \cdot |Q|)$ and space $O(|A| \cdot (|A| + |Q|))$.*

So far we have been moving only moderately beyond queries obtained from hypertree decompositions. However, XPath (and OrdXPath) supports position arithmetics which require more sophisticated contexts than AggXPath , where contexts are simply nodes. For OrdXPath , a single context node is not sufficient; for instance, the expression “ $\text{position}() = \text{last}()$ ” relies on the position of a node within a set and the cardinality of that set as contexts (see (P2’) in Section 2).

We extend context-value tables to be sets of tuples (n, j, k, v) , where n is a context node, j and k are integers denoting a position j in and the size k of a set of nodes, v is a value, and the contexts n, i, k identify their tuples.

Values (including strings and numbers) were shown in [GKP02] to remain small in XPath. The algorithm of [GKP02] inductively computes context-value tables $\{(n, j, k, v) \mid \llbracket e \rrbracket_{Type(e)}(n, j, k) = v\}$ for each subexpression e of a query bottom-up. Taking into context all the builtin functions of XPath, this yields the following upper bound.

THEOREM 4.17 [GKP02]. *Full XPath 1.0 is in time $O(|A|^5 \cdot |Q|^2)$.*

Improvements yielding somewhat better bounds can be found in [GKP05].

EXAMPLE 4.18. Consider the numerical expression $position() * 2 < last()$. We compute the contex-value tables of its subexpressions bottom-up as

$$\begin{aligned} CVT_{position()} &:= \{(n, j, k, j) \mid (n, j, k) \text{ a context}\} \\ CVT_{position()*2} &:= \{(n, j, k, 2 * v) \mid (n, j, k, v) \in CVT_{position()}\} \\ CVT_{last()} &:= \{(n, j, k, k) \mid (n, j, k) \text{ a context}\} \\ CVT_{position()*2 < last()} &:= \{(n, j, k, (v_1 < v_2)) \mid (n, j, k, v_1) \in CVT_{position()*2}, \\ &\quad (n, j, k, v_2) \in CVT_{last()}\} \end{aligned}$$

In summary, there is a close connection between the context-value table-based dynamic programming algorithm of [GKP02] and the hypertree-width based techniques presented before. However, beyond the difficulties dealt with in the proof of Theorem 4.14, XPath supports built-in functions (e.g. arithmetic and string functions) whose graphs are infinite and aggregations, so claiming the PTIME combined complexity of XPath an immediate consequence of hypertree decomposition techniques would be a bit far-fetched.

4.5 Parallel Complexity

Now that the combined complexity of XPath is known to be polynomial, one may ask whether XPath is also PTIME-hard, or alternatively, whether it is in the complexity class NC and thus effectively parallelizable. Apart from theoretical interest, a precise characterization of XPath evaluation in terms of parallel complexity classes may lead to a better understanding of what computational resources are necessarily required for query evaluation. For example, it is strongly conjectured that all algorithms for solving PTIME-hard problems actually require a polynomial amount of working memory. However, performing XPath query evaluation with limited memory resources is important in practice, e.g. in the context of data stream processing.

For an upper bound for conjunctive FOXPath, we can use the following result about conjunctive queries of bounded hypertree-width together with our Theorem 4.10.

THEOREM 4.19 [GLS01]. *The conjunctive queries of bounded hypertree-width over arbitrary relational structures are in LOGCFL w.r.t. combined complexity.*

COROLLARY 4.20. *Conjunctive FOXPath is in LOGCFL (combined complexity).*

In [GKPS05], LOGCFL membership is proven for a much larger fragment of XPath without negation which even supports arithmetics and aggregations. Here we give a direct proof for positive FOXPath.

PROPOSITION 4.21 [GKPS05]. *Positive FOXPath is in LOGCFL w.r.t. combined complexity.*

Proof Idea. By an encoding as a NauxPDA that runs in polynomial time using a LOGSPACE worktape. We will actually show how to use a NauxPDA to compute the set of nodes to which an XPath query evaluates, even though the complexity class LOGCFL is defined in terms of decision problems and for the above-mentioned

lower bound only a decision problem (e.g. that of checking whether a given node is selected by an XPath query) makes sense.

We will use the symbol $\&$ for creating references and $*$ to dereference them. We will associate each query with its (binary) parse tree obtained in the usual fashion, using grammar rules $p := axis :: A[q]/p \mid axis :: A[q]$ to parse paths (i.e., producing a right-deep tree for a path). An example of such a parse tree is shown in Figure 5. We identify nodes of the query tree with the expressions their subtrees represent. For a path expression p , we use $sel(v_Q)$ to denote the rightmost leaf in the subtree of the query tree corresponding to p ; thus $sel(v_Q)$ denotes the “right tip” of the path which selects nodes.

We use four log-space registers that will be kept on the worktape, sel (to iterate over the nodes of the data tree and check which are to be selected by the query), v_t (to hold a node from the data tree), r_{val} (for a pointer to a data value in the data tree, represented by an integer indicating the starting position of the data value’s representation inside the representation of the data tree), and v_Q (for a current node from the parse tree of the query) on the worktape.

The evaluation of the query proceeds by iterating over all the nodes of the data tree (using register sel), and for each node does a single depth-first left-to right traversal of its parse tree, starting with v_Q the root node of the query tree, v_t the root of the input tree, and $r_{val} = \perp$.

By default, query tree nodes v_Q with two children are processed as follows. First we put (v_Q, v_t, r_{val}) onto the stack. Then we process the first child of v_Q . On returning we take (v_Q, v_t, r_{val}) off the stack (and set the registers). Finally process the second child of v_Q .

There are a few exceptions. When $v_Q = \alpha :: A[q]/p$ and $v_t = n$, we first put n on the stack, nondeterministically guess a node n' such that $\alpha(n, n')$ and $A(n')$, set v_t to n' , and only then we process the two children as just described. Expressions $p/@A/deref()$ are handled similarly.

For $p/@A = p'/@B$, r_{val} is not put on the stack before and taken off the stack after processing the first child. When arriving at $sel(p)$, we set r_{val} to $@A(v_t)$. When arriving at $sel(p')$, we verify that $r_{val} = @B(v_t)$.

If $v_Q = q \vee q'$, we nondeterministically choose either q or q' and verify that it holds relative to the current position v_t .

At $sel(p)$, where p is the query, we check whether $v_t = sel$. If so, we output node sel as a result.

It is not difficult to verify that this nondeterministic algorithm runs on an NAuxPDA in polynomial time, using only logarithmic space on the worktape. \square

EXAMPLE 4.22. The FOXPath query $./A[./B/@C = D[E/@F = G/H]/@I]$ can be evaluated using a NAuxPDA given by the following pseudocode: (1) Guess w such that $[[./A]](v_t, w)$; $v_t := w$; (2) push v_t ; (3) guess w such that $[[./B]](v_t, w)$; $v_t := w$; (4) $r_{val} := \&v_t.@C$; (5) $v_t := \text{pop}$; (6) guess w such that $[[./D]](v_t, w)$; $v_t := w$; push r_{val} ; push v_t ; (7) push v_t ; (8) guess w such that $[[./E]](v_t, w)$; $v_t := w$; (9) $r_{val} := \&v_t.@F$; (10) $v_t := \text{pop}$; (11) guess w such that $[[./G]](v_t, w)$; $v_t := w$; (12) check that $*r_{val} = v_t.@H$; (13) $v_t := \text{pop}$; $r_{val} := \text{pop}$; (14) check that $*r_{val} = v_t.@I$; (15) accept.

Note that this program is faithful to the construction mentioned above except that we do not push or pop the v_Q register (the query has been compiled into the program).

The fact that the run of this NAuxPDA is intuitively a depth-first traversal of the parse tree of the query is illustrated in Figure 5. \square

It was shown in [GKPS05] by a reduction from the SAC¹ circuit value problem that the LOGCFL upper bound of Theorem 4.21 is tight: positive NavXPath is LOGCFL-complete w.r.t. combined complexity.

