HAL
open science

Efficient Static Analysis of XML Paths and Types

Pierre Genevès, Nabil Layaïda, Alan Schmitt

To cite this version:

Pierre Genevès, Nabil Layaïda, Alan Schmitt. Efficient Static Analysis of XML Paths and Types. Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, Jun 2007, San Diego, United States. pp.342–351, 10.1145/1250734.1250773 . hal-00189123

HAL Id: hal-00189123

https://hal.science/hal-00189123v1

Submitted on 20 Nov 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Efficient Static Analysis of XML Paths and Types

Pierre Genevès
Ecole Polytechnique Fédérale de Lausanne *
pierre.geneves@epfl.ch

Nabil Layaïda Alan Schmitt
INRIA Rhône-Alpes
{nabil.layaida, alan.schmitt}@inria.fr

Abstract

We present an algorithm to solve XPath decision problems under regular tree type constraints and show its use to statically type-check XPath queries. To this end, we prove the decidability of a logic with converse for finite ordered trees whose time complexity is a simple exponential of the size of a formula. The logic corresponds to the alternation free modal μ-calculus without greatest fixpoint, restricted to finite trees, and where formulas are cycle-free.

Our proof method is based on two auxiliary results. First, XML regular tree types and XPath expressions have a linear translation to cycle-free formulas. Second, the least and greatest fixpoints are equivalent for finite trees, hence the logic is closed under negation.

Building on these results, we describe a practical, effective system for solving the satisfiability of a formula. The system has been experimented with some decision problems such as XPath emptiness, containment, overlap, and coverage, with or without type constraints. The benefit of the approach is that our system can be effectively used in static analyzers for programming languages manipulating both XPath expressions and XML type annotations (as input and output types).

Categories and Subject Descriptors E.1 [*Data Structures*]: Trees; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—modal logic; F.4.3 [*Mathematical Logic and Formal Languages*]: Formal Languages—decision problems; H.2.1 [*Database Management*]: Logical Design; H.2.3 [*Database Management*]: Languages—Query Languages

General Terms Algorithms, languages, theory, verification

Keywords Modal logic, satisfiability, type checking, XPath

1. Introduction

This work is motivated by the need of efficient type checkers for XML-based programming languages where XML types and XPath queries are used as first class language constructs. In such settings, XPath decision problems in the presence of XML types such as DTDs or XML Schemas arise naturally. Examples of such decision problems include emptiness test (whether an expression ever selects nodes), containment (whether the results of an expression are always included in the results of another one), 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).

XPath decision problems are not trivial in that they need to be checked on a possibly infinite quantification over a set of trees. Another difficulty arises from the combination of upward and downward navigation on trees with recursion [31].

The most basic decision problem for XPath is the emptiness test of an expression [3]. This test is important for optimization of host languages implementations: for instance, if one can decide at compile time that a query result is empty then subsequent bound computations can be ignored. Another basic decision problem is the XPath equivalence problem: whether or not two queries always return the same result. It is important for reformulation and optimization of an expression [17] , which aim at enforcing operational properties while preserving semantic equivalence [23]. The most essential problem for type-checking is XPath containment. It is required for the control-flow analysis of XSLT [25], for checking integrity constraints, and for XML security [12].

The complexity of XPath decision problems heavily depends on the language features. Previous works [28, 3] showed that including general comparisons of data values from an infinite domain may lead to undecidability. Therefore, we focus on a XPath fragment which covers all features except counting [8] and data values.

In our approach to solve XPath decision problems, two issues need to be addressed. First, we identify the most appropriate logic with sufficient expressiveness to capture both regular tree types and our XPath fragment. Second, we solve efficiently the satisfiability problem which allows to test if a given formula of the logic admits a satisfying finite tree.

The essence of our results lives in a sub-logic of the alternation free modal μ-calculus (AFMC) with converse, some syntactic restrictions on formulas, without greatest fixpoint, and whose models are finite trees. We prove that XPath expressions and regular tree type formulas conform to these syntactic restrictions. Boolean closure is the key property for solving the containment (a logical implication). In order to obtain closure under negation, we prove that the least and greatest fixpoint operators collapse in a single fixpoint operator. Surprisingly, the translations of XML regular tree types and a large XPath fragment does not increase complexity since they are linear in the size of the corresponding formulas in the logic. The combination of these ingredients lead to our main result: a satisfiability algorithm for a logic for finite trees whose time complexity is a simple exponential of the size of a formula.

The decision procedure has been implemented in a system for solving XML decision problems such as XPath emptiness, containment, overlap, and coverage, with or without XML type constraints. The system can be used as a component of static analyzers for programming languages manipulating XPath expressions and XML type annotations for both input and output.

* Major part of this work done when the author was at INRIA Rhône-Alpes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'07 June 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006...$5.00

2. Outline

The paper is organized as follows. We first present our data model, trees with focus, and our logic in §3 and §4. We next present XPath and its translation in our logic in §5. Our satisfiability algorithm is introduced and proven correct in §6, and a few details of the implementation are discussed in §7. Applications for type checking and some experimental results are described in §8. We study related work in §9 and conclude in §10.

Detailed proofs and implementation techniques can be found in a long version of this paper [16].

3. Trees with Focus

In order to represent XML trees that are easy to navigate we use *focused trees*, inspired by Huet's Zipper data structure [20]. Focused trees not only describe a tree but also its context: its previous siblings and its parent, including its parent context recursively. Exploring such a structure has the advantage to preserve all information, which is quite useful when considering languages such as XPath that allow forward and backward axes of navigation.

Formally, we assume an alphabet Σ of labels, ranged over by σ.

t	$::=$	$\sigma[tl]$	tree
tl	$::=$		list of trees
		ϵ	empty list
	$\mid$	$t :: tl$	cons cell
c	$::=$		context
		(tl, Top, tl)	root of the tree
	$\mid$	$(tl, c[\sigma], tl)$	context node
f	$::=$	(t, c)	focused tree

In order to deal with decision problems such as containment, we need to represent in a focused tree the place where the evaluation was started using a *start mark*, often simply called "mark" in the following. To do so, we consider focused trees where a single tree or a single context node is marked, as in $\sigma^{\textcircled{S}}[tl]$ or $(tl, c[\sigma^{\textcircled{S}}], tl)$. When the presence of the mark is unknown, we write it as $\sigma^{\circ}[tl]$. We write $\mathcal{F}$ for the set of finite focused trees with a single mark. The *name* of a focused tree is defined as $\mathtt{nm}(\sigma^{\circ}[tl], c) = \sigma$.

We now describe how to navigate focused trees, in binary style. There are four directions that can be followed: for a focused tree f, $f\,\langle 1\rangle$ changes the focus to the children of the current tree, $f\,\langle 2\rangle$ changes the focus to the next sibling of the current tree, $f\,\langle \overline{1}\rangle$ changes the focus to the parent of the tree *if the current tree is a leftmost sibling*, and $f\,\langle \overline{2}\rangle$ changes the focus to the previous sibling.

Formally, we have:

$$(\sigma^{\circ}[t :: tl], c)\,\langle 1\rangle \stackrel{\text{def}}{=} (t, (\epsilon, c[\sigma^{\circ}], tl))$$
$$(t, (tl_l, c[\sigma^{\circ}], t' :: tl_r))\,\langle 2\rangle \stackrel{\text{def}}{=} (t', (t :: tl_l, c[\sigma^{\circ}], tl_r))$$
$$(t, (\epsilon, c[\sigma^{\circ}], tl))\,\langle \overline{1}\rangle \stackrel{\text{def}}{=} (\sigma^{\circ}[t :: tl], c)$$
$$(t', (t :: tl_l, c[\sigma^{\circ}], tl_r))\,\langle \overline{2}\rangle \stackrel{\text{def}}{=} (t, (tl_l, c[\sigma^{\circ}], t' :: tl_r))$$

When the focused tree does not have the required shape, these operations are not defined.

