A System for the Static Analysis of XPath

PIERRE GENEVÈS and NABIL LAYAÏDA
INRIA Rhône-Alpes

XPath is the standard language for navigating XML documents and returning a set of matching nodes. We present a sound and complete decision procedure for containment of XPath queries, as well as other related XPath decision problems such as satisfiability, equivalence, overlap, and coverage. The considered XPath fragment covers most of the language features used in practice. Specifically, we propose a unifying logic for XML, namely, the alternation-free modal μ-calculus with converse. We show how to translate major XML concepts such as XPath and regular XML types (including DTDs) into this logic. Based on these embeddings, we show how XPath decision problems, in the presence or absence of XML types, can be solved using a decision procedure for μ-calculus satisfiability. We provide a complexity analysis of our system together with practical experiments to illustrate the efficiency of the approach for realistic scenarios.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.2.4 [**Database Management**]: Systems—*Query processing*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*

General Terms: Algorithms, Languages, Standardization, Theory

Additional Key Words and Phrases: Containment, equivalence, logic, query, XML, XPath

1. INTRODUCTION

XPath [Clark and DeRose 1999] is the standard declarative language for querying an XML tree and returning a set of nodes. It is increasingly popular due to its expressive power and compact syntax. These advantages have given XPath a central role both in other key XML specifications and XML applications. It is used in XQuery as a core query language; in XSLT as a node selector in transformations; in XML Schema to define keys; and in XLink and XPointer to reference portions of XML data. XPath is also used in many applications, such as update languages [Sur et al. 2004] and XML access control [Fan et al. 2004].

Several XPath decision problems arise naturally in these use cases. The most basic decision problem for a query language is satisfiability [Benedikt et al. 2005]: whether or not an expression yields a nonempty result. XPath

Author's addresses: P. Genevès, N. Layaïda, INRIA Rhône-Alpes, 655 Avenue de l'Europe, Montbonnot, 38334 St Ismier Cedex, France; email: Pierre.geneves@inria.fr.

satisfiability is important for optimization of host language implementations: For instance, if we can decide at compile-time that a query is not satisfiable, then subsequent bound computations can be avoided. Another basic decision problem is that of XPath equivalence: whether or not two queries always return the same result. It is important for reformulation and optimization of the query itself [Genevès and Vion-Dury 2004a], which aim at enforcing operational properties, while preserving semantic equivalence [Abiteboul and Vianu 1999; Levin and Pierce 2005]. These two decision problems are reducible to XPath containment: whether or not, for any tree, the result of a particular query is included in the result of another. Query containment is itself critical for the static analysis of XML specifications and especially for type-checking transformations [Martens and Neven 2004; Tozawa 2001]. Such applications introduce XPath decision problems in the presence of XML types such as DTDs [Bray et al. 2004] or XML schemas [Fallside and Walmsley 2004]. Other decision problems needed in applications include, for example, XPath overlap (whether two expressions select common nodes) and coverage (whether nodes selected by an expression are always contained in the union of the results selected by several other expressions).

In the literature, much attention has been paid to classifying the containment problem for simple XPath subfragments in complexity classes. This allowed the identification of subsets of XPath for which the containment can be efficiently decided.

In this article, our goal is to describe sound, complete, and efficient decision procedures that are usable in practice for a large XPath fragment in the presence or absence of XML types. We first briefly introduce the XPath language, and present the approach and outline of the article.

1.1 Introduction to XPath

XPath [Clark and DeRose 1999] has been introduced by the W3C as the standard query language for retrieving information in XML documents. It allows navigation in XML trees and returning a set of matching nodes. In their simplest form, XPath expressions look like "directory navigation paths." For example, the XPath

/book/chapter/section

navigates from the root of a document (designated by the leading slash "/") through the top-level "book" element to its "chapter" child elements and on to its "section" child elements. The result of the evaluation of the entire expression is the set of all the "section" elements that can be reached in this manner, returned in the order they occurred in the document. At each step in the navigation, the selected nodes for this step can be filtered using qualifiers. A qualifier is a Boolean expression between brackets that can test path existence. So, if we ask for

/book/chapter/section[citation]

then the result is *all* the "section" elements that have a least one child element named "citation." The situation becomes more interesting when combined with

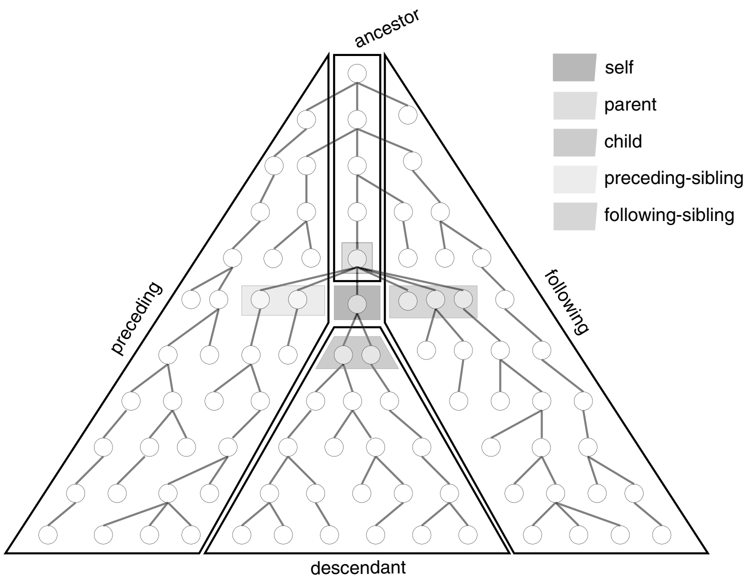


Fig. 1. XPath axes partition from context node.

XPath's capability of searching along "axes" other than the shown "children of" axis. Indeed, the aforementioned XPath is a shorthand for

/child::book/child::chapter/child::section[child::citation]

where it is made explicit that each *path step* is meant to search the "child" axis containing all children of the previous context node. If we instead asked for

/child::book/descendant::*[child::citation]

then the last step selects nodes of any kind that are among the descendants of the top element "book" and have a "citation" subelement.

Previous examples are all *absolute* XPath expressions. The meaning of a *relative* expression (without the leading "/") is defined with respect to a context node in the tree. A key to XPath success is its compactness due to the powerful navigation made possible by the various axes. Starting from a particular context node in a tree, every other node can be reached. Axes define a partitioning of a tree from any context node. Figure 1 illustrates this on a sample tree. More informal details on the complete XPath standard can be found in the W3C specification [Clark and DeRose 1999].

A variety of factors contribute to the complexity of XPath decision problems, such as the operators allowed in XPath queries and their combination. We present here the common distinctions between XPath fragments found in the literature, taken from Benedikt et al. [2005]:

—positive vs. nonpositive: depending whether the negation operator is considered inside qualifiers;

—downward vs. upward: depending whether queries specify downward or upward traversal of the tree, or both;

—recursive vs. nonrecursive: depending whether XPath transitive closure axes (e.g., "descendant" or "ancestor") are considered;
—qualified vs. nonqualified: depending whether queries allow filtering qualifiers; and
—with vs. without data values: depending whether comparisons of data values expressing joins are allowed.

From the results of Benedikt et al. [2005] and Schwentick [2004], we know that the combination of some previous factors with data values may lead to undecidability of decision problems such as containment. In the remaining part of the article, we focus on a large XPath fragment covering all previous factors, except data values. This fragment, detailed in Section 3, is the largest considered so far in the literature, and covers most features of XPath 1.0.[1]

1.2 Approach and Outline

In this article, we propose alternation-free modal μ-calculus with converse as the appropriate logic for effectively solving XPath decision problems in the presence or absence of XML types. We show how XPath can be linearly translated into μ-calculus. We also show how to embed regular tree types (including DTDs) in the μ-calculus. We express XPath decision problems (containment, satisfiability, equivalence, overlap, coverage) as formulae in this logic. We use an EXPTIME decision procedure for μ-calculus satisfiability to solve the generated formula and construct relevant example and/or counter-example XML trees. We provide experimental results which shed light, for the first time, on the cost of solving XML decision problems in practice. The system has been fully implemented [Genevès and Layaïda 2006] and can be used for the static analysis of XML specifications.

The remaining part of the article is organized as follows: in Section 2 we introduce the logic we propose for reasoning on XML trees; in Section 3 we describe the translation of XPath queries into this logic; Section 4 embeds regular XML types into the logic. Based on these translations, Section 5 explains how to formulate and solve the considered decision problems. We present an implementation of the system, along with practical experiments, in Section 6, before discussing related work in Section 7 and concluding in Section 8.

2. A LOGIC FOR XML

In this section we introduce a specific variant of the modal μ-calculus as a formalism for reasoning on XML trees.

2.1 XML Documents and Finite Binary Trees

We consider an XML document as a finite ordered and labeled tree of unbounded depth and arity. Since there is no a priori bound on the number of children

[1]The fragment also includes two extensions from the forthcoming XPath 2.0 [Berglund et al. 2006] language: qualified paths (e.g., $(p)[q]$), instead of XPath 1.0 qualified steps (e.g., $a :: n[q]$), and path intersection ($p_1 \cap p_2$).

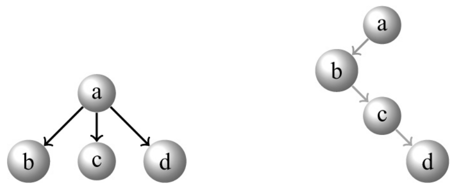


Fig. 2. Unranked and binary tree representations.

of a node, such a tree is therefore *unranked* [Neven 2002b]. Tree nodes are labeled with symbols taken from a countably infinite alphabet Σ. There is a straightforward isomorphism between sequences of unranked and binary trees [Hosoya et al. 2005; Neven 2002a]. In order to describe this, we first define the set $\mathcal{T}^n_\Sigma$ of unranked trees:

$$\mathcal{T}^n_\Sigma \ni t ::= \sigma(h),$$

where $\sigma \in \Sigma$ and h is a hedge, that is, a sequence of unranked trees, defined as follows:

$$\mathcal{H}_\Sigma \ni h ::= \sigma(h), h' | ()$$

A binary tree t is either a σ-labeled root of two subtrees ($\sigma \in \Sigma$) or the empty tree:

$$\mathcal{T}^2_\Sigma \ni t ::= \sigma(t, t') | \epsilon$$

Unranked trees can be translated into binary trees with the following function:

$$\begin{aligned} \beta(\cdot) &: \mathcal{H}_\Sigma \to \mathcal{T}^2_\Sigma \\ \beta(\sigma(h), h') &= \sigma(\beta(h), \beta(h')) \\ \beta(()) &= \epsilon \end{aligned}$$

The inverse translation function converts a binary tree into a sequence of unranked trees:

$$\begin{aligned} \beta^{-1}(\cdot) &: \mathcal{T}^2_\Sigma \to \mathcal{H}_\Sigma \\ \beta^{-1}(\sigma(t, t')) &= \sigma(\beta^{-1}(t)), \beta^{-1}(t') \\ \beta^{-1}(\epsilon) &= () \end{aligned}$$

For example, Figure 2 illustrates how a sample unranked tree is mapped to its binary representation and vice versa.

Note that the translation of a single unranked tree results in a binary tree of the form $\sigma(t, \epsilon)$. Reciprocally, the inverse translation of such a binary tree always yields a single unranked tree. When modeling XML, we therefore restrict our attention to binary trees of the form $\sigma(t, \epsilon)$, without loss of generality.

We now introduce the logic we propose for reasoning over these structures.