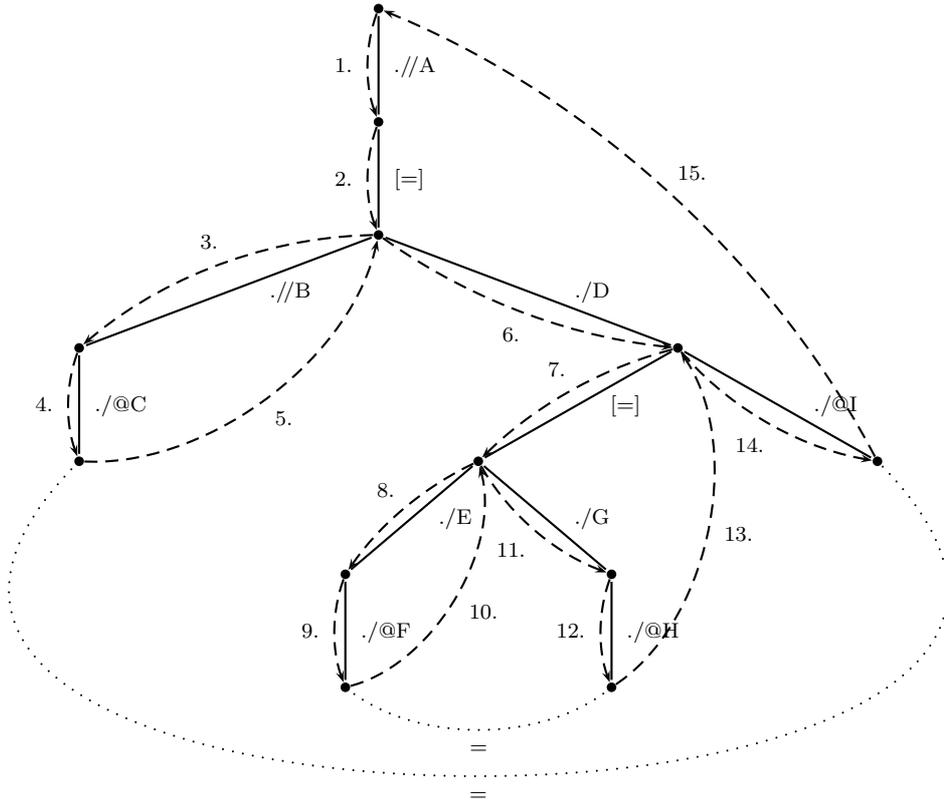


Fig. 5. N AuxPDA run for query $./@a[./@b/@c = d[e/@f = g/h]/@i]$.

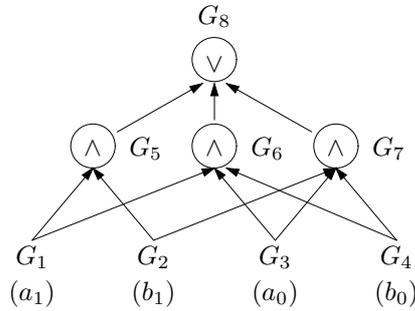


Fig. 6. A 2-bit full adder carry-bit circuit.

Unfortunately, the positive result on the parallel complexity of positive XPath does not extend to full XPath, or even NavXPath.

THEOREM 4.23 [GKPS05]. *NavXPath is PTIME-hard (combined complexity).*

Proof. The proof is by reduction from the *monotone Boolean circuit value* problem, which is PTIME-complete. Note that the classical reduction from PTIME-bounded Turing machines to (monotone) Boolean circuits proving this (see e.g. the proof of Theorem 8.1 in [Pap94]) only produces *layered* circuits.⁴

Given an instance of this problem, a monotone Boolean circuit and a mapping θ that assigns either 0 or 1 to each of the input gates, let M denote the number of input gates and let $N \geq 1$ denote the number of all other gates in the circuit (the

⁴A circuit is called layered if there is a mapping l that assigns to each gate an integer such that if there is an edge from gate G_i to G_j , then $l(G_j) = l(G_i) + 1$.

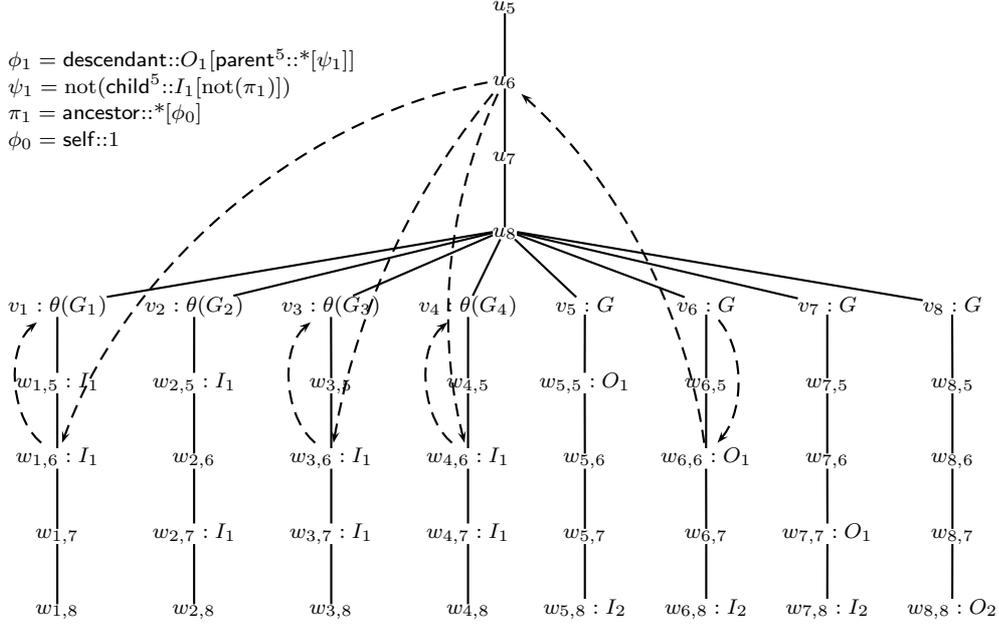


Fig. 7. Document tree corresponding to the carry-bit circuit. The figure also illustrates that $[\phi_1]_{\text{Boolean}}(v_6) \Leftrightarrow \theta(G_1) = 1 \wedge \theta(G_3) = 1 \wedge \theta(G_4) = 1$.

internal gates). Let K be the number of layers in the circuit, that is, the height of the circuit. Let the gates be named $G_1 \dots G_{M+N}$. Without loss of generality⁵, we may assume that the gates $G_1 \dots G_{M+N}$ are numbered in some order such that no gate G_i depends on the output of another gate G_j with $j > i$. In particular, the input gates are named $G_1 \dots G_M$ and the output gate is G_{M+N} . We may assume that there is precisely one gate at the topmost layer K , the output gate.

Figure 6 shows an example of a circuit with appropriately numbered gates. This circuit computes the carry-bit of a two-bit full-adder, i.e. it tells whether adding the two-bit numbers a_1a_0 and b_1b_0 leads to an overflow. The carry-bit c_1 is computed as $(a_1 \wedge b_1) \vee (a_1 \wedge c_0) \vee (b_1 \wedge c_0)$ where $c_0 = a_0 \wedge b_0$ is the carry-bit of the lower digit (a_0 and b_0).

For a given instance of the monotone Boolean circuit value problem, we compute a pair consisting of a document tree and a **NavXPath** query as follows.

The **document tree** consists of nodes u_j , v_i , and $w_{i,j}$ for all $1 \leq i \leq M+N$, $M+1 \leq j \leq M+N$. The root node is u_{M+1} , and there are edges

- from u_j to u_{j+1} for $M+1 \leq j < M+N$,
- from u_{M+N} to v_i and from v_i to $w_{i,M+1}$ for all $1 \leq i \leq M+N$, and
- from $w_{i,j}$ to $w_{i,j+1}$ for all $1 \leq i \leq M+N$, $M+1 \leq j < M+N$.