4. The Logic

We introduce in this section the logic to which XPath expressions and XML regular tree types are going to be translated, a sub-logic of the alternation free modal μ-calculus with converse. We also introduce a restriction on the formulas we consider and give an interpretation of formulas as sets of finite focused trees. We finally show that the logic has a single fixpoint for these models and that it is closed under negation.

$\mathcal{L}_\mu \ni \varphi, \psi$	$::=$		formula
		$\top$	true
	$\mid$	$\sigma \mid \neg\sigma$	atomic prop (negated)
	$\mid$	$\textcircled{S} \mid \neg\textcircled{S}$	start prop (negated)
	$\mid$	X	variable
	$\mid$	$\varphi \vee \psi$	disjunction
	$\mid$	$\varphi \wedge \psi$	conjunction
	$\mid$	$\langle a\rangle\,\varphi \mid \neg\langle a\rangle\,\top$	existential (negated)
	$\mid$	$\mu\overline{X_i.\varphi_i}$ in ψ	least n-ary fixpoint
	$\mid$	$\nu\overline{X_i.\varphi_i}$ in ψ	greatest n-ary fixpoint

Figure 1. Logic formulas

$$[\![\top]\!]_V \stackrel{\text{def}}{=} \mathcal{F} \qquad [\![\sigma]\!]_V \stackrel{\text{def}}{=} \{f \mid \mathtt{nm}(f) = \sigma\}$$
$$[\![X]\!]_V \stackrel{\text{def}}{=} V(X) \qquad [\![\neg\sigma]\!]_V \stackrel{\text{def}}{=} \{f \mid \mathtt{nm}(f) \neq \sigma\}$$
$$[\![\varphi \vee \psi]\!]_V \stackrel{\text{def}}{=} [\![\varphi]\!]_V \cup [\![\psi]\!]_V \qquad [\![\textcircled{S}]\!]_V \stackrel{\text{def}}{=} \left\{f \mid f = (\sigma^{\textcircled{S}}[tl], c)\right\}$$
$$[\![\varphi \wedge \psi]\!]_V \stackrel{\text{def}}{=} [\![\varphi]\!]_V \cap [\![\psi]\!]_V \qquad [\![\neg\textcircled{S}]\!]_V \stackrel{\text{def}}{=} \{f \mid f = (\sigma[tl], c)\}$$

$$[\![\langle a\rangle\,\varphi]\!]_V \stackrel{\text{def}}{=} \{f\,\langle \overline{a}\rangle \mid f \in [\![\varphi]\!]_V \wedge f\,\langle \overline{a}\rangle \text{ defined}\}$$
$$[\![\neg\langle a\rangle\,\top]\!]_V \stackrel{\text{def}}{=} \{f \mid f\,\langle a\rangle \text{ undefined}\}$$
$$[\![\mu\overline{X_i.\varphi_i} \text{ in } \psi]\!]_V \stackrel{\text{def}}{=} \text{let } T_i = \left(\bigcap \left\{\overline{T_i} \subseteq \overline{\mathcal{F}} \mid [\![\overline{\varphi_i}]\!]_{V[\overline{T_i/X_i}]} \subseteq \overline{T_i}\right\}\right)_i \text{ in } [\![\psi]\!]_{V[\overline{T_i/X_i}]}$$
$$[\![\nu\overline{X_i.\varphi_i} \text{ in } \psi]\!]_V \stackrel{\text{def}}{=} \text{let } T_i = \left(\bigcup \left\{\overline{T_i} \subseteq \overline{\mathcal{F}} \mid \overline{T_i} \subseteq [\![\overline{\varphi_i}]\!]_{V[\overline{T_i/X_i}]}\right\}\right)_i \text{ in } [\![\psi]\!]_{V[\overline{T_i/X_i}]}$$

Figure 2. Interpretation of formulas

In the following definitions, $a \in \{1, 2, \overline{1}, \overline{2}\}$ are *programs* and atomic propositions σ correspond to labels from Σ. We also assume that $\overline{\overline{a}} = a$. Formulas defined in Fig. 1 include the truth predicate, atomic propositions (denoting the name of the tree in focus), start propositions (denoting the presence of the start mark), disjunction and conjunction of formulas, formulas under an existential (denoting the existence a subtree satisfying the sub-formula), and least and greatest n-ary fixpoints. We chose to include a n-ary version of fixpoints because regular types are often defined as a set of mutually recursive definitions, making their translation in our logic more succinct. In the following we write "$\mu X.\varphi$" for "$\mu X.\varphi$ in φ".

We define in Fig. 2 an interpretation of our formulas as sets of finite focused trees with a single start mark. The interpretation of the n-ary fixpoints first compute the smallest or largest interpretation for each φ_i then returns the interpretation of ψ.

We now restrict the set of valid formulas to *cycle-free formulas*, i.e. formulas that have a bound on the number of *modality cycles* independently of the number of unfolding of their fixpoints. A modality cycle is a subformula of the form $\langle a\rangle\,\varphi$ where φ contains a *top-level* existential of the form $\langle \overline{a}\rangle\,\psi$. (By "top-level" we mean under an arbitrary number of conjunctions or disjunctions, but not under any other construct.) For instance, the formula "$\mu X.\,\langle 1\rangle\,(\varphi \vee \langle \overline{1}\rangle\,X)$ in X" is not cycle free: for any integer n, there is an unfolding of the formula with n modality cycles. On the other hand, the formula "$\mu X.\,\langle 1\rangle\,(X \vee Y),\ Y.\,\langle \overline{1}\rangle\,(Y \vee \top)$ in X" is cycle free: there is at most one modality cycle.

Cycle-free formulas have a very interesting property, which we now describe. To test whether a tree satisfies a formula, one may

define a straightforward inductive relation between trees and formulas that only holds when the root of the tree satisfies the formula, unfolding fixpoints if necessary. Given a tree, if a formula φ is cycle free, then every node of the tree will be tested a finite number of time against any given subformula of φ. The intuition behind this property, which holds a central role in the proof of lemma 4.2, is the following. If a tree node is tested an infinite number of times against a subformula, then there must be a cycle in the navigation in the tree, corresponding to some modalities occurring in the subformula, between one occurrence of the test and the next one. As we consider trees, the cycle implies there is a modality cycle in the formula (as cycles of the form $\langle 1\rangle\langle 2\rangle\langle\overline{1}\rangle\langle\overline{2}\rangle$ cannot occur). Hence the number of modality cycles in any expansion of φ is unbounded, thus the formula is not cycle free.

We are now ready to show a first result: in the finite focused-tree interpretation, the least and greatest fixpoints coincide for cycle-free formulas. To this end, we prove a stronger result that states that a given focused tree is in the interpretation of a formula if it is in a finite unfolding of the formula. In the base case, we use the formula $\sigma \wedge \neg\sigma$ as "false".