2.2 The μ-Calculus

The *propositional μ-calculus* is a propositional modal logic extended with least and greatest fixpoint operators [Kozen 1983]. A *signature* Ξ for the μ-calculus

$$\begin{array}{lcl}
[\![\cdot]\!]_V^K & : & \mathcal{L}_\mu \longrightarrow 2^W \\
[\![\top]\!]_V^K & = & W \\
[\![\bot]\!]_V^K & = & \emptyset \\
[\![p]\!]_V^K & = & L(p) \\
[\![\neg\varphi]\!]_V^K & = & W \setminus [\![\varphi]\!]_V^K \\
[\![\varphi_1 \vee \varphi_2]\!]_V^K & = & [\![\varphi_1]\!]_V^K \cup [\![\varphi_2]\!]_V^K \\
[\![\varphi_1 \wedge \varphi_2]\!]_V^K & = & [\![\varphi_1]\!]_V^K \cap [\![\varphi_2]\!]_V^K \\
[\![[\alpha]\,\varphi]\!]_V^K & = & \{w : \forall w'(w,w') \in R(\alpha) \Rightarrow w' \in [\![\varphi]\!]_V^K\} \\
[\![\langle\alpha\rangle\,\varphi]\!]_V^K & = & \{w : \exists w'(w,w') \in R(\alpha) \wedge w' \in [\![\varphi]\!]_V^K\} \\
[\![\mu X.\varphi]\!]_V^K & = & \bigcap\{W' \subseteq W : [\![\varphi]\!]_{V[X/W']}^K \subseteq W'\} \\
[\![\nu X.\varphi]\!]_V^K & = & \bigcup\{W' \subseteq W : [\![\varphi]\!]_{V[X/W']}^K \supseteq W'\} \\
[\![X]\!]_V^K & = & V(X)
\end{array}$$

Fig. 3. Semantics of the μ-calculus.

consists of a set *Prop* of atomic propositions, a set Var of propositional variables, and a set *FProg* of atomic programs. In the XML context, atomic propositions represent the symbols of the alphabet Σ used to label XML trees. Atomic programs allow navigation in trees.

The μ-calculus with converse[2] [Vardi 1998] augments propositional μ-calculus by associating with each atomic program a its converse $\bar{a}$. A *program* α is either an atomic program or its converse. We note *Prog* the set $FProg \cup \{\bar{a} \mid a \in FProg\}$. This is the only difference between the propositional μ-calculus that lacks converse programs. It is important to note that the addition of converse programs preserves the EXPTIME upper bound for the satisfiability problem [Vardi 1998].

The set $\mathcal{L}_\mu$ of formulae of the μ-calculus with converse over the signature Ξ is defined as follows:

$$\begin{array}{rl}\mathcal{L}_\mu \ni \varphi ::= & \top \mid \bot \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \\ & [\alpha]\,\varphi \mid \langle\alpha\rangle\,\varphi \mid X \mid \mu X.\varphi \mid \nu X.\varphi \end{array},$$

where $p \in Prop$, $X \in Var$, and α is a program. Note that X should not occur negatively in $\mu X.\varphi$ and in $\nu X.\varphi$.

The semantics of the full μ-calculus is given with respect to a *Kripke structure* $K = \langle W, R, L\rangle$, where W is a set of nodes, $R : Prog \to 2^{W \times W}$ assigns to each atomic program a transition relation over W, and L is an interpretation function that assigns to each atomic proposition a set of nodes.

The formal semantics function $[\cdot]_V^K$ shown in Figure 3 defines the semantics of a μ-calculus formula in terms of a Kripke structure K and valuation V. A valuation $V : Var \to 2^W$ maps each variable to a subset of W. For a valuation V, variable X, and set of nodes $W' \subseteq W$, $V[X/W']$ denotes the valuation that is obtained from V by assigning W' to X.

Note that if φ is a sentence (i.e., all propositional variables occurring in φ are bound), then no valuation is required. For a node $w \in W$ and sentence φ, we say that φ holds at w in K, denoted $K, w \models \varphi$ iff $w \in [\varphi]^K$.

The two modalities $\langle a\rangle\varphi$ (possibility) and $[a]\,\varphi$ (necessity) are operators for navigating the structure.

[2]The μ-calculus with converse is also known as the *full* μ-calculus, or alternatively, as the *two-way* μ-calculus in the literature.

$$\begin{array}{lcl}
\neg\,[\alpha]\,\varphi & = & \langle\alpha\rangle\,\neg\varphi \\
\neg\,\langle\alpha\rangle\,\varphi & = & [\alpha]\,\neg\varphi \\
\neg\mu X.\varphi & = & \nu X.\neg\varphi[X/\neg X] \\
\neg\nu X.\varphi & = & \mu X.\neg\varphi[X/\neg X] \\
\neg(\varphi_1 \wedge \varphi_2) & = & \neg\varphi_1 \vee \neg\varphi_2 \\
\neg(\varphi_1 \vee \varphi_2) & = & \neg\varphi_1 \wedge \neg\varphi_2 \\
\neg\neg\varphi & = & \varphi
\end{array}$$

Fig. 4. Dualities for negation normal form.

The syntax of $\mathcal{L}_\mu$ formulae as given previously is in fact redundant. Actually, we only have to deal with a subset of $\mathcal{L}_\mu$ composed of formulae in negation normal form. We say that a formula is in *negation normal form* if and only if all negations in the formula appear only before atomic propositions. Every formula is equivalent to a formula in negation normal form [Kozen 1983], which can be obtained by expanding negations using De Morgan's rules, together with standard dualities for modalities and fixpoints (see Figure 4). For readability purposes, the translations of XPath expressions given in Section 3 are not given in negation normal form.

For reasoning on XML trees, we are in fact interested in a specific subset of $\mathcal{L}_\mu$, namely, the alternation-free modal μ-calculus with converse over finite binary trees.

We recall that a $\mathcal{L}_\mu$ formula φ in negation normal form is alternation-free whenever the following condition holds[3]: if $\mu X.\varphi_1$ (respectively, $\nu X.\varphi_1$) is a subformula of φ and $\nu Y.\varphi_2$ (respectively, $\mu Y.\varphi_2$) is a subformula of φ_1, then X does not occur freely in φ_2.

The following section now introduces the additional restrictions of $\mathcal{L}_\mu$ related to finite binary trees.

2.3 XML Constraints on Kripke Structures

In this section, we restrict the satisfiability problem of $\mathcal{L}_\mu$ over Kripke structures to the satisfiability problem over finite binary trees.

The propositional μ-calculus has the *finite tree model property*: A formula that is satisfiable is also satisfiable on a finite tree [Kozen 1988]. Unfortunately, the introduction of converse programs causes the loss of the finite model property [Vardi 1998]. Therefore, we need to reinforce the finite model property and introduce some others to ensure that we work on finite binary trees encoding XML structures.

First, each XML node has, at most, one Σ-label, that is, $p \wedge p'$ never holds for distinct atomic propositions p and p'. This can be easily incorporated in a μ-calculus satisfiability solver.

Second, for navigating binary trees, we only use two atomic programs 1 and 2, and their associated relations $R(1) = \prec_{\text{fc}}$ and $R(2) = \prec_{\text{ns}}$, whose meaning is to respectively connect a node to its left child and right child. For any $(x, y) \in W \times W$, $x \prec_{\text{fc}} y$ holds iff y is the left child of x (i.e., the first child in the unranked tree representation) and $x \prec_{\text{ns}} y$ holds iff y is the right child of

[3]For instance, $\nu X.(\mu Y.\langle 1\rangle Y \wedge p) \vee \langle 2\rangle X$ is alternation-free, but $\nu X.(\mu Y.\langle 1\rangle Y \wedge X) \vee p$ is not, since X bound by ν appears freely in the scope of μY.

x in the binary tree representation (i.e., the next sibling in the unranked tree representation).

For each atomic program $a \in \{1, 2\}$, we define $R(\overline{a})$ to be the relational inverse of $R(a)$, that is, $R(\overline{a}) = \{(v, u) : (u, v) \in R(a)\}$. We thus consider programs $\alpha \in \{1, 2, \overline{1}, \overline{2}\}$ inside modalities for navigating downward and upward in trees.

We now define restrictions for a Kripke structure to form a finite binary tree. A Kripke structure $t = \langle W, R, L \rangle$ is a finite binary tree if it satisfies the following conditions:

(1) W is finite;
(2) the set of nodes W, together with the accessibility relation $\prec_{\mathrm{fc}} \cup \prec_{\mathrm{ns}}$, define a tree; and
(3) $\prec_{\mathrm{fc}}$ and $\prec_{\mathrm{ns}}$ are partial functions, that is, for all $m \in W$ and $j \in \{1, 2\}$, there is at most one $m_j \in W$ such that $(m, m_j) \in R(j)$.

We say that a finite binary tree $t = \langle W, R, L \rangle$ satisfies φ if $t, r \models \varphi$, where $r \in W$ is the root of the tree t.

For accessing the root, we use the $\mathcal{L}_\mu$ formula

$$\varphi_{\mathrm{root}} = \left[\overline{1}\right] \bot \wedge \left[\overline{2}\right] \bot,$$

which selects a node, provided it has no parent.

For ensuring finiteness, we rely on König's lemma, which states that *a finitely branching infinite tree has some infinite path*, or in other words, a finitely branching tree in which every branch is finite is finite. The expression $\nu X. \langle 1 \rangle X \vee \langle 2 \rangle X$ is only satisfied by structures containing infinite or cyclic paths. To prevent the existence of such paths, we negate the previous formula and, by propagating negation using the rules presented in Figure 4, we obtain:

$$\varphi_{\mathrm{ft}} = \mu X. [1] X \wedge [2] X$$

Here, φ_{ft} states that all descending branches are finite from the current context node (φ_{ft} is vacuously satisfied at the leaves). In our case, we need φ_{ft} to hold at the root (i.e., $\varphi_{\mathrm{root}} \wedge \varphi_{\mathrm{ft}}$ must hold) in order to ensure that we work with a finite structure. This is for condition (1) to be satisfied.

We still need to enforce (2) and (3). We do this by rewriting existential modalities such that if a successor is supposed to exist, then there exists at least one, and if there are many, all verify the same property. This is a way to overcome the difficulty that in μ-calculus, we cannot naturally express a property like "a node has exactly n successors." Technically, we denote by φ^{FBT} the formula φ where all occurrences of $\langle \alpha \rangle \psi$ are replaced by $\langle \alpha \rangle \top \wedge [\alpha] \psi^{\mathrm{FBT}}$. This replacement is enough to enforce conditions (2) and (3).

PROPOSITION 2.1. *A $\mathcal{L}_\mu$ formula φ is satisfied by a finite binary tree model if and only if the formula $\varphi_{\mathrm{root}} \wedge \varphi_{\mathrm{ft}} \wedge \varphi^{FBT}$ is satisfied by a Kripke structure.*

The proof of the "if" part iteratively constructs a tree model and proceeds by induction on the structure on φ. The "only if" part is almost immediate [Tanabe et al. 2005].

Proposition 2.1 gives the adequate framework for formulating decision problems on XML structures in terms of a μ-calculus formula.