Node labels are taken from the alphabet $\Sigma = \{0, 1, G, I_1, \dots, I_K, O_1, \dots, O_K\}$ and each tree node is assigned at most one such label. (We allow for “unlabeled” nodes, which can be considered to simply carry a label not from Σ .) This is done as follows. Each node out of v_i for $1 \leq i \leq M$ is assigned $\theta(G_i)$ as a label (either 0 or 1). The nodes $v_{M+1} \dots v_{M+N}$ are each assigned the label G . We assign label I_k to node $w_{i,j}$ iff internal gate G_j is in layer $1 \leq k \leq K$ and takes input from gate G_i . We assign label O_k to node $w_{j,j}$ iff internal gate G_j is in layer k . For our carry-bit example of Figure 6 with $M = 4$ and $N = 4$, the data tree is as shown in Figure 7, where $\theta(G_1), \dots, \theta(G_4) \in \{0, 1\}$ are the truth values a_1, b_1, a_0 , and b_0 , respectively, at the input gates.

⁵The gates can be “sorted” to adhere to such an ordering in logarithmic space. This is trivial if the circuit is layered, which we may assume by the observation made above.

In the following, we will abbreviate the n -times repeated application of an axis χ , $(\chi::*/)^{n-1}\chi::*$, as $\chi^n::*$. By $\chi^n::c$, we denote $(\chi::*/)^{n-1}\chi::c$.

The **query** evaluating the circuit is

$$/\text{descendant}::G[\phi_K]$$

with the condition expressions

$$\begin{aligned} \phi_k &:= \text{descendant}::O_k[\text{parent}^{N+1}::*[\psi_k]] \\ \psi_k &:= \begin{cases} \text{child}^{N+1}::I_k[\pi_k] & \dots \text{ layer } k \text{ consists of } \vee\text{-gates} \\ \text{not}(\text{child}^{N+1}::I_k[\text{not}(\pi_k)]) & \dots \text{ layer } k \text{ consists of } \wedge\text{-gates} \end{cases} \\ \pi_k &:= \begin{cases} \text{ancestor}::G[\phi_{k-1}] & \dots k > 1 \\ \text{ancestor}::*[\phi_{k-1}] & \dots k = 1 \end{cases} \end{aligned}$$

for $1 \leq k \leq K$ and $\phi_0 := \text{self}::1$.

It uses the intuition of processing the circuit one layer at a time.

We will check whether our query on our document includes the particular node v_{M+N} . Indeed, by our construction, the query will select node v_{M+N} iff the circuit evaluates to true, and no other node will be selected.

It is easy to see that the reduction can be effected in LOGSPACE. We next argue that it is also correct.

The ϕ_k , ψ_k , and π_k are condition expressions (qualifiers), and we have already given a formal meaning $\llbracket \phi_k \rrbracket_{\text{Boolean}}(w)$ to the notion “ ϕ_k matches node w ” or equivalently “node w satisfies ϕ_k ” (and analogously to $\llbracket \psi_k \rrbracket_{\text{Boolean}}(w)$ and $\llbracket \pi_k \rrbracket_{\text{Boolean}}(w)$).

Claim. *Let $0 \leq k \leq K$. Then, for all gates G_i in layer k ,*

$$\llbracket \phi_k \rrbracket_{\text{Boolean}}(v_i) \Leftrightarrow \text{gate } G_i \text{ evaluates to true.}$$

This can be shown by an easy induction.

Induction start ($k = 0$). The gates of layer 0 are the input gates. By definition, an input gate G_i is true iff node v_i is labeled 1. but on precisely these nodes $\phi_0 = \text{self}::1$ is true. Thus our claim holds for $k = 0$.

Induction step. Now assume that our claim holds for ϕ_{k-1} . We show that it also holds for ϕ_k .

To start, it is easy to see that for all i, j ,

$$\llbracket \pi_k \rrbracket_{\text{Boolean}}(w_{i,j}) \Leftrightarrow \llbracket \phi_{k-1} \rrbracket_{\text{Boolean}}(v_i).$$

Now observe that by our construction of the data tree, the nodes $w_{1,j}, \dots, w_{j,j-1}$ encode the connections of gate G_j with its inputs. Gate G_i is an input to gate G_j if and only if node $w_{i,j}$ is labeled I_k , for k the layer of gate G_j . The node $w_{j,j}$ is labeled O_k . Observe also that the node u_j is precisely $N + 1$ levels above the nodes $w_{1,j}, \dots, w_{M+N,j}$ in the data tree.

For \vee -gate G_j in layer k ,

$$\begin{aligned} \llbracket \psi_k \rrbracket_{\text{Boolean}}(u_j) &\Leftrightarrow \exists i I_k(w_{i,j}) \wedge \llbracket \pi_k \rrbracket_{\text{Boolean}}(w_{i,j}) \\ &\Leftrightarrow \text{gate } G_i \text{ is an input to } G_j \text{ and } G_i \text{ is true} \end{aligned}$$

for \wedge -gate G_j in layer k ,

$$\begin{aligned} \llbracket \psi_k \rrbracket_{\text{Boolean}}(u_j) &\Leftrightarrow \forall i I_k(w_{i,j}) \rightarrow \llbracket \pi_k \rrbracket_{\text{Boolean}}(w_{i,j}) \\ &\Leftrightarrow \text{all inputs to } G_j \text{ are true} \end{aligned}$$

Finally, since

$$\llbracket \phi_k \rrbracket_{\text{Boolean}}(v_j) \Leftrightarrow \llbracket \psi_k \rrbracket_{\text{Boolean}}(u_j),$$

our claim is shown for ϕ_k , $0 \leq k \leq K$.

Figure 7 illustrates the computation of the truth value of gate G_6 of our circuit example.

The overall query $/\text{descendant}::G[\phi_K]$ has a nonempty result (consisting of precisely the node v_{M+N}) exactly if the output gate G_{M+N} of the circuit evaluates to true, because G_{M+N} is the only gate in layer K , v_{M+N} is the only node labeled G that has an O_K descendant, and $\llbracket\phi_K\rrbracket_{\text{Boolean}}(v_{M+N})$ if and only if G_{M+N} evaluates to true.

In summary, we have provided a LOGSPACE reduction that maps any monotone Boolean circuit to a NavXPath query and a document tree such that the query evaluated on the tree returns node v_{M+N} precisely if the circuit evaluates to true. As the monotone Boolean circuit value problem is P-complete, our theorem is proven. \square

Note that the above proof of the PTIME lower bound does not employ axis steps with multiple qualifier brackets $\text{axis}[:]\dots[:]$; indeed, as observed before, even for AggXPath, $\text{axis}[q_1]\dots[q_n]$ is equivalent to $\text{axis}[q_1 \wedge \dots \wedge q_n]$, but this is not true for OrdXPath. And indeed, the interaction of multiple qualifier brackets and position arithmetics has an impact on the complexity of XPath:

THEOREM 4.24 [GKPS05]. *Positive OrdXPath is PTIME-hard w.r.t. combined complexity.*

The PTIME-hardness result actually only uses a fragment of OrdXPath with $\text{last}()$ and steps with multiple qualifier brackets, but without $\text{position}()$ or aggregation operations.

We give a brief overview over the remaining complexity results known for XPath. First, the PTIME-hardness result of Theorem 4.23 essentially depends on the presence of transitive axes: NavXPath using only the child and parent axes is in LOGSPACE w.r.t. combined complexity [GKPS05].

The data complexity of XPath depends on encodings. XPath 1.0 on DOM trees (pointer structures) is LOGSPACE-complete if the concatenation operation on strings and multiplication are excluded from the language.

So far, we have always assumed that the input is basically given as a pointer structure (using signature σ_{dom}). But XML documents can also be considered in their natural textual (string) representation. The distinction is only relevant for the very small complexity class inside LOGSPACE, for which completeness is usually defined in terms of reductions not strong enough to map between DOM trees and strings. On string representations, NavXPath was shown to be in TC^0 [GKPS05], a complexity class inside LOGSPACE. Of course, on a relational encoding of the tree with all binary axis relations part of the encoding, FOXPath is first-order and inherits its AC^0 upper bound (yet inside TC^0) on the data complexity.