DEFINITION 4.1 (Finite unfolding). *The* finite unfolding *of a formula φ is the set $unf(\varphi)$ inductively defined as*

$$\begin{aligned}
unf(\varphi) &\stackrel{def}{=} \{\varphi\} \textit{ for } \varphi = \top, \sigma, \neg\sigma, \textcircled{S}, \neg\textcircled{S}, X, \neg\langle a\rangle\top \\
unf(\varphi \vee \psi) &\stackrel{def}{=} \{\varphi' \vee \psi' \mid \varphi' \in unf(\varphi), \psi' \in unf(\psi)\} \\
unf(\varphi \wedge \psi) &\stackrel{def}{=} \{\varphi' \wedge \psi' \mid \varphi' \in unf(\varphi), \psi' \in unf(\psi)\} \\
unf(\langle a\rangle\varphi) &\stackrel{def}{=} \{\langle a\rangle\varphi' \mid \varphi' \in unf(\varphi)\} \\
unf(\mu\overline{X_i.\varphi_i} \textit{ in } \psi) &\stackrel{def}{=} unf(\psi\{^{\mu\overline{X_i.\varphi_i} \textit{ in } X_i}/_{X_i}\}) \cup \{\sigma \wedge \neg\sigma\} \\
unf(\nu\overline{X_i.\varphi_i} \textit{ in } \psi) &\stackrel{def}{=} unf(\psi\{^{\nu\overline{X_i.\varphi_i} \textit{ in } X_i}/_{X_i}\}) \cup \{\sigma \wedge \neg\sigma\}
\end{aligned}$$

LEMMA 4.2. *Let φ a cycle-free formula, then $[\![\varphi]\!]_V = [\![unf(\varphi)]\!]_V$.*

The reason why this lemma holds is the following. Given a tree satisfying φ, we deduce from the hypothesis that φ is cycle free the fact that every node of the tree will be tested a finite number of times against every subformula of φ. As the tree and the number of subformulas are finite, the satisfaction derivation is finite hence only a finite number of unfolding is necessary to prove that the tree satisfies the formula, which is what the lemma states. As least and greatest fixpoints coincide when only a finite number of unfolding is required, this is sufficient to show that they collapse. Note that this would not hold if infinite trees were allowed: the formula $\mu X.\langle 1\rangle X$ is cycle free, but its interpretation is empty, whereas the interpretation of $\nu X.\langle 1\rangle X$ includes every tree with an infinite branch of $\langle 1\rangle$ children.

We now illustrate why formulas need to be cycle free for the fixpoints to collapse. Consider the formula $\mu X.\langle 1\rangle\langle\overline{1}\rangle X$. Its interpretation is empty. The interpretation of $\nu X.\langle 1\rangle\langle\overline{1}\rangle X$ however contains every focused tree that has one $\langle 1\rangle$ child.

In the rest of the paper, we only consider least fixpoints. An important consequence of Lemma 4.2 is that the logic restricted in this way is closed under negation using De Morgan's dualities, extended to eventualities and fixpoints as follows:

$$\begin{aligned}
\neg\langle a\rangle\varphi &\stackrel{\text{def}}{=} \neg\langle a\rangle\top \vee \langle a\rangle\neg\varphi \\
\neg\mu\overline{X_i.\varphi_i} \text{ in } \psi &\stackrel{\text{def}}{=} \mu\overline{X_i.\neg\varphi_i\{^{X_i}/_{\neg X_i}\}} \text{ in } \neg\psi\{^{X_i}/_{\neg X_i}\}
\end{aligned}$$

5. XPath and Regular Tree Languages

XPath [6] is a powerful language for navigating in XML documents and selecting sets of nodes matching a predicate. In their simplest form, XPath expressions look like "directory navigation paths". For example, the XPath expression

$$/\text{child::book}/\text{child::chapter}/\text{child::section}$$

navigates from the root of a document (designated by the leading "/") through the top-level "book" node to its "chapter" child nodes and on to its child nodes named "section". The result of the evaluation of the entire expression is the set of all the "section" nodes that can be reached in this manner. The situation becomes more interesting when combined with XPath's capability of searching along "axes" other than "child". For instance, one may use the "preceding-sibling" axis for navigating backward through nodes of the same parent, or the "ancestor" axis for navigating upward recursively. Furthermore, at each step in the navigation the selected nodes can be filtered using qualifiers: boolean expression between brackets that can test the existence or absence of paths.

We consider a large XPath fragment covering all major features of the XPath recommendation [6] except counting and comparisons between data values.

Fig. 3 gives the syntax of XPath expressions. Fig. 4 gives an interpretation of XPath expressions as functions between sets of focused trees.

$\mathcal{L}_{\text{XPath}} \ni e$		::=		XPath expression
			$/p$	absolute path
		|	p	relative path
		|	$e_1 \mid e_2$	union
		|	$e_1 \cap e_2$	intersection
Path	p	::=		path
			p_1/p_2	path composition
		|	$p[q]$	qualified path
		|	$a{::}\sigma$	step with node test
		|	$a{::}*$	step
Qualif	q	::=		qualifier
			q_1 and q_2	conjunction
		|	q_1 or q_2	disjunction
		|	not q	negation
		|	p	path
Axis	a	::=		tree navigation axis
			child | self | parent	
		|	descendant | desc-or-self	
		|	ancestor | anc-or-self	
		|	foll-sibling | prec-sibling	
		|	following | preceding	

Figure 3. XPath Abstract Syntax.

5.1 XPath Embedding

We now explain how an XPath expression can be translated into an equivalent $\mathcal{L}_\mu$ formula that performs navigation in focused trees in binary style.

Logical Interpretation of Axes The translation of navigational primitives (namely XPath axes) is formally specified in Fig. 5. The translation function, noted "$A^{\rightarrow}[\![a]\!]_\chi$", takes an XPath axis a as input, and returns its $\mathcal{L}_\mu$ translation, parameterized by the $\mathcal{L}_\mu$ formula χ given as parameter. This parameter represents the context in which the axis occurs and is needed for formula composition in order to translate path composition. More precisely, the formula $A^{\rightarrow}[\![a]\!]_\chi$ holds for all nodes that can be accessed through the axis a from some node verifying χ.

Let us consider an example. The formula $A^{\rightarrow}[\![\text{child}]\!]_\chi$, translated as $\mu Z.\langle\overline{1}\rangle\chi \vee \langle\overline{2}\rangle Z$, is satisfied by children of the context χ. These nodes consist of the first child and the remaining children. From the first child, the context must be reached immediately