3. LOGICAL INTERPRETATION OF XPATH QUERIES

We consider a large and realistic XPath fragment which includes negation, both downward and upward navigation, recursion, union, intersection, and qualifiers. The abstract syntax of our XPath fragment is shown next:

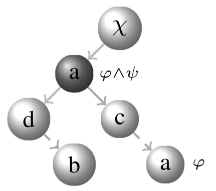
$$\begin{array}{lll}
\mathcal{L}_{\text{XPath}} & e & ::= \; /p \mid p \mid e_1 \shortmid e_2 \mid e_1 \cap e_2 \\
\textit{Path} & p & ::= \; p_1/p_2 \mid p[q] \mid a::n \\
\textit{Qualifier} & q & ::= \; q \text{ and } q \mid q \text{ or } q \mid \text{not } q \mid p \\
\textit{Axis} & a & ::= \; \text{child} \mid \text{descendant} \mid \text{self} \mid \text{parent} \mid \text{ancestor} \mid \text{following} \\
 & & \quad \mid \text{preceding} \mid \text{descendant-or-self} \mid \text{ancestor-or-self} \\
 & & \quad \mid \text{preceding-sibling} \mid \text{following-sibling} \\
\textit{NodeTest} & n & ::= \; \sigma \mid *
\end{array}$$

XPath expressions are directly used for querying unranked XML trees. We first recall XPath denotational semantics over unranked trees [Wadler 2000]. The evaluation of an XPath query returns a set of nodes that is reachable from a context node x in a tree t. The formal semantics functions $\mathcal{S}_e$ and $\mathcal{S}_p$ define the set of nodes respectively returned by expressions and paths:

$$\begin{array}{lll}
\mathcal{S}_e[\![\cdot]\!]^{\cdot}_{\cdot} & : & \mathcal{L}_{\text{XPath}} \times \textit{Node} \times \mathcal{T}^n_\Sigma \longrightarrow \text{Set}(\textit{Node}) \\
\mathcal{S}_e[\![/p]\!]^t_x & = & \mathcal{S}_p[\![p]\!]^t_{root()} \\
\mathcal{S}_e[\![p]\!]^t_x & = & \mathcal{S}_p[\![p]\!]^t_x \\
\mathcal{S}_e[\![e_1 \shortmid e_2]\!]^t_x & = & \mathcal{S}_e[\![e_1]\!]^t_x \cup \mathcal{S}_e[\![e_2]\!]^t_x \\
\mathcal{S}_e[\![e_1 \cap e_2]\!]^t_x & = & \mathcal{S}_e[\![e_1]\!]^t_x \cap \mathcal{S}_e[\![e_2]\!]^t_x \\
\mathcal{S}_p[\![\cdot]\!]^{\cdot}_{\cdot} & : & \textit{Path} \times \textit{Node} \times \mathcal{T}^n_\Sigma \longrightarrow \text{Set}(\textit{Node}) \\
\mathcal{S}_p[\![p_1/p_2]\!]^t_x & = & \{x_2 \mid x_1 \in \mathcal{S}_p[\![p_1]\!]^t_x \wedge x_2 \in \mathcal{S}_p[\![p_2]\!]^t_{x_1}\} \\
\mathcal{S}_p[\![p[q]]\!]^t_x & = & \{x_1 \mid x_1 \in \mathcal{S}_p[\![p]\!]^t_x \wedge \mathcal{S}_q[\![q]\!]^t_{x_1}\} \\
\mathcal{S}_p[\![\text{a}::\sigma]\!]^t_x & = & \{x_1 \mid x_1 \in \mathcal{S}_a[\![\text{a}]\!]^t_x \wedge \textit{name}(x_1) = \sigma\} \\
\mathcal{S}_p[\![\text{a}::*]\!]^t_x & = & \{x_1 \mid x_1 \in \mathcal{S}_a[\![\text{a}]\!]^t_x\}
\end{array}$$

The function $\mathcal{S}_q$ defines the semantics of qualifiers that basically state the existence (or absence) of one or more paths from a context node x:

$$\begin{array}{lll}
\mathcal{S}_q[\![\cdot]\!]^{\cdot}_{\cdot} & : & \textit{Qualifier} \times \textit{Node} \times \mathcal{T}^n_\Sigma \longrightarrow \textit{Boolean} \\
\mathcal{S}_q[\![q_1 \text{ and } q_2]\!]^t_x & = & \mathcal{S}_q[\![q_1]\!]^t_x \wedge \mathcal{S}_q[\![q_2]\!]^t_x \\
\mathcal{S}_q[\![q_1 \text{ or } q_2]\!]^t_x & = & \mathcal{S}_q[\![q_1]\!]^t_x \vee \mathcal{S}_q[\![q_2]\!]^t_x \\
\mathcal{S}_q[\![\text{not } q]\!]^t_x & = & \neg\, \mathcal{S}_q[\![q]\!]^t_x \\
\mathcal{S}_q[\![p]\!]^t_x & = & \mathcal{S}_p[\![p]\!]^t_x \neq \emptyset
\end{array}$$



Translated Query: child::a[child::b]

$$\underbrace{a \wedge (\mu X. \langle \bar{1} \rangle \chi \vee \langle \bar{2} \rangle X)}_{\varphi} \wedge \underbrace{\langle 1 \rangle \mu Y. b \vee \langle 2 \rangle Y}_{\psi}$$

Fig. 5. XPath translation example.

Eventually, the function $\mathcal{S}_a$ gives the denotational semantics of axes:

$$\begin{array}{ll}
\mathcal{S}_a[\![\cdot]\!]^{\cdot}_{\cdot} & : \ Axis \times Node \times \mathcal{T}^n_\Sigma \longrightarrow \text{Set}(Node) \\
\mathcal{S}_a[\![\text{child}]\!]^t_x & = children(x) \\
\mathcal{S}_a[\![\text{parent}]\!]^t_x & = parent(x) \\
\mathcal{S}_a[\![\text{descendant}]\!]^t_x & = children^+(x) \\
\mathcal{S}_a[\![\text{ancestor}]\!]^t_x & = parent^+(x) \\
\mathcal{S}_a[\![\text{self}]\!]^t_x & = \{x\} \\
\mathcal{S}_a[\![\text{descendant-or-self}]\!]^t_x & = \mathcal{S}_a[\![\text{descendant}]\!]^t_x \cup \mathcal{S}_a[\![\text{self}]\!]^t_x \\
\mathcal{S}_a[\![\text{ancestor-or-self}]\!]^t_x & = \mathcal{S}_a[\![\text{ancestor}]\!]^t_x \cup \mathcal{S}_a[\![\text{self}]\!]^t_x \\
\mathcal{S}_a[\![\text{preceding}]\!]^t_x & = \{y \mid y \ll x\} \setminus \mathcal{S}_a[\![\text{ancestor}]\!]^t_x \\
\mathcal{S}_a[\![\text{following}]\!]^t_x & = \{y \mid x \ll y\} \setminus \mathcal{S}_a[\![\text{descendant}]\!]^t_x \\
\mathcal{S}_a[\![\text{following-sibling}]\!]^t_x & = \{y \mid y \in child(parent(x)) \wedge x \ll y\} \\
\mathcal{S}_a[\![\text{preceding-sibling}]\!]^t_x & = \{y \mid y \in child(parent(x)) \wedge y \ll x\}
\end{array}$$

in which *root*(), *children*(x), and *parent*(x) are primitives for navigating unranked trees, $\ll$ is the ordering relation ($x \ll y$ holds if and only if the node x is before the node y in depth-first traversal order of the tree), and *name*() is the mean for accessing the labeling of the tree.

3.1 A Translation into the μ-Calculus

We now explain how an XPath expression can be translated into an equivalent formula in $\mathcal{L}_\mu$ over binary trees. The translation adheres to XPath formal semantics in the sense that the translated formula holds for nodes which are selected by the XPath query. Navigation, as performed by XPath, in unranked trees is thus translated in terms of navigation in the binary tree representation.

For example, Figure 5 gives the intuition of the translation of the XPath expression "child::a[child::b]." In an unranked tree, this expression selects all "a" child nodes of a given context which have at least one "b" child. The translated $\mathcal{L}_\mu$ formula holds for "a" nodes, which are selected by the expression. By navigating upward in the binary tree from these nodes (specifically, any number of steps upward from a right child, and then once upward from a left child), we must reach the initial context. Then, starting back from the candidate "a" nodes, we must navigate downward in the tree in order to verify the existence of a "b" child.

$$\begin{array}{lcl}
A^{\rightarrow}[\![\cdot]\!](\cdot) & : & \mathit{Axis} \times \mathcal{L}_\mu \longrightarrow \mathcal{L}_\mu \\
A^{\rightarrow}[\![\text{self}]\!](\chi) & = & \chi \\
A^{\rightarrow}[\![\text{following-sibling}]\!](\chi) & = & \mu Z. \langle \overline{2} \rangle \chi \vee \langle \overline{2} \rangle Z \\
A^{\rightarrow}[\![\text{child}]\!](\chi) & = & \mu Z. \langle \overline{1} \rangle \chi \vee \langle \overline{2} \rangle Z \\
A^{\rightarrow}[\![\text{descendant}]\!](\chi) & = & \mu Z. \langle \overline{1} \rangle (\chi \vee Z) \vee \langle \overline{2} \rangle Z \\
A^{\rightarrow}[\![\text{descendant-or-self}]\!](\chi) & = & \mu Z.\chi \vee \mu Y. \langle \overline{1} \rangle (Y \vee Z) \vee \langle \overline{2} \rangle Y \\
A^{\rightarrow}[\![\text{parent}]\!](\chi) & = & \langle 1 \rangle \mu Z.\chi \vee \langle 2 \rangle Z \\
A^{\rightarrow}[\![\text{ancestor}]\!](\chi) & = & \langle 1 \rangle \mu Z.\chi \vee \langle 1 \rangle Z \vee \langle 2 \rangle Z \\
A^{\rightarrow}[\![\text{ancestor-or-self}]\!](\chi) & = & \mu Z.\chi \vee \langle 1 \rangle \mu Y.Z \vee \langle 2 \rangle Y \\
A^{\rightarrow}[\![\text{preceding-sibling}]\!](\chi) & = & \mu Z. \langle 2 \rangle \chi \vee \langle 2 \rangle Z \\
A^{\rightarrow}[\![\text{following}]\!](\chi) & = & A^{\rightarrow}[\![\text{descendant-or-self}]\!](A^{\rightarrow}[\![\text{following-sibling}]\!](\eta)) \\
A^{\rightarrow}[\![\text{preceding}]\!](\chi) & = & A^{\rightarrow}[\![\text{descendant-or-self}]\!](A^{\rightarrow}[\![\text{preceding-sibling}]\!](\eta)) \\
 & & \text{where } \eta \text{ is a shorthand for } A^{\rightarrow}[\![\text{ancestor-or-self}]\!](\chi)
\end{array}$$

Fig. 6. Translation of XPath axes.

XPath	μ-Calculus	CPDL
π/following-sibling::$*$	$\mu Z. \langle \overline{2} \rangle \pi \vee \langle \overline{2} \rangle Z$	$\langle \overline{2}^* \cdot \overline{2} \rangle \pi$
π/child::$*$	$\mu Z. \langle \overline{1} \rangle \pi \vee \langle \overline{2} \rangle Z$	$\langle \overline{2}^* \cdot \overline{1} \rangle \pi$
π/descendant::$*$	$\mu Z. \langle \overline{1} \rangle (\pi \vee Z) \vee \langle \overline{2} \rangle Z$	$\langle (\overline{1} \mid \overline{2})^* \cdot \overline{1} \rangle \pi$
π/descendant-or-self::$*$	$\mu Z.\pi \vee \mu Y. \langle \overline{1} \rangle (Y \vee Z) \vee \langle \overline{2} \rangle Y$	$\langle \text{nil} \mid (\overline{1} \mid \overline{2})^* \cdot \overline{1} \rangle \pi$
π/parent::$*$	$\langle 1 \rangle \mu Z.\pi \vee \langle 2 \rangle Z$	$\langle 1 \cdot 2^* \rangle \pi$
π/ancestor::$*$	$\langle 1 \rangle \mu Z.\pi \vee \langle 1 \rangle Z \vee \langle 2 \rangle Z$	$\langle 1 \cdot (1 \mid 2)^* \rangle \pi$
π/ancestor-or-self::$*$	$\mu Z.\pi \vee \langle 1 \rangle \mu Y.Z \vee \langle 2 \rangle Y$	$\langle \text{nil} \mid 1 \cdot (1 \mid 2)^* \rangle \pi$
π/preceding-sibling::$*$	$\mu Z. \langle 2 \rangle \pi \vee \langle 2 \rangle Z$	$\langle 2^* \cdot 2 \rangle \pi$

Fig. 7. Logical correspondences in terms of the early CPDL operators.

Note that without converse programs, we would have been unable to differentiate selected nodes from nodes whose existence is tested. This is because we must state properties on both the ancestors and descendants of the selected node. Therefore, equipping the $\mathcal{L}_\mu$ logic with converse programs is crucial for supporting XPath.[4] Logics without converse programs may only be used for solving XPath satisfiability, but not other decision problems, such as containment.