The query complexity of XPath 1.0 is in LOGSPACE [GKP05]. This is a slightly curious fact. While for virtually all known traditional query languages, the query complexity is greater than the data complexity by at least an exponential factor (cf. e.g. [AHV95]), this is not the case of XPath.

4.6 Stream Processing

Because of the role of XML as a data exchange format, the problem of evaluating XPath on streaming XML data has attracted quite some research work (cf. e.g. [AF00; CFGR00; OMFB02] for early work). A natural technique for processing XPath on streams is based on automata [GMOS03]. In this context, the *XPath filtering problem*, the problem of testing whether a given XPath query relative to the root node has any matches (i.e., the problem of testing whether $\llbracket p \rrbracket_{\text{Boolean}}(\text{root})$ is true for query p), has found some consideration. The usual scenario is that of a stream of XML documents and a set of XPath queries describing subscriptions to documents on the stream matching the XPath queries, and has been referred to by *selective dissemination of information*. This problem has been considered in [AF00;

CFGR00; GMOS03] with the additional difficulty that algorithms have to scale to very large numbers – even millions – of queries to be matched in parallel.

A *streaming algorithm* scans its input data once – and only once – from left to right. Since data streams for practical purposes can be assumed to be infinitely long, or at least very long, one usually assumes that main memory is a limited resource. Ideally, streaming algorithms should cope with a fixed amount of memory, but as we will see below, constant memory is not sufficient for evaluating even the simplest XPath queries.

Starting with [BYFJ04], techniques from communication complexity have been used for studying memory lower bounds of streaming XPath evaluation algorithms [BYFJ04; BYFJ05; GKS05]. We only give one such lower bound result which uses the standard notion of complexity for XPath queries. We denote the depth of a tree T by $\text{depth}(T)$. It has been observed that

PROPOSITION 4.25 [GKS05]. *There can be no streaming algorithm with memory consumption $o(\text{depth}(\cdot))$ for the NavXPath filtering problem.*

Of course, there are trees whose depth is linear in their size, so one can read this result in the sense that there can be no streaming algorithm for NavXPath that takes space less than linear in the *size* of the XML stream, so memory-efficient – and thus scalable – stream processing for XPath is, from a certain point of view, in the worst case impossible. Fortunately, XML trees tend to be shallow in practice, so showing this lower bound tight can be considered a positive result.

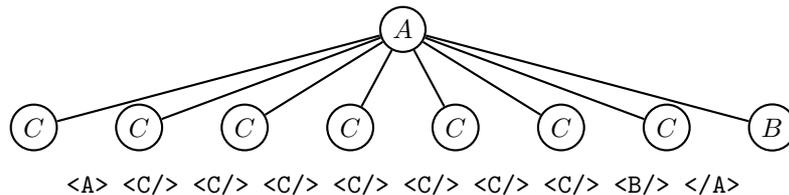
As discussed early in this section, bottom-up tree automata allow to check MSO sentences in a single traversal of the tree. Using automata-based techniques, checking MSO queries, and thus solving the XPath filtering problem, is feasible using only memory of size bounded by the depth of the tree (which in practice, for XML, is small).

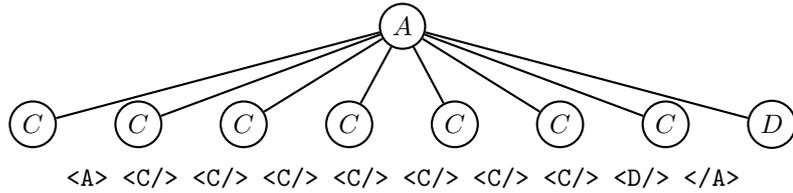
THEOREM 4.26 IMPLICIT IN [NS98; SV02]. *Let \mathbf{T} be a tree-language. If \mathbf{T} is definable by an MSO-sentence over vocabulary σ_{nav} , then \mathbf{T} can be recognized by a streaming algorithm using memory $O(\text{depth}(\cdot))$.*

COROLLARY 4.27. *There is a streaming algorithm for the NavXPath filtering problem with memory consumption $O(\text{depth}(\cdot))$.*

Translating XPath queries into deterministic pushdown automata has been studied in several works [GMOS03; GS03]; the blow-up required to compute such automata is exponential, and the precise sources of this exponentiality were explored in [GMOS03]. The first work to present a streaming algorithm for the XPath filtering problem that takes only memory linear in the depth of the tree and runs in time polynomial in the size of (the data and) the query was [OKB03]; there, the exponential size of automata is avoided by not compiling automata for managing and recognizing the subexpressions of an XPath query into a single automata but keeping them apart, as a *transducer network*. A similar transducer-network based approach to streaming XPath processing was developed in [PC03]. A different algorithm for polynomial-time streaming XPath processing was presented in [JF05].

For the problem of selecting nodes matched by XPath queries, the situation is worse. Consider the trees





To select the C -nodes of the upper tree but not those of the lower tree, the query $/\text{child}::A[\text{child}::B]/\text{child}::C$ will in the worst case have to buffer all C -children of the A -node before a B -node is seen on the stream that confirms that the C -nodes are to be selected. In the worst case this may amount to all but two nodes of the document.

4.7 Further Bibliographic Remarks

The dynamic programming algorithm for full XPath 1 of [GKP05] demonstrates in a rather straightforward way that XPath 1 can be evaluated in polynomial time. When introduced, this algorithm was the first of its kind, and it was observed that all XPath engines available at the time were taking exponential time in the worst case for evaluating XPath 1. However, the dynamic programming algorithm computes many useless intermediate results and consumes much memory. To fix this, a more efficient top-down algorithm is given in [GKP05] as well. This algorithm still runs in polynomial time, with better worst-case upper bounds on running time and memory consumption. Further work on polynomial-time algorithms for full XPath 1 which elaborates on the results of [GKP05] and integrates them into a native XML database management system can be found in [BHKM05]. This work also shows how to integrate XQuery and efficient XPath processing using a single native algebra.

[BGK03; FGK03] studies XPath query evaluation on XML data compressed using a bisimulation-based tree compression scheme.

XPath plays an important role as part of XQuery, and there has been much work on processing XPath and tree pattern queries both in the context of native XML databases and even more so on relational representations of XML databases. Writing research papers on the evaluation of structural joins [AKJP⁺02], i.e., joins along the navigational structure of an XML tree, tree pattern or twig queries [BSK02], as well as labeling schemes for assigning identity to tree nodes (which is particularly important in XML-to-relational mappings) has become a favorite pastime of data management researchers, and we shall not try to list all work on this topic here.

5. STATIC ANALYSIS

5.1 Satisfiability

It is fairly easy to see that satisfiability of Navigational XPath expressions is decidable. One argument is via Proposition 3.1, and the fact that first-order logic over finite ordered labeled trees is known to be decidable [TW68]. The standard proof of decidability for first-order logic is via an inductive translation into a tree automaton. Because complementation of an automaton requires an exponential blow-up in size at every negation step, the complexity of satisfiability for first-order logic over trees is known to be non-elementary [TW68]. However, in the previous section we have shown that NavXPath Boolean queries translate into two-variable first-order logic. The satisfiability problem for FO^2 over arbitrary finite structures is known to be in NEXPTIME [GKV97]. In addition, [GKV97] shows that satisfiable FO^2 sentences have models of size exponential in the size of the sentence. However, this does not imply that the satisfiability problem for FO^2 is in NEXPTIME, since for this problem we have the constraint that the models must be trees (a constraint which is not expressible by an FO^2 sentence).

In [EVW02] it is shown that the satisfiability of FO^2 sentences over words is in NEXPTIME. We modify this below to show the satisfiability problem for trees is in NEXPTIME. Since the translation of NavXPath into FO^2 given in Section 3 is polynomial, we get a NEXPTIME bound for NavXPath.