$$
\begin{aligned}
\mathcal{S}_e[\![\cdot]\!]_\cdot &: \mathcal{L}_{\text{XPath}} \to 2^{\mathcal{F}} \to 2^{\mathcal{F}} \\
\mathcal{S}_e[\![/p]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_p[\![p]\!]_{\mathtt{root}(F)} \\
\mathcal{S}_e[\![p]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_p[\![p]\!]_{\{(\sigma^{\textcircled{S}}[tl],c)\in F\}} \\
\mathcal{S}_e[\![e_1 \mid e_2]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_e[\![e_1]\!]_F \cup \mathcal{S}_e[\![e_2]\!]_F \\
\mathcal{S}_e[\![e_1 \cap e_2]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_e[\![e_1]\!]_F \cap \mathcal{S}_e[\![e_2]\!]_F
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}_p[\![\cdot]\!]_\cdot &: \mathit{Path} \to 2^{\mathcal{F}} \to 2^{\mathcal{F}} \\
\mathcal{S}_p[\![p_1/p_2]\!]_F &\stackrel{\text{def}}{=} \left\{ f' \mid f' \in \mathcal{S}_p[\![p_2]\!]_{(\mathcal{S}_p[\![p_1]\!]_F)} \right\} \\
\mathcal{S}_p[\![p[q]]\!]_F &\stackrel{\text{def}}{=} \{ f \mid f \in \mathcal{S}_p[\![p]\!]_F \wedge \mathcal{S}_q[\![q]\!]_f \} \\
\mathcal{S}_p[\![a{::}\sigma]\!]_F &\stackrel{\text{def}}{=} \{ f \mid f \in \mathcal{S}_a[\![a]\!]_F \wedge \mathtt{nm}(f) = \sigma \} \\
\mathcal{S}_p[\![a{::}*]\!]_F &\stackrel{\text{def}}{=} \{ f \mid f \in \mathcal{S}_a[\![a]\!]_F \}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}_q[\![\cdot]\!]_\cdot &: \mathit{Qualif} \to \mathcal{F} \to \{\text{true}, \text{false}\} \\
\mathcal{S}_q[\![q_1 \text{ and } q_2]\!]_f &\stackrel{\text{def}}{=} \mathcal{S}_q[\![q_1]\!]_f \wedge \mathcal{S}_q[\![q_2]\!]_f \\
\mathcal{S}_q[\![q_1 \text{ or } q_2]\!]_f &\stackrel{\text{def}}{=} \mathcal{S}_q[\![q_1]\!]_f \vee \mathcal{S}_q[\![q_2]\!]_f \\
\mathcal{S}_q[\![\text{not } q]\!]_f &\stackrel{\text{def}}{=} \neg\, \mathcal{S}_q[\![q]\!]_f \\
\mathcal{S}_q[\![p]\!]_f &\stackrel{\text{def}}{=} \mathcal{S}_p[\![p]\!]_{\{f\}} \neq \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}_a[\![\cdot]\!]_\cdot &: \mathit{Axis} \to 2^{\mathcal{F}} \to 2^{\mathcal{F}} \\
\mathcal{S}_a[\![\text{self}]\!]_F &\stackrel{\text{def}}{=} F \\
\mathcal{S}_a[\![\text{child}]\!]_F &\stackrel{\text{def}}{=} \mathtt{fchild}(F) \cup \mathcal{S}_a[\![\text{foll-sibling}]\!]_{\mathtt{fchild}(F)} \\
\mathcal{S}_a[\![\text{foll-sibling}]\!]_F &\stackrel{\text{def}}{=} \mathtt{nsibling}(F) \cup \mathcal{S}_a[\![\text{foll-sibling}]\!]_{\mathtt{nsibling}(F)} \\
\mathcal{S}_a[\![\text{prec-sibling}]\!]_F &\stackrel{\text{def}}{=} \mathtt{psibling}(F) \cup \mathcal{S}_a[\![\text{prec-sibling}]\!]_{\mathtt{psibling}(F)} \\
\mathcal{S}_a[\![\text{parent}]\!]_F &\stackrel{\text{def}}{=} \mathtt{parent}(F) \\
\mathcal{S}_a[\![\text{descendant}]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_a[\![\text{child}]\!]_F \cup \mathcal{S}_a[\![\text{descendant}]\!]_{(\mathcal{S}_a[\![\text{child}]\!]_F)} \\
\mathcal{S}_a[\![\text{desc-or-self}]\!]_F &\stackrel{\text{def}}{=} F \cup \mathcal{S}_a[\![\text{descendant}]\!]_F \\
\mathcal{S}_a[\![\text{ancestor}]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_a[\![\text{parent}]\!]_F \cup \mathcal{S}_a[\![\text{ancestor}]\!]_{(\mathcal{S}_a[\![\text{parent}]\!]_F)} \\
\mathcal{S}_a[\![\text{anc-or-self}]\!]_F &\stackrel{\text{def}}{=} F \cup \mathcal{S}_a[\![\text{ancestor}]\!]_F \\
\mathcal{S}_a[\![\text{following}]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_a[\![\text{desc-or-self}]\!]_{\left(\mathcal{S}_a[\![\text{foll-sibling}]\!]_{(\mathcal{S}_a[\![\text{anc-or-self}]\!]_F)}\right)} \\
\mathcal{S}_a[\![\text{preceding}]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_a[\![\text{desc-or-self}]\!]_{\left(\mathcal{S}_a[\![\text{prec-sibling}]\!]_{(\mathcal{S}_a[\![\text{anc-or-self}]\!]_F)}\right)} \\
\mathtt{fchild}(F) &\stackrel{\text{def}}{=} \{ f\,\langle 1\rangle \mid f \in F \wedge f\,\langle 1\rangle \text{ defined} \} \\
\mathtt{nsibling}(F) &\stackrel{\text{def}}{=} \{ f\,\langle 2\rangle \mid f \in F \wedge f\,\langle 2\rangle \text{ defined} \} \\
\mathtt{psibling}(F) &\stackrel{\text{def}}{=} \left\{ f\,\langle \overline{2}\rangle \mid f \in F \wedge f\,\langle \overline{2}\rangle \text{ defined} \right\} \\
\mathtt{parent}(F) &\stackrel{\text{def}}{=} \{ (\sigma^\circ[\mathtt{rev_a}(tl_l, t :: tl_r)], c) \\
&\qquad \mid (t, (tl_l, c[\sigma^\circ], tl_r)) \in F \} \\
\mathtt{rev_a}(\epsilon, tl_r) &\stackrel{\text{def}}{=} tl_r \\
\mathtt{rev_a}(t :: tl_l, tl_r) &\stackrel{\text{def}}{=} \mathtt{rev_a}(tl_l, t :: tl_r) \\
\mathtt{root}(F) &\stackrel{\text{def}}{=} \{ (\sigma^{\textcircled{S}}[tl], (tl, \mathit{Top}, tl)) \in F \} \\
&\qquad \cup\, \mathtt{root}(\mathtt{parent}(F))
\end{aligned}
$$

Figure 4. Interpretation of XPath in terms of Focused Trees.

$$
\begin{aligned}
A^{\rightarrow}[\![\cdot]\!]_\cdot &: \mathit{Axis} \to \mathcal{L}_\mu \to \mathcal{L}_\mu \\
A^{\rightarrow}[\![\text{self}]\!]_\chi &\stackrel{\text{def}}{=} \chi \\
A^{\rightarrow}[\![\text{child}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\, \langle \overline{1}\rangle\, \chi \vee \langle \overline{2}\rangle\, Z \\
A^{\rightarrow}[\![\text{foll-sibling}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\, \langle \overline{2}\rangle\, \chi \vee \langle \overline{2}\rangle\, Z \\
A^{\rightarrow}[\![\text{prec-sibling}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\, \langle 2\rangle\, \chi \vee \langle 2\rangle\, Z \\
A^{\rightarrow}[\![\text{parent}]\!]_\chi &\stackrel{\text{def}}{=} \langle 1\rangle\, \mu Z.\chi \vee \langle 2\rangle\, Z \\
A^{\rightarrow}[\![\text{descendant}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\, \langle \overline{1}\rangle\, (\chi \vee Z) \vee \langle \overline{2}\rangle\, Z \\
A^{\rightarrow}[\![\text{desc-or-self}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\chi \vee \mu Y.\, \langle \overline{1}\rangle\, (Y \vee Z) \vee \langle \overline{2}\rangle\, Y \\
A^{\rightarrow}[\![\text{ancestor}]\!]_\chi &\stackrel{\text{def}}{=} \langle 1\rangle\, \mu Z.\chi \vee \langle 1\rangle\, Z \vee \langle 2\rangle\, Z \\
A^{\rightarrow}[\![\text{anc-or-self}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\chi \vee \langle 1\rangle\, \mu Y.Z \vee \langle 2\rangle\, Y \\
A^{\rightarrow}[\![\text{following}]\!]_\chi &\stackrel{\text{def}}{=} A^{\rightarrow}[\![\text{desc-or-self}]\!]_{\eta_1} \\
A^{\rightarrow}[\![\text{preceding}]\!]_\chi &\stackrel{\text{def}}{=} A^{\rightarrow}[\![\text{desc-or-self}]\!]_{\eta_2} \\
\eta_1 &\stackrel{\text{def}}{=} A^{\rightarrow}[\![\text{foll-sibling}]\!]_{A^{\rightarrow}[\![\text{anc-or-self}]\!]_\chi} \\
\eta_2 &\stackrel{\text{def}}{=} A^{\rightarrow}[\![\text{prec-sibling}]\!]_{A^{\rightarrow}[\![\text{anc-or-self}]\!]_\chi}
\end{aligned}
$$