3.1.1 *Logical Interpretation of Axes.* We first translate navigational primitives, namely, XPath axes. The translation is formally specified in Figure 6 as a translation function noted "$A^{\rightarrow}[\![\cdot]\!](\cdot)$" which takes an XPath axis as input, and returns its translation in μ-calculus in terms of the μ-calculus formula given as a parameter to allow further composition. $A^{\rightarrow}[\![a]\!](\chi)$ holds for all nodes that can be accessed through the axis a from some node verifying χ. The formal parameter χ allows us to express the composition of formulae needed for translating path composition.

For readers more familiar with PDL and CPDL (PDL with converse programs), both defined in Fischer and Ladner [1979], we give a correspondence of notations in Figure 7.

[4]We may ask whether it is possible to eliminate upward navigation at the XPath level. It is well-known that such XPath rewritings may cause exponential blow-ups of expression sizes.

$$
\begin{array}{lcl}
E^{\rightarrow}[\![\cdot]\!](\cdot) & : & \mathcal{L}_{\text{XPath}} \times \mathcal{L}_{\mu} \longrightarrow \mathcal{L}_{\mu} \\
E^{\rightarrow}[\![/p]\!](\chi) & = & P^{\rightarrow}[\![p]\!](\boxed{\overline{1}}\,\bot \wedge \boxed{\overline{2}}\,\bot) \\
E^{\rightarrow}[\![p]\!](\chi) & = & P^{\rightarrow}[\![p]\!](\chi) \\
E^{\rightarrow}[\![e_1 \mid e_2]\!](\chi) & = & E^{\rightarrow}[\![e_1]\!](\chi) \vee E^{\rightarrow}[\![e_2]\!](\chi) \\
E^{\rightarrow}[\![e_1 \cap e_2]\!](\chi) & = & E^{\rightarrow}[\![e_1]\!](\chi) \wedge E^{\rightarrow}[\![e_2]\!](\chi)
\end{array}
$$

Fig. 8. Translation of expressions.

$$
\begin{array}{lcl}
P^{\rightarrow}[\![\cdot]\!](\cdot) & : & \mathit{Path} \times \mathcal{L}_{\mu} \longrightarrow \mathcal{L}_{\mu} \\
P^{\rightarrow}[\![p_1/p_2]\!](\chi) & = & P^{\rightarrow}[\![p_2]\!](P^{\rightarrow}[\![p_1]\!](\chi)) \\
P^{\rightarrow}[\![p[q]]\!](\chi) & = & P^{\rightarrow}[\![p]\!](\chi) \wedge Q^{\leftarrow}[\![q]\!](\top) \\
P^{\rightarrow}[\![a{::}\sigma]\!](\chi) & = & A^{\rightarrow}[\![a]\!](\chi) \wedge \sigma \\
P^{\rightarrow}[\![a{::}*]\!](\chi) & = & A^{\rightarrow}[\![a]\!](\chi)
\end{array}
$$

Fig. 9. Translation of paths.

$$
\begin{array}{lcl}
Q^{\leftarrow}[\![\cdot]\!](\cdot) & : & \mathit{Qualifier} \times \mathcal{L}_{\mu} \longrightarrow \mathcal{L}_{\mu} \\
Q^{\leftarrow}[\![q_1 \text{ and } q_2]\!](\chi) & = & Q^{\leftarrow}[\![q_1]\!](\chi) \wedge Q^{\leftarrow}[\![q_2]\!](\chi) \\
Q^{\leftarrow}[\![q_1 \text{ or } q_2]\!](\chi) & = & Q^{\leftarrow}[\![q_1]\!](\chi) \vee Q^{\leftarrow}[\![q_2]\!](\chi) \\
Q^{\leftarrow}[\![\text{not } q]\!](\chi) & = & \neg\, Q^{\leftarrow}[\![q]\!](\chi) \\
Q^{\leftarrow}[\![p]\!](\chi) & = & P^{\leftarrow}[\![p]\!](\chi) \\
 & & \\
P^{\leftarrow}[\![\cdot]\!](\cdot) & : & \mathit{Path} \times \mathcal{L}_{\mu} \longrightarrow \mathcal{L}_{\mu} \\
P^{\leftarrow}[\![p_1/p_2]\!](\chi) & = & P^{\leftarrow}[\![p_1]\!](P^{\leftarrow}[\![p_2]\!](\chi)) \\
P^{\leftarrow}[\![p[q]]\!](\chi) & = & P^{\leftarrow}[\![p]\!](\chi \wedge Q^{\leftarrow}[\![q]\!](\top)) \\
P^{\leftarrow}[\![a{::}\sigma]\!](\chi) & = & A^{\leftarrow}[\![a]\!](\chi \wedge \sigma) \\
P^{\leftarrow}[\![a{::}*]\!](\chi) & = & A^{\leftarrow}[\![a]\!](\chi)
\end{array}
$$

Fig. 10. Translation of qualifiers.

3.1.2 *Logical Interpretation of Expressions.* The translation of XPath expressions into μ-calculus is given in Figure 8. This is formally expressed as a translation function noted "$E^{\rightarrow}[\![\cdot]\!](\cdot)$" which takes an XPath expression as input, and a μ-calculus formula as a parameter that indicates the context from which the expression is applied. Absolute XPath expressions are interpreted from the root (selected by the μ-calculus expression φ_{root}), whereas relative expressions are interpreted relative to any context node. We use a fresh atomic proposition named φ_{context} for distinguishing context nodes.

The translation of expressions relies on the translations of paths shown in Figure 9. XPath's most essential construct p_1/p_2 translates into formula composition in $\mathcal{L}_{\mu}$ such that the resulting formula holds for all nodes accessed through p_2 from those nodes accessed from χ by p_1.

The translation of the branching construct $p[q]$ significantly differs. The resulting formula must hold for all nodes that can be accessed through p and from which q holds (see the XPath denotational semantics given previously in Section 3). To preserve semantics, the translation of $p[q]$ stops the "selecting navigation" to those nodes reached by p, then filters them, depending on whether q holds. We express this by introducing a dual formal translation function for XPath qualifiers, noted $Q^{\leftarrow}[\![\cdot]\!](\cdot)$ (and shown in Figure 10), which performs "filtering" instead of navigation. Specifically, $P^{\rightarrow}[\![\cdot]\!](\cdot)$ can be seen as the "navigational" translating function: The translated formula holds for target nodes of the given path. On the other hand, $Q^{\leftarrow}[\![\cdot]\!](\cdot)$ can be seen as the

$$
\begin{array}{lcl}
A^{\leftarrow}[\![\cdot]\!](\cdot) & : & Axis \times \mathcal{L}_\mu \longrightarrow \mathcal{L}_\mu \\
A^{\leftarrow}[\![\text{self}]\!](\chi) & = & \chi \\
A^{\leftarrow}[\![\text{following-sibling}]\!](\chi) & = & A^{\rightarrow}[\![\text{preceding-sibling}]\!](\chi) \\
A^{\leftarrow}[\![\text{child}]\!](\chi) & = & A^{\rightarrow}[\![\text{parent}]\!](\chi) \\
A^{\leftarrow}[\![\text{descendant}]\!](\chi) & = & A^{\rightarrow}[\![\text{ancestor}]\!](\chi) \\
A^{\leftarrow}[\![\text{descendant-or-self}]\!](\chi) & = & A^{\rightarrow}[\![\text{ancestor-or-self}]\!](\chi) \\
A^{\leftarrow}[\![\text{parent}]\!](\chi) & = & A^{\rightarrow}[\![\text{child}]\!](\chi) \\
A^{\leftarrow}[\![\text{ancestor}]\!](\chi) & = & A^{\rightarrow}[\![\text{descendant}]\!](\chi) \\
A^{\leftarrow}[\![\text{ancestor-or-self}]\!](\chi) & = & A^{\rightarrow}[\![\text{descendant-or-self}]\!](\chi) \\
A^{\leftarrow}[\![\text{preceding-sibling}]\!](\chi) & = & A^{\rightarrow}[\![\text{following-sibling}]\!](\chi) \\
A^{\leftarrow}[\![\text{following}]\!](\chi) & = & A^{\rightarrow}[\![\text{preceding}]\!](\chi) \\
A^{\leftarrow}[\![\text{preceding}]\!](\chi) & = & A^{\rightarrow}[\![\text{following}]\!](\chi)
\end{array}
$$

Fig. 11. Symmetry of axes inside qualifiers.

"filtering" translating function: It states the existence of a path, without moving to its result. The translated formula $Q^{\leftarrow}[\![q]\!](\chi)$ (respectively, $P^{\leftarrow}[\![p]\!](\chi)$) holds for nodes from which there exists a qualifier q (respectively, a path p) leading to a node verifying χ.

XPath translation into μ-calculus is based on these two translating "modes," the first being used for paths and the second for qualifiers. Note that whenever the "filtering" mode is entered, it will never be left. This differs from the denotational semantics given previously in Section 3, in which the formal semantics functions for paths and qualifiers are mutually recursive (and cause naive implementations to be unnecessarily complex, as pointed out by Gottlob et al. [2005]). Translations of paths inside qualifiers are also given in Figure 10. They use the specific translations for axes inside qualifiers, based on XPath symmetry, shown in Figure 11.

The cost of the translation is linear in length of the XPath expression, since there is no duplication of subformulae of arbitrary length in the formal translations. Formulae in which the formal parameter χ appears twice (see Figures 8 and 10) do not cause such duplication, since the value of χ is either φ_{context} or φ_{root} constants.

Note that the translation of an XPath expression is a sentence. Indeed, for absolute XPath expressions, the translation starts from the root (the initial formal parameter is φ_{root}). For relative expressions, the translated formula is closed by the initial formal parameter φ_{context} modeling context nodes.

We can prove that the translated $\mathcal{L}_\mu$ formula over binary trees is semantically equivalent to the original XPath expression over corresponding unranked trees. For instance, if we relate our translations in $\mathcal{L}_\mu$ to the XPath denotational semantics given previously in Section 3:

PROPOSITION 3.1. *Let T' be an XML tree, and e an XPath expression. Then, for all $y' \in T'$, the following are equivalent:*

—*There exists $x' \in T'$ such that $y' \in \mathcal{S}_e[\![e]\!]_{x'}^{T'}$,*
—*$T, y \models \varphi_{root} \wedge \varphi_{ft} \wedge (E^{\rightarrow}[\![e]\!](\varphi_{context}))^{\text{FBT}}$*

where y is the counterpart of y' in the binary tree representation T of T'.

The proof is done by a straightforward structural induction that "peels off" the compositional layers of each set of rules. This result links XPath decision problems in the absence of XML types to satisfiability in $\mathcal{L}_\mu$. We now show how XML types can also be translated in the μ-calculus.

4. XML TYPES

XML types describe structural constraints for XML documents. Several formalisms exist for describing classes of XML documents (see Murata et al. [2005] for an overview). In this article, we translate the class of regular tree languages that gathers all widely used formalisms for describing types of XML documents (including the well-known DTDs and XML schemas) into $\mathcal{L}_\mu$ over binary trees.