THEOREM 5.1. *There is an NEXPTIME algorithm deciding for a given sentence $\phi \in FO^2$ whether or not it is satisfiable by some ordered tree.*

Recall that Proposition 3.4 shows that unnested NavXPath^\cap , the extension of NavXPath with an intersection operator but where union may only occur on the top level, can be translated in polynomial time into FO^2 . From this and Theorem 5.1, it follows that:

COROLLARY 5.2. *The satisfiability problem for unnested NavXPath^\cap (and hence for unnested NavXPath) is in NEXPTIME.*

We will see that this bound is not tight for NavXPath. We do not know the complexity of satisfiability for full NavXPath^\cap . A related language is PDL with an intersection operator, where the satisfiability problem has recently been shown to be 2-EXPTIME hard even on one-letter trees [LL05]. However, this language is more expressive than NavXPath^\cap .

Since we know of no proof of Theorem 5.1 in the literature, we sketch one, following closely the approach of [EVW02]. First, we translate the problem of satisfiability on unranked trees to one on binary trees, using the standard encoding of an unranked tree as a binary tree. Let $FO^2[\sigma_{nav,bin}]$ be FO^2 over the unary signature Σ unioned with FChild, SChild (the first- and second-child relations of the binary tree representation), SChild*, $R_{\text{descendant}}$. We consider a formula of $FO^2[\sigma_{nav,bin}]$ to be interpreted over *binary codes of unranked trees*, structures $T = (V, \dots)$ in which *i)* $(V, \text{FChild} \cup \text{SChild})$ is a tree of outdegree at most two, *ii)* each node is related to at most one node via FChild and at most one variable SChild, with these nodes being distinct, and *iii)* $R_{\text{descendant}}$ is the transitive closure of $\text{FChild} \cup \text{SChild}$, and SChild^* is the transitive closure of SChild. The following is simple to show:

PROPOSITION 5.3. *Satisfiability of FO^2 sentences over unranked trees is reducible in polynomial time to satisfiability of $FO^2[\sigma_{nav,bin}]$ sentences over binary codes of unranked trees.*

For an integer k , a *k-type* is a maximal consistent set of $FO^2[\sigma_{nav,bin}]$ formulas (in some fixed set of variables) where the maximal number of nested quantifiers (i.e. quantifier rank) is at most k . We will deal with k -types in 1 free variable, with such a type typically denoted $\tau(x)$. A binary code structure (V, \dots) is *k-compact* if:

- We do not have nodes $v_1, v_2 \in V$ with the same k -type, and with v_2 a descendant of v_1 .
- Any two nodes with the same k -type have identical subtrees.

The next result shows that we can reduce satisfiability to a search for compact structures:

LEMMA 5.4. *An $FO^2[\sigma_{nav,bin}]$ sentence of quantifier rank $k > 1$ is satisfiable at the root of some binary code iff it is satisfiable at the root of a k -compact binary code.*

Proof. Let ϕ be an $FO^2[\sigma_{nav,bin}]$ sentence of quantifier rank k , and suppose ϕ is satisfiable in $B = (V, \dots)$, and B is the structure of minimal size satisfying ϕ . Suppose there are nodes $v_1, v_2 \in V$ with the same k -type, with v_2 a descendant of v_1 . Let S_1 be all nodes that are descendants of v_1 but are not descendants of v_2 (including v_2). Let B' be the code formed by removing all nodes in S_1 and attaching the subtrees of v_2 to v_1 (i.e. the first child of v_2 becomes the first child

of v_1 , etc.). Let f be the mapping from B' to B that maps a node beneath v_1 in B' to the corresponding node beneath v_2 , and is the identity elsewhere on B' . We now show by induction on i that for each $i \leq k$, the i -type of a node $v \in B'$ is the same as the i -type of $f(v) \in B$.

For $i = 0$ this is clear, since the only atomic formulas in one variable are those that assert the label of a node, and the mapping f preserves labels. For the inductive step $i + 1$, note that a two-variable formula $\phi(x)$ of rank $i + 1$ can be taken to assert the existence or non-existence of a y with a certain axis relation to x and with a fixed i -type. All formulas asserting the non-existence of such a y are clearly preserved from x to $f(x)$, by induction. Suppose that for $x \in B'$ there is a y in B with i -type τ and with a given axis relationship to $f(x)$. If $y = f(w)$ for some w in B' , then we can choose w as a witness to τ in B' , since w will satisfy the same axis relation to x as y does to $f(x)$ (by definition of f), and will satisfy the same i -type as y by induction. Otherwise, it must be that y lies below v_1 but is incomparable to v_2 . Since y lies below v_1 and v_2 has the same k -type in B (hence the same $i + 1$ -type) as v_1 , there is y' below v_1 satisfying the same axes with respect to v_1 as y has to v_2 , and such that the i -type of y' in B is the same as the i -type of y in B . Since y' is below v_1 , $y' = f(w)$ for some $w \in B'$, and now we are done by induction.

The result of the construction above is a smaller tree in which the k -type of the root has the same type as in the original tree, thus violating minimality.

To get the second part of compactness, let Γ be the set of k -types $\tau(x)$ such that the second part is violated in B' : that is, there are two nodes with type τ with distinct subtrees. We proceed by downward induction on $n = |\Gamma|$. If $n > 0$, choose a node $v \in B'$ satisfying a type in Γ that has maximal depth in the tree. Let τ be the k -type of v and S_v be the forest consisting of all descendants of v in B' . All nodes in S_v must satisfy a type outside of Γ . For every other node v' in B' satisfying τ , we replace the forest below v' with S_v (making the subtree below the first child of v into the subtree below the first child of v' , etc.). Notice that the first condition of compactness (already holding of B') ensures that v' is not comparable to v . One can confirm by induction that the k -type of the root is unchanged by this substitution, by an argument identical to that used in the first part of this lemma. In this process, n is decreased by one, and hence the process terminates with a k -compact tree. \square

From Lemma 5.4, Theorem 5.1 follows. The depth of a k -compact tree is at most the number of k -types, which is bounded by an exponential in ϕ . Furthermore, a k -compact tree can be represented via a DAG whose nodes are the k -types realized in the tree. Such a DAG represents the tree formed by duplicating shared subtrees. It is easy to see that one can check whether a given sentence is satisfied on a DAG representation of a tree in polynomial time. Our NEXPTIME algorithm just guesses a DAG structure on the k -types, and then confirms that the corresponding tree satisfies the sentence ϕ .

It is known that FO^2 is NEXPTIME-hard [EVW02]. The example showing NEXPTIME hardness from [EVW02] can be coded easily in unnested NavXPath^\cap , hence we have that:

THEOREM 5.5. *The satisfiability problem for unnested NavXPath^\cap is complete for NEXPTIME.*

From this proof, we get further information:

COROLLARY 5.6 TO THE PROOF OF THEOREM 5.1. *Let ϕ be an FO^2 sentence. If ϕ is satisfiable in some finite tree, then it is satisfiable in some tree of depth exponential in $|\phi|$ and size doubly exponential in $|\phi|$. The same holds for E an expression in unnested NavXPath extended with the intersection operator.*

Is this NEXPTIME-bound tight for NavXPath? First note that the fact that FO^2 is NEXPTIME-hard does not imply the same for NavXPath, since the translation from FO^2 to NavXPath is exponential. [Mar04b] shows that satisfiability of NavXPath expressions can be decided in deterministic exponential time.