Figure 5. Translation of XPath Axes.

$$
\begin{aligned}
E^{\rightarrow}[\![\cdot]\!]_\cdot &: \mathcal{L}_{\text{XPath}} \to \mathcal{L}_\mu \to \mathcal{L}_\mu \\
E^{\rightarrow}[\![/p]\!]_\chi &\stackrel{\text{def}}{=} P^{\rightarrow}[\![p]\!]_{\left((\mu Z.\neg\langle \overline{1}\rangle\top \vee \langle \overline{2}\rangle Z) \wedge (\mu Y.\chi \wedge \textcircled{S} \vee \langle 1\rangle Y \vee \langle 2\rangle Y)\right)} \\
E^{\rightarrow}[\![p]\!]_\chi &\stackrel{\text{def}}{=} P^{\rightarrow}[\![p]\!]_{(\chi \wedge \textcircled{S})} \\
E^{\rightarrow}[\![e_1 \mid e_2]\!]_\chi &\stackrel{\text{def}}{=} E^{\rightarrow}[\![e_1]\!]_\chi \vee E^{\rightarrow}[\![e_2]\!]_\chi \\
E^{\rightarrow}[\![e_1 \cap e_2]\!]_\chi &\stackrel{\text{def}}{=} E^{\rightarrow}[\![e_1]\!]_\chi \wedge E^{\rightarrow}[\![e_2]\!]_\chi
\end{aligned}
$$

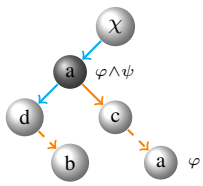
$$
\begin{aligned}
P^{\rightarrow}[\![\cdot]\!]_\cdot &: \mathit{Path} \to \mathcal{L}_\mu \to \mathcal{L}_\mu \\
P^{\rightarrow}[\![p_1/p_2]\!]_\chi &\stackrel{\text{def}}{=} P^{\rightarrow}[\![p_2]\!]_{(P^{\rightarrow}[\![p_1]\!]_\chi)} \\
P^{\rightarrow}[\![p[q]]\!]_\chi &\stackrel{\text{def}}{=}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

Figure 6. Translation of Expressions and Paths.

by going once upward via $\overline{1}$. From the remaining children, the context is reached by going upward (any number of times) via $\overline{2}$ and then finally once via $\overline{1}$.

Logical Interpretation of Expressions Fig. 6 gives the translation of XPath expressions into $\mathcal{L}_\mu$. The translation function "$E^{\rightarrow}[\![e]\!]_\chi$" takes an XPath expression e and a $\mathcal{L}_\mu$ formula χ as input, and returns the corresponding $\mathcal{L}_\mu$ translation. The translation of a relative XPath expression marks the initial context with $\textcircled{S}$. The translation of an absolute XPath expression navigates to the root which is taken as the initial context.

Figure 7 illustrates the translation of the XPath expression "child::a[child::b]". 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. The first part of the translated formula, φ, corresponds to the step



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

$$\underbrace{a \wedge (\mu X.\, \langle \overline{1} \rangle\, (\chi \wedge \textcircled{s}) \vee \langle \overline{2} \rangle\, X)}_{\varphi} \;\wedge\; \underbrace{\langle 1 \rangle\, \mu Y.b \vee \langle 2 \rangle\, Y}_{\psi}$$

Figure 7. XPath Translation Example.

"child::a" which selects candidates "a" nodes. The second part, ψ, navigates downward in the subtrees of these candidate nodes to verify that they have at least one immediate "b" child.

Note that without converse programs we would have been unable to differentiate selected nodes from nodes whose existence is tested: we must state properties on both the ancestors and the descendants of the selected node. Equipping the $\mathcal{L}_\mu$ logic with both forward and converse programs is therefore crucial for supporting XPath. Logics without converse programs may only be used for solving XPath emptiness but cannot be used for solving other decision problems such as containment efficiently.

XPath composition 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 through p_1 from χ. 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. 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 or not. We express this by introducing a dual formal translation function for XPath qualifiers, noted $Q^{\leftarrow}[\![q]\!]_{\cdot}$ and defined in Fig. 8, that 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 opposite, $Q^{\leftarrow}[\![\cdot]\!]_{\cdot}$ can be seen as the "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 is based on these two translating "modes", the first one being used for paths and the second one for qualifiers. Whenever the "filtering" mode is entered, it will never be left.

The translation of paths inside qualifiers is also given in Fig. 8. It uses the translation for axes and is based on XPath symmetry: *symmetric*(a) denotes the symmetric XPath axis corresponding to the axis a (for instance *symmetric*(child) = parent).

We may now state that our translation is correct, by relating the interpretation of an XPath formula applied to some set of trees to the interpretation of its translation, by stating that the translation of a formula is cycle-free, and by giving a bound in the size of this translation.

We restrict the sets of trees to which an XPath formula may be applied to those that may be denoted by an $\mathcal{L}_\mu$ formula. This restriction will be justified in Section 5.2 where we show that every regular tree language may be translated to an $\mathcal{L}_\mu$ formula.

PROPOSITION 5.1 (Translation Correctness). *The following hold for an XPath expression e and a $\mathcal{L}_\mu$ formula φ denoting a set of focused trees, with $\psi = E^{\rightarrow}[\![e]\!]_\varphi$:*

$$Q^{\leftarrow}[\![\cdot]\!]_{\cdot} : \mathit{Qualif} \rightarrow \mathcal{L}_\mu \rightarrow \mathcal{L}_\mu$$
$$Q^{\leftarrow}[\![q_1 \text{ and } q_2]\!]_\chi \stackrel{\text{def}}{=} Q^{\leftarrow}[\![q_1]\!]_\chi \wedge Q^{\leftarrow}[\![q_2]\!]_\chi$$
$$Q^{\leftarrow}[\![q_1 \text{ or } q_2]\!]_\chi \stackrel{\text{def}}{=} Q^{\leftarrow}[\![q_1]\!]_\chi \vee Q^{\leftarrow}[\![q_2]\!]_\chi$$
$$Q^{\leftarrow}[\![\text{not } q]\!]_\chi \stackrel{\text{def}}{=} \neg\, Q^{\leftarrow}[\![q]\!]_\chi$$
$$Q^{\leftarrow}[\![p]\!]_\chi \stackrel{\text{def}}{=} P^{\leftarrow}[\![p]\!]_\chi$$

$$P^{\leftarrow}[\![\cdot]\!]_{\cdot} : \mathit{Path} \rightarrow \mathcal{L}_\mu \rightarrow \mathcal{L}_\mu$$
$$P^{\leftarrow}[\![p_1/p_2]\!]_\chi \stackrel{\text{def}}{=} P^{\leftarrow}[\![p_1]\!]_{(P^{\leftarrow}[\![p_2]\!]_\chi)}$$
$$P^{\leftarrow}[\![p[q]]\!]_\chi \stackrel{\text{def}}{=} P^{\leftarrow}[\![p]\!]_{(\chi \wedge Q^{\leftarrow}[\![q]\!]_\top)}$$
$$P^{\leftarrow}[\![a{::}\sigma]\!]_\chi \stackrel{\text{def}}{=} A^{\leftarrow}[\![a]\!]_{(\chi \wedge \sigma)}$$
$$P^{\leftarrow}[\![a{::}*]\!]_\chi \stackrel{\text{def}}{=} A^{\leftarrow}[\![a]\!]_\chi$$

$$A^{\leftarrow}[\![\cdot]\!]_{\cdot} : \mathit{Axis} \rightarrow \mathcal{L}_\mu \rightarrow \mathcal{L}_\mu$$
$$A^{\leftarrow}[\![a]\!]_\chi \stackrel{\text{def}}{=} A^{\rightarrow}[\![\mathit{symmetric}(a)]\!]_\chi$$

Figure 8. Translation of Qualifiers.