We begin with the syntactic definition of tree type expressions. We define a type T as follows:

$$\begin{aligned}\mathcal{L}_{CFT} \ni T \ ::= \ & \emptyset \mid () \mid X \mid l[T] \mid T_1, T_2 \mid T_1 \mid T_2 \mid, \\ & \text{let}\,(X_i \to T_i)_{1\le i\le m}\,\text{in}\,T\end{aligned}$$

where $l \in \Sigma$ and $X \in \mathit{TVar}$, assuming that TVar is a countably infinite set of type variables. Abbreviated type expressions can be defined as follows:

$$\begin{aligned} T? &= () \mid T \\ T* &= \text{let} X \to T \,\text{in}\, T, X \mid () \\ T^+ &= T, T* \end{aligned}$$

Given an environment θ of type-variable bindings, the semantics of tree types is given by the denotation function $[\![\cdot]\!]_\theta$:

$$\begin{aligned}
[\![\cdot]\!]_\cdot \quad & : \ \mathcal{L}_{CFT} \times (\mathit{TVar} \to 2^{\mathcal{T}_\Sigma^n}) \to 2^{\mathcal{T}_\Sigma^n} \\
[\![\emptyset]\!]_\theta \quad & = \emptyset \\
[\![()]\!]_\theta \quad & = \{()\} \\
[\![X]\!]_\theta \quad & = \theta(X) \\
[\![l[T])]\!]_\theta \quad & = \{l'(t) \mid l' \prec l \wedge t \in [\![T]\!]_\theta\} \\
[\![T_1, T_2]\!]_\theta \quad & = \{t_1, t_2 \mid t_1 \in [\![T_1]\!]_\theta \wedge t_2 \in [\![T_2]\!]_\theta\}, \\
[\![T_1 \mid T_2]\!]_\theta \quad & = [\![T_1]\!]_\theta \cup [\![T_2]\!]_\theta \\
[\![\text{let}\,(X_i \to T_i)_{1\le i\le m}\,\text{in}\,T]\!]_\theta \quad & = [\![T]\!]_{\mathit{lfp}(S)}
\end{aligned}$$

where $\prec$ is a global subtagging relation: a reflexive and transitive relation on labels,[5] and $S(\theta') = \theta[X_i \mapsto [\![T_i]\!]_{\theta'}]_{i\ge1}$. Note that each function S is monotone, according to the ordering $\subseteq$ on $\mathit{TVar} \to 2^{\mathcal{T}_\Sigma^n}$, and thus has a least fixpoint $\mathit{lfp}(S)$.

Types as previously defined actually correspond to arbitrary context-free tree types, for which the decision problem for inclusion is known to be undecidable [Hopcroft et al. 2000]. We impose the additional restriction used in Hosoya et al. [2005] to reduce the expressive power of considered types so that they correspond to regular tree languages. The restriction consists in a simple syntactic condition that allows unguarded (i.e., not enclosed by a label) recursive

[5]Subtagging goes beyond the expressive power of DTDs, but a similar notion called "substitution groups" exists in XML schemas (see Hosoya et al. [2005] for more details on subtagging).

uses of variables, but restricts them to tail positions.[6] This condition ensures regularity, and we name $\mathcal{L}_{RT}$ the resulting class of regular tree languages. From an XML point of view, regular tree types form a superset of standards, such as XML schemas and DTDs. We further detail the connection with the widely used DTD standard.

4.1 Document Type Definitions

As defined in the W3C recommendation, DTDs [Bray et al. 2004] are local tree grammars[7] which are strictly less expressive than regular tree types. In XML terminology, a type expression is often called the *content model*. DTD content models are described by the following syntax:

$$T ::= l \mid T_1 \shortmid T_2 \mid T_1, T_2 \mid T? \mid T^* \mid T^+ \mid (),$$

where $l \in \Sigma$. From the W3C specification, we see a DTD as a function that associates a content model to each label taken from a subset Σ' of Σ such that Σ' gathers all labels used in content models. We thus represent the set $\mathcal{L}_{DTD}$ of tree types described by DTDs as follows:

$$\begin{aligned}\mathcal{L}_{DTD} \ni T \ ::= \ & l \mid T_1 \shortmid T_2 \mid T_1, T_2 \mid T? \mid T^* \mid T^+ \mid () \mid \\ & \text{let}\,(l_i \to T_i)_{1 \le i \le m} \,\text{in}\, T\end{aligned}$$

Note that $\mathcal{L}_{DTD} \subseteq \mathcal{L}_{RT}$ is obvious, by associating a unique type variable to each label. In the following, we therefore no longer distinguish DTDs from general regular tree types.

4.2 Binarization of Types

In Section 2.1, we used a straightforward isomorphism between binary trees and sequences of unranked trees. There is also an isomorphism between unranked and binary tree types, which follows exactly the same intuition as for trees.

Binary tree types are described by the following syntax:

$$\begin{aligned}\mathcal{L}_{BT} \ni T \ ::= \ & \emptyset \mid \epsilon \mid T_1 \shortmid T_2 \mid l(X_1, X_2) \mid \\ & \text{let}\,(X_i \to T_i)_{1 \le i \le m} \,\text{in}\, T\end{aligned}$$

For any type, there is an equivalent binary type, and vice versa. We use the translation function shown in Figure 12 (adapted from that found in Hosoya et al. [2005]) to convert a type into its corresponding binary representation. The function considers the environment $\theta : \mathit{TVar} \to \mathcal{L}_{RT}$ for accessing the type bound to a variable X_i by constructs of the form "$\text{let}\,(X_i \to T_i)_{1 \le i \le m} \,\text{in}\, T$."

[6]For instance, the type "let$(X \to a[], Y)(Y \to b[], X \shortmid ())$ in X" is allowed.

[7]A local tree grammar is a regular tree grammar without *competing* nonterminals. Two nonterminals A and B of a tree grammar are said to compete with each other if one production rule has A in its lefthand-side, one production rule has B in its lefthand-side, and these two rules share the same terminal symbol in the righthand-side.

$$\begin{array}{lll}
\mathcal{B}(\cdot) & : & \mathcal{L}_{RT} \to \mathcal{L}_{BT} \\
\mathcal{B}(\emptyset) & = & \emptyset \\
\mathcal{B}(()) & = & \epsilon \\
\mathcal{B}(X) & = & \mathcal{B}(\theta(X)) \\
\mathcal{B}(l[T]) & = & \text{let } (X_1 \to \mathcal{B}(T))(X_2 \to \epsilon) \text{ in } l(X_1, X_2) \\
\mathcal{B}(T_1 \mid T_2) & = & \mathcal{B}(T_1) \mid \mathcal{B}(T_2) \\
\mathcal{B}(\text{let } (X_i \to T_i)_{1\le i\le m} \text{ in } T) & = & \text{let } (X_i \to \mathcal{B}(T_i))_{1\le i\le m} \text{ in } \mathcal{B}(T) \\
\mathcal{B}(\emptyset, T) & = & \emptyset \\
\mathcal{B}((), T) & = & \mathcal{B}(T) \\
\mathcal{B}(X, T) & = & \mathcal{B}(\theta(X), T) \\
\mathcal{B}(l[T_1], T_2) & = & \text{let } (X_1 \to \mathcal{B}(T_1))(X_2 \to \mathcal{B}(T_2)) \text{ in } l(X_1, X_2) \\
\mathcal{B}((T_1 \mid T_2), T_3) & = & \mathcal{B}(T_1, T_3) \mid \mathcal{B}(T_2, T_3) \\
\mathcal{B}((T_1, T_2), T_3) & = & \mathcal{B}(T_1, (T_2, T_3)) \\
\mathcal{B}(\text{let } (X_i \to T_i)_{1\le i\le m} \text{ in } T, T') & = & \text{let } (X_i \to \mathcal{B}(T_i))_{1\le i\le m} \text{ in } \mathcal{B}(T, T')
\end{array}$$

Fig. 12. Binarization of tree types.

4.3 Translation into μ-Calculus

We now introduce the translation of regular tree types into μ-calculus, which is based on the binary representation of types. In order to simplify translation, we introduce a notation for a n-ary least fixpoint binder:

$$\text{let}_\mu\ (X_i.\varphi_i)_{1\le i\le m} \text{ in } \psi$$

This notation is actually a syntactic sugar for ψ, where all free occurrences of X_i have been replaced by $\mu X_i.\varphi_i$ until ψ becomes closed (i.e., all X_i in ψ are in scope of their corresponding unary μ-binder). This provides a shorthand for denoting a $\mathcal{L}_\mu$ formule which would be of exponential size if expressed using only the unary least fixpoint construct. Such a naive expansion contains unnecessary duplicate formulae, whereas the satisfiability solver operates only on a single copy of them (see Section 6.1). Therefore, the n-ary binder is a useful compact notation for representing $\mathcal{L}_\mu$ translations of recursive types, without introducing useless blow-ups between representation of formulae and their satisfiability test.

The translation from binary regular tree types into $\mathcal{L}_\mu$ formulae is given by the following rules:

$$\begin{array}{lll}
[\![\cdot]\!] & : & \mathcal{L}_{BT} \to \mathcal{L}_\mu \\
[\![\emptyset]\!] & = & \bot \\
[\![\epsilon]\!] & = & \bot \\
[\![T_1 \mid T_2]\!] & = & [\![T_1]\!] \vee [\![T_2]\!] \\
[\![l(X_1, X_2)]\!] & = & l \wedge \mathit{succ}_1(X_1) \wedge \mathit{succ}_2(X_2) \\
[\![\text{let}(X_i \to T_i)_{1\le i\le m} \text{ in } T]\!] & = & \text{let}_\mu\ (X_i.[\![T_i]\!])_{1\le i\le m} \text{ in } [\![T]\!]
\end{array},$$

where there is an implicit bijective correspondence between $\mathcal{L}_{BT}$ variables from *TVar* and $\mathcal{L}_\mu$ variables from *Var*. Note that the translations of the empty tree type and the empty tree are the same, since we choose not to explicitly mention empty trees in satisfiability results. The function $\mathit{succ}_\cdot(\cdot)$ sets the tree frontier accordingly:

$$\mathit{succ}_\cdot(\cdot) \quad : \quad \mathit{Prog} \times \mathit{TVar} \to \mathcal{L}_\mu$$

$$\mathit{succ}_\alpha(X) = \begin{cases} [\alpha]X & \mathit{if\ nullable}(X) \\ \langle\alpha\rangle X & \mathit{if\ not\ nullable}(X) \end{cases}$$

The predicate $nullable(\cdot)$ indicates whether a type contains the empty tree:

$$\begin{array}{lcl}
nullable(\cdot) & : & TVar \cup \mathcal{L}_{BT} \rightarrow \{\text{true}, \text{false}\} \\
nullable(X) & = & nullable(\theta(X)) \\
nullable(\emptyset) & = & \text{false} \\
nullable(\epsilon) & = & \text{true} \\
nullable(l) & = & \text{false} \\
nullable(T_1 \mid T_2) & = & nullable(T_1) \vee nullable(T_2) \\
nullable(l(X_1, X_2)) & = & \text{false} \\
nullable(\text{let}(X_i \rightarrow T_i)_{1 \leq i \leq m} \text{ in } T) & = & nullable(T)
\end{array}$$

5. XML DECISION PROBLEMS

We have translated both XPath over unranked trees, and regular unranked tree types in the unifying $\mathcal{L}_\mu$ logic over binary trees. Owing to these embeddings, we now reduce XML decision problems (such as XPath containment, equivalence, satisfiability, overlap, and coverage) to satisfiability in $\mathcal{L}_\mu$.

We first introduce some simplified notations. For an XPath expression $e \in \mathcal{L}_{\text{XPath}}$, we note φ_e the translated formula $E^{\rightarrow}[\![e]\!](\varphi_{\text{context}}) \in \mathcal{L}_\mu$. Furthermore, we note $\mathcal{T}$ the set of trees: By default, $\mathcal{T} = \mathcal{T}_\Sigma^n$, and whenever an optional DTD $d \in \mathcal{L}_{DTD}$ is specified, $\mathcal{T} = [\![d]\!]_\emptyset$. Finally, we note $\varphi_\mathcal{T}$ the $\mathcal{L}_\mu$ embedding of the tree language $\mathcal{T}$. In the absence of DTDs $\varphi_\mathcal{T} = \top$, and $\varphi_\mathcal{T} = [\![\mathcal{B}(d)]\!]$ in the presence of $d \in \mathcal{L}_{DTD}$.