THEOREM 5.7 [MAR04B]. *NavXPath satisfiability is decidable in EXPTIME. Furthermore, since equivalence for NavXPath expressions can be reduced to satisfiability of a single expression, the equivalence problem can be decided in EXPTIME.*

[Mar04b] actually shows this for an extension of NavXPath that allows regular expressions on axes. Since the treatment in Marx’s papers [Mar04b; Mar04a; ABD⁺05] is quite detailed, we give here only some comments on the proof. The proof is by reduction to the satisfiability problem for Deterministic Propositional Dynamic Logic (PDL) with Converse. PDL is similar to XPath, in that it is a modal language that allows the definition of binary relations (in dynamic logic “programs”) as well as unary relations (“formulas”). As with XPath, the grammars for binary relations and unary relations are mutually recursive. Dynamic logics have a different data model than XPath, being defined over node and edge-labeled graphs. However, since formulas in the language can see only a part of the graph at a time, the behavior of the logic on general structures is closely related to its behavior on trees. Deterministic PDL with converse is formed over a set of atomic programs (analogous to axes in XPath) each of which is a function maps nodes in a graph to at most one other node. For each atomic program there is a “converse program” representing the inverse of the binary relation. In a binary tree the “first child” and “second child” relations are functional; here we can interpret Deterministic PDL with Converse with two atomic program over binary trees, with the two programs chosen to be first and second child. Using the standard encoding of ordered unranked trees as binary trees, deterministic PDL with Converse over two programs can be interpreted on ordered trees. Because PDL allows new binary relations to be built up from old using regular expressions, the recursive axes, and in fact all of NavXPath (and more [Mar04b]), can be defined within it. Hence the satisfiability of XPath is reduced to the satisfiability problem for Deterministic PDL with Converse sentences over binary trees. In [VW86] it is shown that deterministic PDL with converse is decidable over all structures is in EXPTIME. The proof relies on translating PDL programs into alternating automata on trees. [Mar04b] shows that the proof in [VW86] can be modified to give the same bound over the class of codings of finite ordered trees. In [ABD⁺05], a variant of PDL defined directly on ordered trees is given, which yields an alternate route (also going through [VW86]) to the EXPTIME bound.

[NS03] shows that containment of NavXPath expressions is EXPTIME-hard. An inspection of the proof shows that only unnested expressions are needed for the hardness proof. Since containment of two (unnested) NavXPath expressions can be reduced to satisfiability of a single (unnested) expression, it follows that unnested NavXPath satisfiability is EXPTIME-hard. Hence we see that the EXPTIME bound is tight:

COROLLARY 5.8 COMBINING [NS03] AND [MAR04B]. *The satisfiability problems for both NavXPath and unnested NavXPath are EXPTIME-complete.*

5.2 Satisfiability for other XPath fragments

Now that we know that NavXPath has EXPTIME satisfiability, we can look at what happens as features are added or subtracted.

Better bounds can be obtained for sublanguages of NavXPath: Satisfiability of NavXPath with only child and parent is shown to be PSPACE-complete in [BFG05]. Satisfiability for PNavXPath is easily seen to be in NP (see [Hid03]), and this is

extended to PFOXPath in [BFG05]. It is also shown in [BFG05] that very simple fragments of PNavXPath have an NP-complete satisfiability problem – in the presence of both downward and upward axes, the problem is NP-complete, as well as in the presence of both left and right sibling axes. For PNavXPath with only downward axes, all expressions are clearly satisfiable; however, the satisfiability problem with respect to a given DTD can be NP-hard [BFG05].

We now consider satisfiability as we move up in expressiveness from NavXPath. It is shown in [BFG05] that the satisfiability of a FOXPath expression with respect to a DTD is undecidable. By using sibling axes instead of a DTD, one can see the following:

THEOREM 5.9 [GF05]. *The satisfiability problem for FOXPath is undecidable.*

The proof uses a reduction from the halting problem for two-register machines which is known to be undecidable (see, e.g., [BGG97]). Although full FOXPath is undecidable, the exact borderline of decidability is not well understood.

QUESTION 5.10. *Is FOXPath without the sibling axes decidable?*

In fact, decidability is open even in the case of FOXPath with only child and parent.

One can also look at decidability on restricted classes of documents:

QUESTION 5.11. *Is FOXPath decidable on documents with no branching (i.e. those where every element has at most one child)?*

5.3 Containment

The *containment problem* takes as input XPath expressions E and E' , asking whether the output of E is contained in the output of E' on any source document at any node. Variations of the problem are *containment with respect to a DTD*, which takes a DTD as an additional argument, asking whether the above holds for E and E' over any source document satisfying the DTD. A special case of this is the *containment problem for a finite alphabet*, which takes a label alphabet Σ as additional parameter, asking whether containment holds for all source documents with labels in Σ .

The containment problem has been investigated extensively in the relational case for conjunctive queries, where it has close connections both to issues in data integration and query optimization, as well as to constraint satisfaction [KV00; GLS01]. The general conjunctive query containment problem is known to be NP complete; however, many special cases are known to be in PTIME, including those in which the dependency graphs of the queries have bounded tree-width [CR97] or the queries have bounded hypertree-width [GLS99]. In the case of conjunctive queries, containment of Q_1 in Q_2 reduces to determining whether Q_1 is satisfiable on an instance formed from Q_2 , hence the complexity of containment is bounded by the combined complexity of evaluation. In the XPath setting there is no obvious correspondence between a query and a “canonical instance”, and indeed the complexity of containment and evaluation turn out to be quite different.

Starting with the relational case as motivation, [AYCLS01; MS02; Woo01] initiated the study of containment for XPath, beginning with subclasses of NavXPath without either the union operator or disjunction within filters (conjunctive NavXPath). The survey article of Schwentick [Sch04] gives an overview of the techniques used in getting bounds on containment; here we summarize only some of the results and the open questions. A modification of the minimal model technique for conjunctive queries shows that the containment problem for conjunctive Navigational XPath is in co-NP – given queries P and Q one can generate a finite set of instances $I_i : i < n$ of size polynomial in P such that $P \subseteq Q$ iff each I_i satisfies Q [MS02].

Since satisfaction can be checked in linear time, a CO-NP algorithm is simply to guess an I_i that fails to satisfy Q . In [AYCLS01], it is shown that for conjunctive NavXPath with only descendant axes the containment problem is in PTIME, while in [Woo01] it is noted that the same holds for conjunctive NavXPath with only child axes (indeed this last observation follows directly from the PTIME bounds for acyclic conjunctive queries in [CR97]). When both descendant axes and child axes are present the problem was shown to be CO-NP-complete [MS02]. [NS03] shows that the containment problem for conjunctive NavXPath with a finite alphabet is PSPACE-complete, while the containment problem with respect to a DTD is EXPTIME-complete. A finer analysis of the complexity of containment for conjunctive NavXPath with respect to a DTD and with respect to integrity constraints is given in [Woo03].

The complexity of containment for fragments of XPath larger than conjunctive NavXPath was studied by Neven and Schwentick. For PNavXPath, the general containment problem remains in CO-NP, while if the alphabet is fixed the problem is again PSPACE-complete [NS03]. For full NavXPath, the containment problem, even with respect to a DTD, is in EXPTIME, since it is reducible to the satisfaction problem: this is noted in [Mar04b]. On the other hand, since [NS03] shows that containment of NavXPath expressions is EXPTIME-hard, we have:

THEOREM 5.12 COMBINING [NS03] AND [MAR04B]. *The containment problem for NavXPath is EXPTIME-complete, as is the containment problem for finite alphabet and the containment problem with respect to a DTD.*

When we turn to the XPath fragments with data values, the complexity of containment is not completely understood. The results of Deutsch and Tannen [DT01] imply that containment for PFOXPath is CO-NP-complete, provided that the transitive sibling axes are not permitted and "wildcard steps" (child steps with no restriction on the label) are disallowed. Their technique also yields a Π_2^P bound for full PFOXPath, although neither their terminology nor their fragments match PFOXPath exactly. They also establish Π_2^P bounds in the presence of integrity constraints called SXICs: these are incomparable to both finite alphabets and DTDs. [DT01] also provides lower bounds for containment in the presence of integrity constraints. Neven and Schwentick [NS03] show that PFOXPath without sibling axes and without wildcard is in Π_2^P , and that the containment problem for PFOXPath extended with inequality is undecidable.

To our knowledge, the decidability of containment for general conjunctive FOXPath queries with respect to a DTD or a finite alphabet is open. Indeed we do not know whether one can decide containment of conjunctive queries over signature σ'_{dom} ⁶ in the presence of DTDs. The undecidability techniques of [NS03] rely on disjunction, while [DT01] provides undecidability results with respect to integrity constraints. The upper bounds of both [NS03; DT01] rely on the use of an infinite alphabet.