1. $[\![\psi]\!]_\emptyset = \mathcal{S}_e[\![e]\!]_{[\![\varphi]\!]_\emptyset}$
2. *ψ is cycle-free*
3. *the size of ψ is linear in the size of e and φ*

5.2 Embedding Regular Tree Languages

Several formalisms exist for describing types of XML documents (e.g. DTD, XML Schema, Relax NG). In this paper we embed regular tree languages, which gather all of them [26] into $\mathcal{L}_\mu$. We rely on a straightforward isomorphism between unranked regular tree types and binary regular tree types [19]. Assuming a countably infinite set of type variables ranged over by X, binary regular tree type expressions are defined as follows:

$\mathcal{L}_{BT} \ni T$	::=		tree type expression
		$\emptyset$	empty set
	|	ϵ	leaf
	|	$T_1 \mid T_2$	union
	|	$\sigma(X_1, X_2)$	label
	|	let $\overline{X_i.T_i}$ in T	binder

We refer the reader to [19] for the denotational semantics of regular tree languages, and directly introduce their translation into $\mathcal{L}_\mu$:

$$[\![\cdot]\!] : \mathcal{L}_{BT} \rightarrow \mathcal{L}_\mu$$
$$[\![T]\!] \stackrel{\text{def}}{=} \sigma \wedge \neg\sigma \quad \text{for} \quad T = \emptyset, \epsilon$$
$$[\![T_1 \mid T_2]\!] \stackrel{\text{def}}{=} [\![T_1]\!] \vee [\![T_2]\!]$$
$$[\![\sigma(X_1, X_2)]\!] \stackrel{\text{def}}{=} \sigma \wedge \mathit{succ}_1(X_1) \wedge \mathit{succ}_2(X_2)$$
$$[\![\text{let } \overline{X_i.T_i} \text{ in } T]\!] \stackrel{\text{def}}{=} \mu \overline{X_i.[\![T_i]\!]} \text{ in } [\![T]\!]$$

where we use the formula $\sigma \wedge \neg\sigma$ as "false", and the function $\mathit{succ}_\cdot(\cdot)$ takes care of setting the type frontier:

$$\mathit{succ}_\alpha(X) \quad = \quad \begin{cases} \neg\langle\alpha\rangle\top & \textit{if } X \textit{ is bound to } \epsilon \\ \neg\langle\alpha\rangle\top \vee \langle\alpha\rangle X & \textit{if nullable}(X) \\ \langle\alpha\rangle X & \textit{if not nullable}(X) \end{cases}$$

according to the predicate *nullable*(X) which indicates whether the type $T \neq \epsilon$ bound to X contains the empty tree.

Note that the translation of a regular tree type uses only downward modalities since it describes the allowed subtrees at a given context. No additional restriction is imposed on the context from which the type definition starts. In particular, navigation is allowed in the upward direction so that we can support type constraints for which we have only partial knowledge in a given direction. However, when we know the position of the root, conditions similar to those of absolute paths are added in the form of additional formulas describing the position that need to be satisfied. This is particularly useful when a regular type is used by an XPath expression that starts its navigation at the root ($/p$) since the path will not go above the root of the type (by adding the restriction $\mu Z.\neg\langle\overline{1}\rangle\top\vee\langle\overline{2}\rangle Z$).

On the other hand, if the type is compared with another type (typically to check inclusion of the result of an XPath expression in this type), then there is no restriction as to where the root of the type is (our translation does not impose the chosen node to be at the root). This is particularly useful since an XPath expression usually returns a set of nodes deep in the tree which we may compare to this partially defined type.

6. Satisfiability-Testing Algorithm

In this section we present our algorithm, show that it is sound and complete, and prove a time complexity boundary. To check a formula φ, our algorithm builds satisfiable formulas out of some subformulas (and their negation) of φ, then checks whether φ was produced. We first describe how to extract the subformulas from φ.

6.1 Preliminary Definitions

For $\varphi = (\mu\overline{X_i.\varphi_i} \text{ in } \psi)$ we define $\exp(\varphi) \stackrel{\text{def}}{=} \psi\{\overline{\mu\overline{X_i.\varphi_i} \text{ in } X_i/X_i}\}$ which denotes the formula ψ in which every occurrence of a X_i is replaced by $(\mu\overline{X_i.\varphi_i} \text{ in } X_i)$.

We define the *Fisher-Ladner closure* $\mathrm{cl}(\psi)$ of a formula ψ as the set of all subformulas of ψ where fixpoint formulas are additionally unwound once. Specifically, we define the relation $\rightarrow_e \subseteq \mathcal{L}_\mu \times \mathcal{L}_\mu$ as the least relation that satisfies the following:

- $\varphi_1 \wedge \varphi_2 \rightarrow_e \varphi_1, \varphi_1 \wedge \varphi_2 \rightarrow_e \varphi_2$
- $\varphi_1 \vee \varphi_2 \rightarrow_e \varphi_1, \varphi_1 \vee \varphi_2 \rightarrow_e \varphi_2$
- $\langle a\rangle\,\varphi' \rightarrow_e \varphi'$
- $\mu\overline{X_i.\varphi_i} \text{ in } \psi \rightarrow_e \exp(\mu\overline{X_i.\varphi_i} \text{ in } \psi)$

The closure $\mathrm{cl}(\psi)$ is the smallest set S that contains ψ and closed under the relation $\rightarrow_e$, i.e. if $\varphi_1 \in S$ and $\varphi_1 \rightarrow_e \varphi_2$ then $\varphi_2 \in S$.

We call $\Sigma(\psi)$ the set of atomic propositions σ used in ψ along with another name, σ_x, that does not occur in ψ to represent atomic propositions not occurring in ψ.

We define $\mathrm{cl}^*(\psi) = \mathrm{cl}(\psi) \cup \{\neg\varphi \mid \varphi \in \mathrm{cl}(\psi)\}$. Every formula $\varphi \in \mathrm{cl}^*(\psi)$ can be seen as a boolean combination of formulas of a set called the Lean of ψ, inspired from [27]. We note this set $\mathrm{Lean}(\psi)$ and define it as follows:

$$\mathrm{Lean}(\psi) = \left\{\langle a\rangle\top \mid a \in \{1,2,\overline{1},\overline{2}\}\right\} \cup \Sigma(\psi) \cup \{\text{Ⓢ}\} \cup \{\langle a\rangle\,\varphi \mid \langle a\rangle\,\varphi \in \mathrm{cl}(\psi)\}$$

A ψ-*type* (or simply a "*type*") (Hintikka set in the temporal logic literature) is a set $t \subseteq \mathrm{Lean}(\psi)$ such that:

- $\forall\,\langle a\rangle\,\varphi \in \mathrm{Lean}(\psi), \langle a\rangle\,\varphi \in t \Rightarrow \langle a\rangle\top \in t$ (modal consistency);
- $\langle\overline{1}\rangle\top \notin t \vee \langle\overline{2}\rangle\top \notin t$ (a tree node cannot be both a first child and a second child);
- exactly one atomic proposition $\sigma \in t$ (XML labeling); we use the function $\sigma(t)$ to return the atomic proposition of a type t;
- Ⓢ may belong to t.