Several decision problems needed in applications can be expressed in terms of $\mathcal{L}_\mu$ formulae:

- *XPath Containment*
  —Input: $e_1, e_2 \in \mathcal{L}_{\text{XPath}}$, and optional $d \in \mathcal{L}_{DTD}$
  —Problem: Does e_2 always select all nodes selected by e_1?
  —Definition: $\forall t \in \mathcal{T}, \forall x \in t, \mathcal{S}_e[\![e_1]\!]_x^t \subseteq \mathcal{S}_e[\![e_2]\!]_x^t$
  —Tested $\mathcal{L}_\mu$ formula: $\varphi_{e_1} \wedge \neg\varphi_{e_2}$
- *XPath Equivalence*
  —Input: $e_1, e_2 \in \mathcal{L}_{\text{XPath}}$, and optional $d \in \mathcal{L}_{DTD}$
  —Problem: Does e_2 always select exactly the same nodes as e_1?
  —Definition: $\forall t \in \mathcal{T}, \forall x \in t, \mathcal{S}_e[\![e_1]\!]_x^t = \mathcal{S}_e[\![e_2]\!]_x^t$
  —Equivalence can be tested by two successive and separate containment checks
- *XPath Satisfiability*
  —Input: $e \in \mathcal{L}_{\text{XPath}}$ and optional $d \in \mathcal{L}_{DTD}$
  —Problem: Will e ever return a nonempty set of nodes?
  —Definition: $\forall t \in \mathcal{T}, \forall x \in t, \mathcal{S}_e[\![e]\!]_x^t = \emptyset$
  —Tested $\mathcal{L}_\mu$ formula: φ_e
- *XPath Overlap*
  —Input: $e_1, e_2 \in \mathcal{L}_{\text{XPath}}$, and optional $d \in \mathcal{L}_{DTD}$
  —Problem: May e_1 and e_2 select common nodes?
  —Definition: $\forall t \in \mathcal{T}, \forall x \in t, \mathcal{S}_e[\![e_1]\!]_x^t \cap \mathcal{S}_e[\![e_2]\!]_x^t = \emptyset$
  —Tested $\mathcal{L}_\mu$ formula: $\varphi_{e_1} \wedge \varphi_{e_2}$

- *XPath Coverage*
  —Input: $e_1, e_2, ..., e_n \in \mathcal{L}_{\text{XPath}}$, and optional $d \in \mathcal{L}_{DTD}$
  —Problem: Are nodes selected by e_1 always selected by one of the $e_2, ..., e_n$?
  —Definition: $\forall t \in \mathcal{T}, \forall x \in t, \mathcal{S}_e[\![e_1]\!]_x^t \subseteq \bigcup_{2\leq i\leq n} \mathcal{S}_e[\![e_i]\!]_x^t$
  —Tested $\mathcal{L}_\mu$ formula: $\varphi_{e_1} \wedge \bigwedge_{2\leq i\leq n} \neg\varphi_{e_i}$

Note that for the containment problem, we actually test the unsatisfiability of $\varphi_{e_1} \wedge \neg\varphi_{e_2}$. Indeed, checking that an XPath expression e_1 is contained into another expression e_2 consists of checking that the implication $\varphi_{e_1} \Rightarrow \varphi_{e_2}$ holds for all trees. In other words, there exists no tree for which the results of e_1 are not included in those of e_2, that is, the negated implication $\varphi_{e_1} \wedge \neg\varphi_{e_2}$ is unsatisfiable.

Since we need to enforce the finite binary tree model property (as seen in Section 2.1), we formulate decision problems from the root, and the actually checked formula becomes:

$$\varphi_{\text{root}} \wedge \varphi_{\text{ft}} \wedge (\varphi_{\mathcal{T}} \wedge \mu X.\varphi_{\text{tested}} \vee \langle 1\rangle X \vee \langle 2\rangle X)^{\text{FBT}}, \tag{1}$$

where φ_{tested} corresponds to a particular XPath decision problem from those given previously. Intuitively, the fixpoint is introduced for "plunging" XPath navigation performed by φ_{tested} at any location in the tree. It is, for example, necessary for relative XPath expressions that involve upward navigation in the tree.

It is important to note that formula (1) is always alternation-free, since embeddings of both XPath and tree types produce alternation-free formulae, and the negation of an alternation-free sentence remains alternation-free. In practice, the negated sentences introduced by XPath embeddings are turned into negation normal form by applying the rules given in Figure 4.

6. SYSTEM EVALUATION

The proposed approach has been fully implemented and the working system is available [Genevès and Layaïda 2006]. A compiler takes XPath expressions as input, and translates them into $\mathcal{L}_\mu$ formulae. Another compiler takes regular tree types as input (DTDs) and outputs their $\mathcal{L}_\mu$ translation. The formula of a particular decision problem is then composed, normalized, and solved.

6.1 Complexity Analysis and Implementation Principles

Our μ-calculus satisfiability solver is specialized for alternation-free μ-calculus with converse. A detailed description of the solver is beyond the scope of the article. We rather focus on its aspects which allow us to establish precise complexity results for the XML decision problems considered in this article. The $\mathcal{L}_\mu$ satisfiability solver is inspired by the tableau methods described in Tanabe et al. [2005] and Pan et al. [2002]. The algorithm relies on a top-down tableau method which attempts to construct satisfying Kripke structures by a fixpoint computation. Nodes of the tableau are specific subsets of a set called the Lean [Pan et al. 2002]. Given a formula $\psi \in \mathcal{L}_\mu$, the Lean is the subset of the Fischer-Ladner closure [Fischer and Ladner 1979] of ψ composed of atomic and modal subformulae of ψ [Pan et al. 2002]. The algorithm starts from the set of all possible nodes, and repeatedly removes inconsistent nodes until a fixpoint is reached. At the end of the computation, if ψ is present in a node of the fixpoint, then ψ is satisfiable. In this case, the fixpoint contains a satisfying model that can be easily extracted and used as a satisfying example XML tree.

We can now state the complexity of XML decision problems addressed in this article:

PROPOSITION 6.1. *XPath containment, equivalence, satisfiability, overlap, and coverage decision problems, in the presence or absence of regular tree constraints, can be solved in time complexity $2^{O(n \cdot \log n)}$, where n is the Lean size of the corresponding $\mathcal{L}_\mu$ formula.*

This upper bound is derived from:

(1) the linear translations of XPath and regular tree types into the μ-calculus; and
(2) the $2^{O(n \cdot \log n)}$ time complexity of our solver, which corresponds to the best known complexity for deciding alternation-free μ-calculus with converse over Kripke structures [Tanabe et al. 2005]. Note that this is more efficient than the complexity for the whole μ-calculus with converse [Vardi 1998], which is known to be $2^{O(n^4 \cdot \log n)}$ [Grädel et al. 2002].

The key observation for the linear translation of regular tree types is that only distinct atomic and modal subformulae of the translated formula are present in the Lean, even for an n-ary binder $\varphi = \text{let}_\mu\ (X_i.\varphi_i)_{1 \leq i \leq m}$ in X_k. More precisely, the Lean corresponding to the translation of φ contains at most:

—the two eventualities $\langle a \rangle \top$ for $a = 1, 2$;
—$2 \cdot m$ universalities $[a]\,\varphi$, where m is the number of binary tree type variables in the binder and the constant factor corresponds to the downward programs $a = 1, 2$; and
—the atomic propositions representing the alphabet symbols used in φ.

Deriving complexity from properties of the closure of a formula was first used by Fischer and Ladner [1979] for establishing the decidability of PDL in single exponential time. Analog observations have also been made for the modal logic K [Pan et al. 2002], and the μ-calculus over general Kripke structures [Tanabe et al. 2005]. Our results can be seen as an application of this technique to the case where regular tree types are combined with XPath bidirectional queries over finite trees.

Keys of the efficiency of the method for large practical instances are as follows:

(1) Nodes of the tableau contain only modal formulae and exactly one atomic proposition (for XML), which greatly reduces the number of enumerated nodes for large alphabets.
(2) Negation in the μ-calculus is rather straightforward compared to automata techniques. Indeed, handling $\mathcal{L}_\mu$ formulae in negation normal form simply reduces to checking membership of atomic propositions in tableau nodes.

This contrasts with tree automata techniques, which require for every negation the full construction and complementation of automata with an exponential blow-up. As pointed out in Baader and Tobies [2001] and Pan et al. [2002], tableau methods for logics with the tree model property can be viewed as implementations of the automata-theoretic approach, which avoids an explicit automata construction.

(3) Our implementation relies on representing sets of nodes and operating on them symbolically using Binary Decision Diagrams (BDDs) [Bryant 1986]. BDDs provide a canonical representation of boolean functions and their effectiveness is well known in formal verification of systems [Edmund et al. 1999]. In our approach, BDD variables encode truth status of Lean formulae. The cost of BDD operations is very sensitive to variable ordering. Finding the optimal variable ordering is known to be NP-complete [Hojati et al. 1996], however several heuristics are known to perform well in practice [Edmund et al. 1999]. Choosing a good initial variable order does significantly improve performance. We found out that preserving locality of the initial problem is essential. We observed that the variable order determined by the breadth-first traversal of the initial formula (thus keeping sister subformulae in close proximity while ordering Lean formulae) yields better results in practice.

6.2 Experimental Results

The objective of the section aims at testing the practical performance of our method. We carried out several testing scenarios.[8] First, we used an XPath benchmark [Franceschet 2005] whose goal is to cover XPath features by gathering a significant variety of XPath expressions met in real-world applications. In this first test series, we do not yet consider types and only focus on the XPath containment problem, since its logical formulation (presented in Section 5) is the most complex because it requires the logic to be closed under negation. The first test series consists of finding the relation holding for each pair of queries from the benchmark. This means checking the containment of each query of the benchmark against all others. We note $q_i \subseteq q_j$ whenever the query q_i is contained in the query q_j. Comparisons of two queries q_i and q_j may yield three different results:

(1) $q_i \subseteq q_j$ and $q_j \subseteq q_i$, the queries are semantically equivalent, we note $q_i \equiv q_j$;
(2) $q_i \subseteq q_j$ but $q_j \not\subseteq q_i$, we denote, by $q_i \subset q_j$ or alternatively, by $q_j \supset q_i$; or
(3) $q_i \not\subseteq q_j$ and $q_j \not\subseteq q_i$, queries are not related, we note $q_i \not\sim q_j$.

Queries are presented in Figure 13. Corresponding results, together with the running times of the decision procedure, are summarized in Table I. Times reported in milliseconds correspond to the actual running time of the μ-calculus satisfiability solver, without the extra time spent for parsing XPath nor the

[8]Experiments have been conducted on a Pentium 4, 3 Ghz, with 512Mb of RAM, running Eclipse on Windows XP.

q_1	/site/regions/*/item
q_2	/site/closedauctions/closedauction/annotation/description/parlist/listitem/text/keyword
q_3	//keyword
q_4	/descendant-or-self::listitem/descendant-or-self::keyword
q_5	/site/regions/*/item[parent::namerica or parent::samerica]
q_6	//keyword/ancestor::listitem
q_7	//keyword/ancestor-or-self::mail
q_8	/site/regions/namerica/item | /site/regions/samerica/item
q_9	/site/people/person[address and (phone or homepage)]

Fig. 13. XPath queries taken from the XPathmark benchmark.

Table I. Results and Total Computation Times

Relation	Time (ms)	
	$\subseteq$	$\supseteq$
$q_1 \not\sim q_2$	18	23
$q_1 \not\sim q_3$	14	22
$q_1 \not\sim q_4$	9	12
$q_1 \supset q_5$	16	7
$q_1 \not\sim q_6$	22	13
$q_1 \not\sim q_7$	15	12
$q_1 \supset q_8$	9	11
$q_1 \not\sim q_9$	16	16
$q_2 \subset q_3$	32	33
$q_2 \subset q_4$	38	36
$q_2 \not\sim q_5$	24	23
$q_2 \not\sim q_6$	22	35
$q_2 \not\sim q_7$	31	36
$q_2 \not\sim q_8$	26	24
$q_2 \not\sim q_9$	34	31
$q_3 \supset q_4$	17	21
$q_3 \not\sim q_5$	5	7
$q_3 \not\sim q_6$	4	9
$q_3 \not\sim q_7$	12	12
$q_3 \not\sim q_8$	14	8
$q_3 \not\sim q_9$	13	15
$q_4 \not\sim q_5$	24	15
$q_4 \not\sim q_6$	9	12
$q_4 \not\sim q_7$	22	12
$q_4 \not\sim q_8$	14	21
$q_4 \not\sim q_9$	13	14
$q_5 \not\sim q_6$	14	11
$q_5 \not\sim q_7$	11	9
$q_5 \equiv q_8$	8	12
$q_5 \not\sim q_9$	18	21
$q_6 \not\sim q_7$	21	21
$q_6 \not\sim q_8$	15	17
$q_6 \not\sim q_9$	14	15
$q_7 \not\sim q_8$	26	18
$q_7 \not\sim q_9$	14	16
$q_8 \not\sim q_9$	11	10

(linear) cost of the translation into μ-calculus. Obtained results show that all tests are solved in several milliseconds. This suggests that XPath expressions used in real-world scenarios can be efficiently handled in practice.