5.4 Further Bibliographic Remarks

While above we have dealt with the satisfiability and containment problems, a broader goal would be an algebraic simplification framework for XPath. [BFK03] presents algebraic equations for simplification of XPath expressions. A system of equations is presented that is complete for equivalence of XPath expressions for a very small fragment (without filters and with only child axes). [OMFB02] gives a rewriting system geared not toward general equivalence, but for removing backward axes. [AYCLS01] deals not with equivalence but with optimization; it presents an algorithm for minimization of tree patterns in the presence of integrity constraints.

⁶Recall that this is the relational signature with binary predicates for the graph of each attribute function, unary predicates for the labels, and binary predicates for the major axes.

A natural question not addressed above is the implementation of satisfiability and containment tests for XPath. [BBFV05] implements a satisfiability test for a fragment of PNavXPath, in the presence of DTDs, based on a conversion to tree automata. [LRWZ04] implements a satisfiability test for a tree pattern language that includes data value manipulation (incomparable in expressiveness with the XPath languages we consider here).

An additional static analysis problem is recognizing whether a query is in a given XPath fragment. In the context of navigational XPath, the problem of recognizing whether a first-order logic query is in NavXPath is open. This is closely-related to the (likewise open) problem of determining whether a tree automaton is equivalent to an FO^2 sentence. The problem of determining whether a first-order query over σ'_{dom} is in FOXPath is undecidable – this follows from the results of [BFG05]. The problem of determining whether a conjunctive query over σ'_{dom} is expressible in conjunctive FOXPath has not been investigated (to our knowledge). Likewise, nothing is known concerning the problem of determining whether a first-order query (or a NavXPath query) is equivalent to a query in PNavXPath.

REFERENCES

- Loredana Afanasiev, Patrick Blackburn, Ioanna Dimitriou, Bertrand Gaiffe, Evan Goris, Maarten Marx, and Maarten de Rijke. “PDL for Ordered Trees”. *Journal of Applied Non-Classical Logics*, 15:115–135, 2005.
- Mehmet Altinel and Mike Franklin. “Efficient Filtering of XML Documents for Selective Dissemination of Information”. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'2000)*, pages 53–64, Cairo, Egypt, 2000.
- Loredana Afanasiev, Massimo Franceschet, Maarten Marx, and Maarten de Rijke. “CTL Model Checking for Processing Simple XPath Queries”. In *Proc. TIME*, pages 117–124, 2004.
- Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. “Structural Joins: A Primitive for Efficient XML Query Pattern Matching”. In *18th International Conference on Data Engineering (ICDE'02)*, 2002.
- S. Amer-Yahia, S. Cho, Laks V.S. Lakshmanan, and Divesh Srivastava. “Minimization of Tree Pattern Queries”. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*, pages 497–508, Santa Barbara, California, USA, 2001.
- Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. “Verification of Tree Updates for Optimization”. In *Proceedings of the 17th International Conference on Computer Aided Verification*, 2005.
- Steven Bird, Yi Chen, Susan Davidson, Haejoong Lee, and Yifeng Zheng. “Extending XPath to Support Linguistic Queries”. In *PLAN-X*, 2005.
- J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. “Symbolic Model Checking: 10^{20} States and Beyond”. In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science (LICS)*, 1990.
- Michael Benedikt, Wenfei Fan, and Floris Geerts. “XPath Satisfiability in the presence of DTDs”. In *Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'05)*, 2005.
- Michael Benedikt, Wenfei Fan, and Gabriel Kuper. “Structural Properties of XPath Fragments”. In *Proc. of the 9th International Conference on Database Theory (ICDT)*, pages 79–95, Siena, Italy, 2003.
- Egon Börger, Eric Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Springer, 1997.
- Peter Buneman, Martin Grohe, and Christoph Koch. “Path Queries on Compressed XML”. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 141–152, 2003.
- Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. “Full-fledged Algebraic XPath Processing in Natix”. In *Proceedings of the 21st IEEE International Conference on Data Engineering (ICDE)*, 2005.
- Michael Benedikt and Christoph Koch. “XPath with Data Values Revisited”, 2005.
- Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. “Regular Tree and Regular Hedge Languages over Non-ranked Alphabets: Version 1, April 3, 2001”. Technical Report HKUST-

- TCSC-2001-05, Hong Kong University of Science and Technology, Hong Kong SAR, China, 2001.
- Danièle Beauquier and Jean-Eric Pin. “Factors of Words”. In *Proc. ICALP*, pages 63–79, 1989.
- Nicolas Bruno, Divesh Srivastava, and Nick Koudas. “Holistic Twig Joins: Optimal XML Pattern Matching”. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD’02)*, Madison, Wisconsin, June 2002.
- Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. “On the Memory Requirements of XPath Evaluation over XML Streams”. In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS’04)*, pages 177–188, 2004.
- Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. “Buffering in Query Evaluation over XML Streams”. In *Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS’05)*, 2005.
- Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE)*, San Jose, California, USA, February 26–March 1, 2002, 2000.
- E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
- Julien Carme, Joachim Niehren, and Marc Tommasi. “Querying Unranked Trees with Stepwise Tree Automata”. In *Rewriting Techniques and Applications*, 2004.
- Bruno Courcelle. “Graph Rewriting: An Algebraic and Logic Approach”. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2, chapter 5, pages 193–242. Elsevier Science Publishers B.V., 1990.
- Chandra Chekuri and Anand Rajaraman. Conjunctive Query Containment Revisited”. In *Proc. of the 6th International Conference on Database Theory (ICDT)*, pages 56–70, Delphi, Greece, 1997.
- J. Doner. “Tree Acceptors and some of their Applications”. *Journal of Computer and System Sciences*, 4:406–451, 1970.
- Alin Deutsch and Val Tannen. Containment and Integrity Constraints for XPath. In *Proc. KRDB 2001*, CEUR Workshop Proceedings 45, 2001.
- Kousha Etessami, Moshe Vardi, and Thomas Wilke. “First Order Logic with Two Variables and Unary Temporal Logic”. *Information and Computation*, 179, 2002.
- Kousha Etessami and Thomas Wilke. “An Until Hierarchy and Other Applications of an Ehrenfeucht-Fraïssé Game for Temporal Logic”. *Information and Computation*, 160:88–108, 2000.
- W. Fan, CheeYong Chan, and M. Garofalakis. Secure XML querying with security views. In *SIGMOD*, 2004.
- Jörg Flum, Markus Frick, and Martin Grohe. “Query Evaluation via Tree-Decompositions”. *Journal of the ACM*, 49(6):716–752, 2002.
- Markus Frick, Martin Grohe, and Christoph Koch. “Query Evaluation on Compressed Trees”. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS)*, Ottawa, Canada, June 2003.
- Floris Geerts and Wenfei Fan. “XPath Satisfiability with Sibling Axes”. In *Proc. 10th DBPL*, 2005.
- Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.
- Georg Gottlob and Christoph Koch. “Monadic Queries over Tree-Structured Data”. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 189–202, Copenhagen, Denmark, July 2002.
- Georg Gottlob and Christoph Koch. “Monadic Datalog and the Expressive Power of Web Information Extraction Languages”. *Journal of the ACM*, 51(1):74–113, 2004.
- Georg Gottlob, Christoph Koch, and Reinhard Pichler. “Efficient Algorithms for Processing XPath Queries”. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 95–106, Hong Kong, China, 2002.
- Georg Gottlob, Christoph Koch, and Reinhard Pichler. “Efficient Algorithms for Processing XPath Queries”. *ACM Transactions on Database Systems*, 30(2):444–491, June 2005.
- Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. “The Complexity of XPath Query Evaluation and XML Typing”. *Journal of the ACM*, 52(2):284–335, March 2005.
- Georg Gottlob, Christoph Koch, and Klaus U. Schulz. “Conjunctive Queries over Trees”. In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS’04)*, pages 189–200, Paris, France, 2004.
- Martin Grohe, Christoph Koch, and Nicole Schweikardt. “Tight Lower Bounds for Query Processing on Streaming and External Memory Data”. In *Proc. ICALP*, 2005.