$$\frac{}{\top \dot{\in} t \Longrightarrow (\emptyset,\emptyset)} \qquad \frac{\varphi \in \mathrm{Lean}(\psi) \qquad \varphi \in t}{\varphi \dot{\in} t \Longrightarrow (\{\varphi\},\emptyset)}$$

$$\frac{\varphi_1 \dot{\in} t \Longrightarrow (T_1,F_1) \qquad \varphi_2 \dot{\in} t \Longrightarrow (T_2,F_2)}{\varphi_1 \wedge \varphi_2 \dot{\in} t \Longrightarrow (T_1 \cup T_2, F_1 \cup F_2)}$$

$$\frac{\varphi_1 \dot{\in} t \Longrightarrow (T_1,F_1)}{\varphi_1 \vee \varphi_2 \dot{\in} t \Longrightarrow (T_1,F_1)} \qquad \frac{\varphi_2 \dot{\in} t \Longrightarrow (T_2,F_2)}{\varphi_1 \vee \varphi_2 \dot{\in} t \Longrightarrow (T_2,F_2)}$$

$$\frac{\varphi \dot{\notin} t \Longrightarrow (T,F)}{\neg\varphi \dot{\in} t \Longrightarrow (T,F)} \qquad \frac{\exp(\mu\overline{X_i.\varphi_i} \text{ in } \psi) \dot{\in} t \Longrightarrow (T,F)}{\mu\overline{X_i.\varphi_i} \text{ in } \psi \dot{\in} t \Longrightarrow (T,F)}$$

$$\frac{\varphi \in \mathrm{Lean}(\psi) \qquad \varphi \notin t}{\varphi \dot{\notin} t \Longrightarrow (\emptyset,\{\varphi\})}$$

$$\frac{\varphi_1 \dot{\notin} t \Longrightarrow (T_1,F_1) \qquad \varphi_2 \dot{\notin} t \Longrightarrow (T_2,F_2)}{\varphi_1 \vee \varphi_2 \dot{\notin} t \Longrightarrow (T_1 \cup T_2, F_1 \cup F_2)}$$

$$\frac{\varphi_1 \dot{\notin} t \Longrightarrow (T_1,F_1)}{\varphi_1 \wedge \varphi_2 \dot{\notin} t \Longrightarrow (T_1,F_1)} \qquad \frac{\varphi_2 \dot{\notin} t \Longrightarrow (T_2,F_2)}{\varphi_1 \wedge \varphi_2 \dot{\notin} t \Longrightarrow (T_2,F_2)}$$

$$\frac{\varphi \dot{\in} t \Longrightarrow (T,F)}{\neg\varphi \dot{\notin} t \Longrightarrow (T,F)} \qquad \frac{\exp(\mu\overline{X_i.\varphi_i} \text{ in } \psi) \dot{\notin} t \Longrightarrow (T,F)}{\mu\overline{X_i.\varphi_i} \text{ in } \psi \dot{\notin} t \Longrightarrow (T,F)}$$

Figure 9. Truth assignment of a formula

We call $\mathrm{Types}(\psi)$ the set of ψ-types. For a ψ-type t, the *complement* of t is the set $\mathrm{Lean}(\psi) \setminus t$.

A type determines a truth assignment of every formula in $\mathrm{cl}^*(\psi)$ with the relation $\dot{\in}$ defined in Fig. 9. Note that such derivations are finite because the number of naked $\mu\overline{X_i.\varphi_i}$ in ψ (that do not occur under modalities) strictly decreases after each expansion.

We often write $\varphi \dot{\in} t$ if there are some T, F such that $\varphi \dot{\in} t \Longrightarrow (T,F)$. We say that a formula φ is true at a type t iff $\varphi \dot{\in} t$.

We now relate a formula to the truth assignment of its ψ-types.

PROPOSITION 6.1. *If $\varphi \dot{\in} t \Longrightarrow (T,F)$, then we have $T \subseteq t$, $F \subseteq$ Lean$(\varphi) \setminus t$, and $\bigwedge_{\psi\in T}\psi \wedge \bigwedge_{\psi\in F}\neg\psi$ implies φ (every tree in the interpretation of the first formula is in the interpretation of the second). If $\varphi \dot{\notin} t \Longrightarrow (T,F)$, then we have $T \subseteq t$, $F \subseteq$ Lean$(\varphi) \setminus t$, and $\bigwedge_{\psi\in T}\psi \wedge \bigwedge_{\psi\in F}\neg\psi$ implies $\neg\varphi$.*

We next define a compatibility relation between types to state that two types are related according to a modality.

DEFINITION 6.2 (Compatibility relation). *Two types t and t' are* compatible *under $a \in \{1,2\}$, written $\Delta_a(t,t')$, iff*

$$\forall\,\langle a\rangle\,\varphi \in \mathit{Lean}(\psi), \langle a\rangle\,\varphi \in t \Leftrightarrow \varphi \dot{\in} t'$$
$$\forall\,\langle \overline{a}\rangle\,\varphi \in \mathit{Lean}(\psi), \langle \overline{a}\rangle\,\varphi \in t' \Leftrightarrow \varphi \dot{\in} t$$

6.2 The Algorithm

The algorithm works on sets of triples of the form (t, w_1, w_2) where t is a type, and w_1 and w_2 are sets of types which represent every witness for t according to relations $\Delta_1(t,\cdot)$ and $\Delta_2(t,\cdot)$.

The algorithm proceeds in a bottom-up approach, repeatedly adding new triples until a satisfying model is found (i.e. a triple whose first component is a type implying the formula), or until no

$$\begin{aligned}\texttt{Upd}(X) \stackrel{\text{def}}{=} X \cup \{(t, \texttt{w}_1(t, X^\circ), \texttt{w}_2(t, X^\circ)) \mid \textcircled{S} \notin t \subseteq \text{Types}(\psi) \wedge \langle 1\rangle \top \in t \Rightarrow \texttt{w}_1(t, X^\circ) \neq \emptyset \wedge \langle 2\rangle \top \in t \Rightarrow \texttt{w}_2(t, X^\circ) \neq \emptyset\}\\ \cup \left\{(t, \texttt{w}_1(t, X^\circ), \texttt{w}_2(t, X^\circ))^{\textcircled{S}} \mid \textcircled{S} \in t \subseteq \text{Types}(\psi) \wedge \langle 1\rangle \top \in t \Rightarrow \texttt{w}_1(t, X^\circ) \neq \emptyset \wedge \langle 2\rangle \top \in t \Rightarrow \texttt{w}_2(t, X^\circ) \neq \emptyset\right\}\\ \cup \left\{(t, \texttt{w}_1(t, X^{\textcircled{S}}), \texttt{w}_2(t, X^\circ))^{\textcircled{S}} \mid \textcircled{S} \notin t \subseteq \text{Types}(\psi) \wedge \langle 1\rangle \top \in t \Rightarrow \texttt{w}_1(t, X^{\textcircled{S}}) \neq \emptyset \wedge \langle 2\rangle \top \in t \Rightarrow \texttt{w}_2(t, X^\circ) \neq \emptyset\right\}\\ \cup \left\{(t, \texttt{w}_1(t, X^\circ), \texttt{w}_2(t, X^{\textcircled{S}}))^{\textcircled{S}} \mid \textcircled{S} \notin t \subseteq \text{Types}(\psi) \wedge \langle 1\rangle \top \in t \Rightarrow \texttt{w}_1(t, X^\circ) \neq \emptyset \wedge \langle 2\rangle \top \in t \Rightarrow \texttt{w}_2(t, X^{\textcircled{S}}) \neq \emptyset\right\}\end{aligned}$$

$$\texttt{w}_a(t, X) \stackrel{\text{def}}{=} \{\texttt{type}(x) \mid x \in X \wedge \langle \overline{a}\rangle \top \in \texttt{type}(x) \wedge \Delta_a(t, \texttt{type}(x))\} \qquad X^{\textcircled{S}} \stackrel{\text{def}}{=} \left\{x \in X \mid x = (_, _, _)^{\textcircled{S}}\right\}$$

$$\texttt{FinalCheck}(\psi, X) \stackrel{\text{def}}{=} \exists x \in X^{\textcircled{S}}, \texttt{dsat}(x, \psi) \wedge \forall a \in \{\overline{1}, \overline{2}\}, \langle a\rangle \top \notin \texttt{type}(x) \qquad X^\circ \stackrel{\text{def}}{=} \{x \in X \mid x = (_, _, _)\}$$

$$\texttt{dsat}((t, w_1, w_2), \psi) \stackrel{\text{def}}{=} \psi \dot{\in} t \vee \exists x', \texttt{dsat}(x', \psi) \wedge (x' \in w_1 \vee x' \in w_2) \qquad \texttt{type}((t, w_1, w_2)) \stackrel{\text{def}}{=} t$$