As a second test series, we compare expressions found in research papers on the containment of XPath expressions. Figure 14 presents the expressions we collected. For this set of expressions, the tree pattern homomorphism technique [Miklau and Suciu 2004] returns false negatives, whereas our approach is complete. Figure 14 also shows the results obtained with our system. These suggest that our system is able to reasonably handle containment instances which are difficult to solve using other techniques.

Figure 15 presents the results of a third test series, including examples with intersection, and axes such as "following" and "preceding," which are not illustrated in the previous series.

The fourth test series aims at evaluating the effectiveness of the system for XPath decision problems in the presence of DTDs. We used a small recursive DTD (given in Figure 16), and real-world DTDs of the SMIL [Hoschka 1998] and XHTML [Pemberton 2000] standards. Table II gives the size of each DTD by

e_1	$/a[.//b[c/*//d]/b[c//d]/b[c/d]]$
e_2	$/a[.//b[c/*//d]/b[c/d]]$
e_3	$a[b]/*/d/*/g$
e_4	$a[b]/(b\|c)/d/(e\|f)/g$
e_5	$a[b]/b/d/e/g\|a/b/d/f/g$
e_6	$a/b/s//c/b/s/c//d$
e_7	$a//b/*/c//*/d$
e_8	$a[b/e][b/f][c]$
e_9	$a[b/e][b/f]$
e_{10}	/descendant::editor[parent::journal]
e_{11}	/descendant-or-self::journal/child::editor

Relation	Time (ms)	
	$\subseteq$	$\supseteq$
$e_1 \subset e_2$	281	153
$e_3 \supset e_4$	19	27
$e_3 \supset e_5$	26	14
$e_4 \supset e_5$	33	29
$e_6 \subset e_7$	47	33
$e_8 \subset e_9$	7	13
$e_{10} \equiv e_{11}$	16	15

Fig. 14. Results on XPath containment instances found in research papers.

e_{12}	a/b//c/following-sibling::d/e
e_{13}	a//d[preceding-sibling::c]/e
e_{14}	//a//b//c/following-sibling::d/e
e_{15}	//b[ancestor::a]//*[preceding-sibling::c]/e
e_{16}	/b[preceding::a]//following::c
e_{17}	/a/b//following::c
e_{18}	a/b[//c]/following::d/e
e_{19}	a//d[preceding::c]/e
e_{20}	a/b//d[preceding-sibling::c]/e
e_{21}	a/c/following::d/e
e_{22}	a/d[preceding::c]/e
e_{23}	a/b[//c]/following::d/e $\cap$ a/d[preceding::c]/e
e_{24}	a/c/following::d/e $\cap$ a/d[preceding::c]/e

Relation	Time (ms)	
	$\subseteq$	$\supseteq$
$e_{12} \subset e_{13}$	27	18
$e_{14} \subset e_{15}$	11	19
$e_{16} \subset e_{17}$	21	27
$e_{18} \subset e_{19}$	20	13
$e_{20} \equiv e_{12}$	24	21
$e_{21} \not\subset e_{22}$	13	18
$e_{23} \subset e_{21}$	21	19
$e_{24} \not\subset e_{18}$	17	12

Fig. 15. Results on examples including "following" and "preceding" axes.

```
<!ELEMENT people (person*)>
<!ELEMENT person (name,birthdate?,gender?,children?)>
<!ELEMENT name (firstname+, lastname) >
<!ELEMENT firstname (#PCDATA) >
<!ELEMENT lastname (#PCDATA) >
<!ELEMENT birthdate (#PCDATA) >
<!ELEMENT gender (#PCDATA) >
<!ELEMENT children (person+) >
```

Fig. 16. (People.dtd): a simple recursive DTD.

Table II. DTDs Used in Practical Experiments

DTD Examples	Symbols	Type Variables	Binary Type Variables
People.dtd (Figure 16)	8	15	11
SMIL 1.0 [Hoschka 1998]	19	29	11
XHTML 1.0 Strict [Pemberton 2000]	77	104	325

p_1	people/*
p_2	//person
p_3	/descendant-or-self/people/person
p_4	//children/person
p_5	switch/layout
p_6	smil/head//layout
p_7	smil/head//layout[ancestor::switch]
p_8	*/switch[ancestor::head]/descendant::seq/descendant::audio[preceding-sibling::video]
p_9	descendant::a[ancestor::a]
p_{10}	/descendant::*
p_{11}	html/(head ∣ body)
p_{12}	html/head/descendant::*
p_{13}	html/body/descendant::*
p_{14}	//img
p_{15}	//img[not *]

Fig. 17. XPath expressions used in the presence of DTDs.

Table III. Some Decision Problems in the Presence of DTDs and Results

XPath Decision Problem	Instance	DTD	Answer	Time (ms)
Containment	$p_1 \subseteq p_2$	People.dtd	true	32
Coverage	$p_2 \subseteq p_3 \cup p_4$	People.dtd	true	41
Satisfiability	p_5	SMIL 1.0	true	110
Overlap	$p_5 \cap p_6 \neq \emptyset$	SMIL 1.0	false	174
Containment	$p_6 \subseteq p_7$	SMIL 1.0	false	120
Satisfiability	p_8	SMIL 1.0	true	157
Satisfiability	p_9	XHTML 1.0	true	2630
Coverage	$p_{10} \subseteq p_{11} \cup p_{12} \cup p_{13}$	XHTML 1.0	true	2872
Containment	$p_{14} \subseteq p_{15}$	XHTML 1.0	true	2931

presenting the number of symbols used (alphabet size) and number of grammar production rules (type variables) in the unranked and binary representations.

For each DTD, we built several XPath decision problems using the expressions shown in Figure 17. Some decision problems and their results are presented in Table III. The system performs well for the respectively small, medium, and large recursive DTDs.

The satisfiability test for p_9 illustrates an additional benefit of our technique that automatically outputs a satisfying XML document (shown in Figure 18), enriched with XPath context and target information. Interestingly, this example also shows that the official XHTML DTD does not syntactically prohibit the nesting of anchors.

For the large XHTML case, we observe that the time needed is significantly more important, but deciding XPath problems remains practically feasible, especially for static analysis purposes wherein such operations are performed at compile-time.

These preliminary measurements shed light, for the first time, on the cost of solving XPath decision problems in practice.

```
<html>
  <head>
    <title/>
  </head>
  <body>
    <h1>
      <a>
        <span context="true">
          <a target="true">
            <br/>
          </a>
        </span>
      </a>
    </h1>
  </body>
</html>
```

Fig. 18. Generated XML tree for satisfiability of p_9 in the presence of the XHTML DTD.

7. RELATED WORK

Relatively close in spirit to our work is the constructive connection between XPath and formal logics, which is actively studied [Marx 2004b; Benedikt et al. 2005; Barceló and Libkin 2005]. From Marx [2004a] and Genevès and Vion-Dury [2004b], we know that XPath expressive power is close to first-order logic (FO). However, FO does not fully capture regular tree types [Benedikt and Segoufin 2005]. Thus, attempts to characterize XPath subfragments in terms of FO variants such as computational tree logic (CTL) [Marx 2004b; Miklau and Suciu 2004], which is equivalent to FO over tree structures [Marx 2004a; Barceló and Libkin 2005], are not intended to support regular XML types. The work found in Afanasiev et al. [2005] proposes a variant of propositional dynamic logic (PDL) [Fischer and Ladner 1979] with a similar EXPTIME complexity for reasoning about ordered trees, but its exact expressive power is still under study.

One of the most expressive (yet decidable) known logics is monadic second-order logic (MSO) over tree structures, which extends FO by quantification over sets of nodes. Specifically, the appropriate MSO variant which exactly captures regular tree types is the weak monadic second-order logic of two successors (WS2S) [Thatcher and Wright 1968; Doner 1970]. From Arnold and Niwinski [1992] and Kupferman and Vardi [1999], we know that WS2S is exactly as expressive as the alternation-free fragment (AFMC) of the propositional modal μ-calculus introduced in Kozen [1983]. However, the satisfiability problem for WS2S is nonelementary[9] while in EXPTIME[10] AFMC. Moreover,

[9]We recall that the term *elementary* introduced by Grzegorczyk [1953] refers to functions obtained from some basic functions by operations of limited summation and limited multiplication. Consider the function *tower*() defined by:

$$\begin{cases} tower(n,0) = n \\ tower(n,k+1) = 2^{tower(n,k)} \end{cases}$$

Grzegorczyk [1953] has shown that every elementary function in one argument is bounded by $\lambda n.tower(n,c)$ for some constant c. Hence, the term *nonelementary* refers to a function that grows faster than any such function.

[10]The complexity class EXPTIME is the set of all decision problems solvable by a deterministic Turing machine in $O(2^{p(n)})$ time, where $p(n)$ is a polynomial function of the input size n.

AFMC subsumes all early logics, such as CTL [Clarke and Emerson 1981] and PDL [Fischer and Ladner 1979]. Furthermore, the work in Vardi [1998] adds converse programs to propositional modal μ-calculus and shows that the resulting logic still admits an EXPTIME decision procedure for satisfiability. It follows that alternation-free modal μ-calculus with converse sounds like an appropriate logic for XML: It is expressive enough to capture a significant class of XPath decision problems, while potentially providing efficient and practically effective decision procedures.

From the point of view of computational complexity, some EXPTIME upper bounds are already known for the satisfiability and containment of specific subsets of our XPath fragment. The complexity of XPath satisfiability in the presence of DTDs is studied in Benedikt et al. [2005]. XPath containment has specifically attracted a lot of research attention [Amer-Yahia et al. 2001, Deutsch and Tannen 2001, Miklau and Suciu 2004, Neven and Schwentick 2003, Schwentick 2004, Wood 2000; 2003]. Prior work concentrated on various combinations of the previous factors for obtaining complexity results (see Schwentick [2004] for an overview). Specifically, the focus was given to restricted positive XPath subfragments without upward axes. In particular, Neven and Schwentick [2003] proves an EXPTIME upper bound for containment (in the presence of DTDs) of queries containing the "child" and "descendant" axes, and union of paths. Deutsch and Tannen [2001] consider XPath containment in the presence of DTDs and simple XPath integrity constraints (SXICS). They maintain that this problem is undecidable in general and in the presence of bounded SXICs and DTDs. Containment for the fragment $XP^{\{*,//,[\,]\}}$ is shown to be coNP-complete in Miklau and Suciu [2004], where the containment mapping technique relies on a polynomial-time tree homomorphism algorithm that gives a sufficient, but not necessary, condition for containment of $XP^{\{*,//,[\,]\}}$ in general. Additionally, the containment problem is shown to be in EXPTIME for the fragments $XP^{\{//,[\,]\}}$, $XP^{\{//,[\,],|\}}$, $XP^{\{//,|\}}$ in the presence of DTDs in Wood [2003].

Compared to all these previous works, the XPath fragment we consider is far more complete and much more realistic. We also present a single unifying logical framework in which all major XPath features, but also regular tree types, fit together. Moreover, our framework yields effective decision procedures that are usable in practice for real-world scenarios (whereas no usable working system has been reported in prior work).