- Erich Grädel, Phokion Kolaitis, and Moshe Vardi. On the decision problem for two-variable first-order logic. *Bulletin of Symbolic Logic*, 3:53–69, 1997.
- Georg Gottlob, Nicola Leone, and Francesco Scarcello. “Hypertree Decompositions and Tractable Queries”. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS’99)*, pages 21–32, 1999.
- Georg Gottlob, Nicola Leone, and Francesco Scarcello. “The Complexity of Acyclic Conjunctive Queries”. *Journal of the ACM*, **48**(1):431–498, 2001.
- Georg Gottlob, Nicola Leone, and Francesco Scarcello. “Hypertree Decompositions and Tractable Queries”. *Journal of Computer and System Sciences*, **64**(3):579–627, 2002.
- Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. “Processing XML Streams with Deterministic Automata”. In *Proc. of the 9th International Conference on Database Theory (ICDT)*, 2003.
- A. K. Gupta and D. Suciu. “Stream Processing of XPath Queries with Predicates”. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD’03)*, pages 419–430, 2003.
- Jan Hidders. “Satisfiability of XPath Expressions”. In *Proc. 9th DBPL*, 2003.
- Neil Immerman. *“Descriptive Complexity”*. Springer Graduate Texts in Computer Science, 1999.
- Vanja Josifovski and Marcus F. Fontoura. “Querying XML Streams”. *VLDB Journal*, **14**(2):197–210, April 2005.
- David S. Johnson. “A Catalog of Complexity Classes”. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 1, chapter 2, pages 67–161. Elsevier Science Publishers B.V., 1990.
- H. Kamp. *“Tense Logic and the Theory of Linear Order”*. PhD thesis, University of California, Los Angeles, 1968.
- Christoph Koch. “Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach”. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 249–260, 2003.
- Phokion Kolaitis and Moshe Vardi. “Conjunctive Query Containment and Constraint Satisfaction”. *Journal of Computer and System Sciences*, **61**(2):302–332, 2000.
- Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.
- Martin Lange and Carsten Lutz. “2-ExpTime lower bounds for propositional dynamic logics with intersection”. *Journal of Symbolic Logic*, **70**(4):1072–1086, 2005.
- Laks V. S. Lakshmanan, Ganesh Ramesh, Hui Wang, and Zheng Zhao. “On Testing Satisfiability of Tree Pattern Queries”. In *VLDB*, pages 120–131, 2004.
- Maarten Marx. “Conditional XPath, the First Order Complete XPath Dialect”. In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS’04)*, pages 13–22, 2004.
- Maarten Marx. “XPath with Conditional Axis Relations”. In *Proc. EDBT*, pages 477–494, 2004.
- Maarten Marx. “First order paths in ordered trees”. In *Proc. of the 10th International Conference on Database Theory (ICDT)*, 2005.
- Maarten Marx and Maarten de Rijke. “Semantic Characterizations of XPath”. In *TDM’04 Workshop on XML Databases and Information Retrieval*, Twente, The Netherlands, 2004.
- Albert R. Meyer. “Weak Monadic Second Order Theory of Successor is not Elementary-Recursive”. In *Logic Colloquium, Lecture Notes in Mathematics 453*, pages 132–154. Springer-Verlag, N.Y., 1975.
- Gerome Miklau and Dan Suciu. “Containment and Equivalence for an XPath Fragment”. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS’02)*, pages 65–76, Madison, Wisconsin, 2002.
- Frank Neven. “Automata Theory for XML Researchers”. *SIGMOD Record*, **31**(3), September 2002.
- Andreas Neumann and Helmut Seidl. “Locating Matches of Tree Patterns in Forests”. In *Proc. 18th FSTTCS, LNCS 1530*, pages 134–145, 1998.
- Frank Neven and Thomas Schwentick. “Query Automata on Finite Trees”. *Theoretical Computer Science*, **275**:633–674, 2002.
- Frank Neven and Thomas Schwentick. “XPath Containment in the Presence of Disjunction, DTDs, and Variables”. In *Proc. of the 9th International Conference on Database Theory (ICDT)*, pages 315–329, 2003.
- Frank Neven and Jan van den Bussche. “Expressiveness of Structured Document Query Languages Based on Attribute Grammars”. *Journal of the ACM*, **49**(1):56–100, January 2002.
- Dan Olteanu, Tobias Kiesling, and François Bry. “An Evaluation of Regular Path Expressions with Qualifiers against XML Streams”. In *Proceedings of 19th International Conference on Data*

- Engineering (ICDE)*, Bangalore, India, 5th - 8th March 2003. Full version in Technical Report PMS-FB-2002-12, Ludwig-Maximilians-Universität München, Munich, Germany, 2002.
- Dan Olteanu, Holger Meuss, Tim Furche, and Francois Bry. “XPath: Looking Forward”. In *Proc. EDBT Workshop on XML Data Management*, volume LNCS 2490, pages 109–127, Prague, Czech Republic, 2002. Springer-Verlag.
- Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
- Feng Peng and Sudarshan Chawathe. “XPath Queries on Streaming Data”. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD’03)*, 2003.
- Klaus Reinhardt. “The Complexity of Translating Logic to Finite Automata”. In E. Grädel, W. Thomas, and T. Wilke, editors, *Automata, Logics, and Infinite Games – A Guide to Current Research*. Springer-Verlag, LNCS 2500, 2002.
- Thomas Schwentick. Xpath query containment. *SIGMOD Record*, 33(1):101–109, 2004.
- G. Sur, J. Hammer, and J. Simeon. “An XQuery-Based Language for Processing Updates in XML”. In *PLAN-X*, 2004.
- Thomas Schwentick, Denis Thérien, and Heribert Vollmer. “Partially-ordered Two-way Automata: A New Characterization of DA”. In *Developments in Language Theory*, pages 239–250, 2001.
- I.H. Sudborough. “Time and Tape Bounded Auxiliary Pushdown Automata”. In *Mathematical Foundations of Computer Science (MFCS’77)*, pages 493–503. Springer Verlag, LNCS 53, 1977.
- Luc Segoufin and Victor Vianu. “Validating Streaming XML Documents”. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS’02)*, 2002.
- J.W. Thatcher and J.B. Wright. “Generalized Finite Automata Theory with an Application to a Decision Problem of Second-order Logic”. *Mathematical Systems Theory*, 2(1):57–81, 1968.
- Moshe Y. Vardi. “The Complexity of Relational Query Languages”. In *Proc. 14th Annual ACM Symposium on Theory of Computing (STOC’82)*, pages 137–146, San Francisco, CA USA, May 1982.
- H. Venkateswaran. “Properties that Characterize LOGCFL”. *Journal of Computer and System Sciences*, 43:380–404, 1991.
- Moshe Y. Vardi and Pierre Wolper. “Automata-theoretic techniques for Modal Logics of Programs”. *Journal of Computer and System Sciences*, 32:183–221, 1986.
- Philip Wadler. “A Formal Semantics of Patterns in XSLT”. In *Markup Technologies*, Philadelphia, December 1999. Revised version in *Markup Languages*, MIT Press, June 2001.
- Philip Wadler. “Two Semantics for XPath”, 2000. Draft paper available at <http://www.research.avayalabs.com/user/wadler/>.
- Peter T. Wood. Minimizing Simple XPath Expressions. In *Proc. of Intl. Workshop on the Web and Databases (WebDB)*, Santa Barbara, California, USA, May 2001.
- Peter T. Wood. “Containment for XPath Fragments under DTD constraints”. In *Proc. of the 9th International Conference on Database Theory (ICDT)*, pages 300–314, 2003.
- World Wide Web Consortium. XSL Transformations (XSLT). W3C Recommendation Version 1.0. <http://www.w3.org/TR/xslt>.
- World Wide Web Consortium. XML Path Language (XPath) Recommendation. <http://www.w3c.org/TR/xpath/>, November 1999.
- World Wide Web Consortium. “XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft (Aug. 16th 2002), 2002. <http://www.w3.org/TR/query-algebra/>.
- Mihalis Yannakakis. “Algorithms for Acyclic Database Schemes”. In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB’81)*, pages 82–94, Cannes, France, 1981.