Finally, from a theoretical perspective, we see the connection between XML and μ-calculus as a simple way of deriving the precise upper bound time complexity $2^{O(n \cdot \log n)}$ of XML decision problems, where n is the combined size of considered XPath queries and tree types.

8. CONCLUSION

We propose a new logical approach for XPath decision problems. XPath queries and regular tree types are translated into the μ-calculus. XML decision problems are expressed as formulae in this logic, then decided using an efficient decision procedure for μ-calculus satisfiability. This article makes several contributions.

First, we propose a specific variant of the μ-calculus, namely, alternation-free modal μ-calculus with converse, as the appropriate logic for reasoning on XML trees, XPath queries, and XML types. As a valuable outcome, we show how both XPath and regular tree types can be linearly translated in the μ-calculus.

Second, we take advantage of these translations to reduce several XML decision problems to satisfiability in $\mathcal{L}_\mu$. We obtain effective EXPTIME decision procedures that are usable in practice. The considered XPath fragment includes union, intersection, path composition (together with all downward and upward axes), branching, Boolean connectives, wildcards, and negation, in the presence or absence of DTDs. This fragment is far more complete than other fragments addressed in previous studies. We provide practical experiments and detailed results that corroborate our claim that this approach is efficient in practice for real-world XPath expressions and DTDs. Our system has been fully implemented [Genevès and Layaïda 2006] and can be used for the static analysis of XML specifications. This strengthens the hope for an effective analysis of standard XML transformations in the near future.

Eventually, an additional advantage of the technique is to allow generation of XML tree examples when the containment does not hold. We believe this makes our method of special interest for many applications, including debuggers or applications that can benefit from a precise reporting during static analysis stages.

One direction of future work consists in specifically tuning the μ-calculus satisfiability solver for XML. Incorporating XML peculiarities directly into the core of the μ-calculus solver (instead of general Kripke structures) may yield even more efficient decision procedures.

REFERENCES

Abiteboul, S. and Vianu, V. 1999. Regular path queries with constraints. *J. Compute. Syst. Sci. 58*, 3, 428–452.

Afanasiev, L., Blackburn, P., Dimitriou, I., Gaiffe, B., Goris, E., Marx, M., and de Rijke, M. 2005. PDL for ordered trees. *J. Appl. Non-Classical Logics 15*, 2, 115–135.

Amer-Yahia, S., Cho, S., Lakshmanan, L. V. S., and Srivastava, D. 2001. Minimization of tree pattern queries. *SIGMOD Record 30*, 2, 497–508.

Arnold, A. and Niwinski, D. 1992. Fixed point characterization of weak monadic logic definable sets of trees. In *Tree Automata and Languages*. North-Holland, Amsterdam, Netherlands. 159–188.

Baader, F. and Tobies, S. 2001. The inverse method implements the automata approach for modal satisfiability. In *IJCAR '01: Proceedings of the 1st International Joint Conference on Automated Reasoning*. Springer-Verlag, London. 92–106.

Barceló, P. and Libkin, L. 2005. Temporal logics over unranked trees. In *LICS: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, New York. 31–40.

Benedikt, M., Fan, W., and Geerts, F. 2005. XPath satisfiability in the presence of DTDs. In *PODS: Proceedings of the 24th ACM Symposium on Principles of Database Systems*. ACM, New York. 25–36.

Benedikt, M. and Segoufin, L. 2005. Regular tree languages definable in FO. In *STACS: Proceedings of the 22nd Annual Symposium on Theoretical Aspects of Computer Science*. Lcture Notes in Computer Science vol. 3404. Springer-Verlag. 327–339.

Berglund, A., Boag, S., Chamberlin, D., Fernández, M. F., Kay, M., Robie, J., and Siméon, J. 2006. XML path language (XPath) 2.0, W3C candidate recommendation. http://www.w3.org/TR/xpath20/.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. 2004. Extensible markup language (XML) 1.0 (3rd ed.), W3C recommendation. http://www.w3.org/TR/2004/REC-xml-20040204/.

Bryant, R. E. 1986. Graph-Based algorithms for Boolean function manipulation. *IEEE Trans. Comput. 35*, 8, 677–691.

Clark, J. and DeRose, S. 1999. XML path language (XPath) version 1.0, W3C recommendation. http://www.w3.org/TR/1999/REC-xpath-19991116.

Clarke, E. M. and Emerson, E. A. 1981. Design and synthesis of synchronization skeletons using branching-time temporal logic. *In Proceedings of the Logic of Programs Workshop*. Lecture Notes in Computer Science vol. 131. Springer-Verlag. 52–71.

Deutsch, A. and Tannen, V. 2001. Containment of regular path expressions under integrity constraints. In *KRDB: Proceedings of the 8th International Workshop on Knowledge Representation Meets Databases*. CEUR Workshop Proceedings vol. 45, 1–11.

Doner, J. 1970. Tree acceptors and some of their applications. *J. Comput. Syst. Sci. 4*, 406–451.

Edmund, M., Clarke, J., Grumberg, O., and Peled, D. A. 1999. *Model Checking*. MIT Press, Cambridge, MA.

Fallside, D. C. and Walmsley, P. 2004. XML Schema part 0: Primer 2nd ed., W3C recommendation. http://www.w3.org/TR/xmlschema-0/.

Fan, W., Chan, C.-Y., and Garofalakis, M. 2004. Secure XML querying with security views. In *SIGMOD: Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, New York. 587–598.

Fischer, M. J. and Ladner, R. E. 1979. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci. 18*, 2, 194–211.

Franceschet, M. 2005. XPathMark—An XPath benchmark for XMark generated data. In *XSYM: Proceedings of the 3rd International Symposium on Database and XML Technologies*. Lecture Notes in Computer Science vol. 3671. Springer-Verlag. 129–143.

Genevès, P. and Layaïda, N. 2006. A μ-calculus satisfiability solver for XML. http://wam.inrialpes.fr/xml.

Genevès, P. and Vion-Dury, J.-Y. 2004a. Logic-Based XPath optimization. In *DocEng: Proceedings of the ACM Symposium on Document Engineering*. ACM Press, New York. 211–219.

Genevès, P. and Vion-Dury, J.-Y. 2004b. XPath formal semantics and beyond: A Coq-Based approach. In *TPHOLs: Emerging Trends Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*. Salt Lake City, UT. 181–198.

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

Grädel, E., Thomas, W., and Wilke, T., Eds. 2002. *Automata Logics, and Infinite Games: A Guide to Current Research*. Springer-Verlag, New York.

Grzegorczyk, A. 1953. Some classes of recursive functions. *Rozprawy Matematyczne 4*, 1–45.

Hojati, R., Krishnan, S. C., and Brayton, R. K. 1996. Early quantification and partitioned transition relations. In *ICCD: Proceedings of the International Conference on Computer Design, VLSI in Computers and Processors*. IEEE Computer Society. 12–19.

Hopcroft, J. E., Motwani, R., Rotwani, and Ullman, J. D. 2000. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman, Boston.

Hoschka, P. 1998. Synchronized multimedia integration language (SMIL) 1.0 specification, W3C recommendation. http://www.w3.org/TR/REC-smil/.

Hosoya, H., Vouillon, J., and Pierce, B. C. 2005. Regular expression types for XML. *ACM Trans. Program. Lang. Syst. 27*, 1, 46–90.

Kozen, D. 1983. Results on the propositional μ-calculus. *Theoret. Comput. Sci. 27*, 333–354.

Kozen, D. 1988. A finite model theorem for the propositional μ-calculus. *Studia Logica 47*, 3, 233–241.

Kupferman, O. and Vardi, M. 1999. The weakness of self-complementation. In *Proceedings of the 16th Symposium on Theoretical Aspects of Computer Science*. Lecture Notes in Computer Science vol. 1563. Springer-Verlag. 455–466.

LEVIN, M. Y. AND PIERCE, B. C. 2005. Type-Based optimization for regular patterns. In *DBPL: Proceedings of the 10th International Symposium on Database Programming Languages*. Lecture Notes in Computer Science vol. 3774. Springer-Verlag.

MARTENS, W. AND NEVEN, F. 2004. Frontiers of tractability for typechecking simple XML transformations. In *PODS: Proceedings of the 23rd ACM Symposium on Principles of Database Systems*. ACM, New York. 23–34.

MARX, M. 2004a. Conditional XPath, the 1st order complete XPath dialect. In *PODS: Proceedings of the 23rd ACM Symposium on Principles of Database Systems*. ACM, New York. 13–22.

MARX, M. 2004b. XPath with conditional axis relations. In *Proceedings of the 9th International Conference on Extending Database Technology*. Lecture Notes in Computer Science vol. 2992. Springer-Verlag. 477–494.

MIKLAU, G. AND SUCIU, D. 2004. Containment and equivalence for a fragment of XPath. *J. ACM 51*, 1, 2–45.

MURATA, M., LEE, D., MANI, M., AND KAWAGUCHI, K. 2005. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Technol. 5*, 4, 660–704.

NEVEN, F. 2002a. Automata, logic, and XML. In *CSL: Proceedings of the 16th International Workshop and 11th Annual Conference of the EACSL on Computer Science Logic*. Springer-Verlag. 2–26.

NEVEN, F. 2002b. Automata theory for XML researchers. *SIGMOD Record 31*, 3, 39–46.

NEVEN, F. AND SCHWENTICK, T. 2003. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT: Proceedings of the 9th International Conference on Database Theory*. Lecture Notes in Computer Science vol. 2572. Springer-Verlag. 315–329.

PAN, G., SATTLER, U., AND VARDI, M. Y. 2002. BDD-Based decision procedures for K. In *CADE: Proceedings of the 18th International Conference on Automated Deduction*. Springer-Verlag. 16–30.

PEMBERTON, S. 2000. XHTML 1.0 the extensible hypertext markup language (2nd ed.), W3C recommendation. http://www.w3.org/TR/xhtml1/.

SCHWENTICK, T. 2004. XPath query containment. *SIGMOD Record 33*, 1, 101–109.

SUR, G., HAMMER, J., AND SIMÉON, J. 2004. Updatex—An XQuery-Based language for processing updates in XML. In *PLAN-X: Proceedings of the International Workshop on Programming Language Technologies for XML.* Venice, Italy. BRICS Notes Series vol. NS-03-4. BRICS, Aarhus, Denmark. 40–53.

TANABE, Y., TAKAHASHI, K., YAMAMOTO, M., TOZAWA, A., AND HAGIYA, M. 2005. A decision procedure for the alternation-free two-way modal μ-calculus. In *TABLEAUX: Proceedings of the 14th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Lecture Notes in Computer Science vol. 3702. Springer-Verlag. 277–291.

THATCHER, J. W. AND WRIGHT, J. B. 1968. Generalized finite automata theory with an application to a decision problem of second-order logic. *Math. Syst. Theory 2*, 1, 57–81.

TOZAWA, A. 2001. Towards static type checking for XSLT. In *DocEng: Proceedings of the ACM Symposium on Document Engineering*. ACM, New York. 18–27.

VARDI, M. Y. 1998. Reasoning about the past with two-way automata. In *ICALP: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*. Springer-Verlag. 628–641.

WADLER, P. 2000. Two semantics for XPath. Internal Tech. Note of the W3C XSL Working Group. http://homepages.inf.ed.ac.uk/wadler/papers/xpath-semantics/xpath-semantics.pdf.

WOOD, P. T. 2000. On the equivalence of XML patterns. In *CL: Proceedings of the 1st International Conference on Computational Logic*. Lecture Notes in Computer Science vol. 1861. Springer-Verlag. 1152–1166.

WOOD, P. T. 2003. Containment for XPath fragments under DTD constraints. In *ICDT: Proceedings of the 9th International Conference on Database Theory*. Lecture Notes in Computer Science vol. 2572. Springer-Verlag. 300–314.

Received September 2005; revised June 2006; accepted August 2006