Figure 10. Operations used by the Algorithm.

more triple can be added. Each iteration of the algorithm builds types representing deeper trees (in the 1 and 2 direction) with pending backward modalities that will be fulfilled at later iterations. Types with no backward modalities are satisfiable, and if such a type implies the formula being tested, then it is satisfiable. The main iteration is as follows:

$X \leftarrow \emptyset$
repeat
  $X' \leftarrow X$
  $X \leftarrow \texttt{Upd}(X')$
  if $\texttt{FinalCheck}(\psi, X)$ **then**
    return "ψ is satisfiable"
until $X = X'$
return "ψ is unsatisfiable"

where $X \subseteq \text{Types}(\psi) \times 2^{\text{Types}(\psi)} \times 2^{\text{Types}(\psi)}$ and the operations $\texttt{Upd}(\cdot)$ and $\texttt{FinalCheck}(\cdot)$ are defined on Fig. 10.

We note X^i the set of triples and T^i the set of types after i iterations: $T^i = \left\{\texttt{type}(x) \mid x \in X^i\right\}$. Note that T^{i+1} is the set of types for which at least one witness belongs to T^i.

6.3 Correctness and Complexity

In this section we define the necessary notions to prove the correctness of the satisfiability testing algorithm, and show that its time complexity is $2^{O(|\text{Lean}(\psi)|)}$.

THEOREM 6.3 (Correctness). *The algorithm decides satisfiability of $\mathcal{L}_\mu$ formulas over finite focused trees.*

Termination For $\psi \in \mathcal{L}_\mu$, since $\text{cl}(\psi)$ is a finite set, $\text{Lean}(\psi)$ and $2^{\text{Lean}(\psi)}$ are also finite. Furthermore, $\texttt{Upd}(\cdot)$ is monotonic and each X^i is included in the finite set $\text{Types}(\psi) \times 2^{\text{Types}(\psi)} \times 2^{\text{Types}(\psi)}$, therefore the algorithm terminates. To finish the proof, it thus suffices to prove soundness and completeness.

Preliminary Definitions for Soundness First, we introduce a notion of partial satisfiability for a formula, where backward modalities are only checked up to a given level. A formula φ is partially satisfied iff $[\![\varphi]\!]^0_V \neq \emptyset$ as defined in Fig. 11.

For a type t, we note $\varphi_c(t)$ its *most constrained formula*, where atoms are taken from $\text{Lean}(\psi)$. In the following, $\circ$ stands for $\textcircled{S}$ if $\textcircled{S} \in t$, and for $\neg\textcircled{S}$ otherwise.

$$\varphi_c(t) = \sigma(t) \wedge \bigwedge_{\sigma \in \Sigma, \sigma \notin t} \neg\sigma \wedge \circ \wedge \bigwedge_{\langle a\rangle \varphi \in t} \langle a\rangle \varphi \wedge \bigwedge_{\langle a\rangle \varphi \notin t} \neg \langle a\rangle \varphi$$

We now introduce a notion of *paths*, written ρ which are concatenations of modalities: the empty path is written ϵ, and path concatenation is written ρa.

$$\begin{aligned}
[\![\top]\!]^n_V &\stackrel{\text{def}}{=} \mathcal{F} & [\![X]\!]^n_V &\stackrel{\text{def}}{=} V(X)\\
[\![\varphi \vee \psi]\!]^n_V &\stackrel{\text{def}}{=} [\![\varphi]\!]^n_V \cup [\![\psi]\!]^n_V & [\![p]\!]^n_V &\stackrel{\text{def}}{=} \{f \mid \texttt{nm}(f) = p\}\\
[\![\varphi \wedge \psi]\!]^n_V &\stackrel{\text{def}}{=} [\![\varphi]\!]^n_V \cap [\![\psi]\!]^n_V & [\![\neg p]\!]^n_V &\stackrel{\text{def}}{=} \{f \mid \texttt{nm}(f) \neq p\}\\
[\![\langle\overline{1}\rangle \varphi]\!]^0_V &\stackrel{\text{def}}{=} \mathcal{F} & [\![\textcircled{S}]\!]^n_V &\stackrel{\text{def}}{=} \left\{f \mid f = (\sigma^{\textcircled{S}}[tl], c)\right\}\\
[\![\langle\overline{2}\rangle \varphi]\!]^0_V &\stackrel{\text{def}}{=} \mathcal{F} & [\![\neg\textcircled{S}]\!]^n_V &\stackrel{\text{def}}{=} \{f \mid f = (\sigma[tl], c)\}
\end{aligned}$$

$$\begin{aligned}
[\![\langle\overline{1}\rangle \varphi]\!]^{n>0}_V &\stackrel{\text{def}}{=} \left\{f\langle 1\rangle \mid f \in [\![\varphi]\!]^{n-1}_V \wedge f\langle 1\rangle \text{ defined}\right\}\\
[\![\langle\overline{2}\rangle \varphi]\!]^{n>0}_V &\stackrel{\text{def}}{=} \left\{f\langle 2\rangle \mid f \in [\![\varphi]\!]^{n-1}_V \wedge f\langle 2\rangle \text{ defined}\right\}\\
[\![\langle 1\rangle \varphi]\!]^n_V &\stackrel{\text{def}}{=} \left\{f\langle\overline{1}\rangle \mid f \in [\![\varphi]\!]^{n+1}_V \wedge f\langle\overline{1}\rangle \text{ defined}\right\}\\
[\![\langle 2\rangle \varphi]\!]^n_V &\stackrel{\text{def}}{=} \left\{f\langle\overline{2}\rangle \mid f \in [\![\varphi]\!]^{n+1}_V \wedge f\langle\overline{2}\rangle \text{ defined}\right\}\\
[\![\neg\langle a\rangle \top]\!]^n_V &\stackrel{\text{def}}{=} \{f \mid f\langle a\rangle \text{ undefined}\}\\
[\![\mu\overline{X_i.\varphi_i} \text{ in } \psi]\!]^n_V &\stackrel{\text{def}}{=} \text{let } T_i = \left(\bigcap \left\{\overline{T_i} \subseteq \overline{\mathcal{F}} \mid [\![\overline{\varphi_i}]\!]^n_{V[\overline{T_i/X_i}]} \subseteq \overline{T_i}\right\}\right)_i\\
&\qquad \text{in } [\![\psi]\!]^n_{V[\overline{T_i/X_i}]}
\end{aligned}$$

Figure 11. Partial satisfiability

Every path may be given a *depth*:

$$\begin{aligned}
depth(\epsilon) &\stackrel{\text{def}}{=} 0\\
depth(\rho a) &\stackrel{\text{def}}{=} depth(\rho) + 1 \quad \text{if } a \in \{1, 2\}\\
depth(\rho a) &\stackrel{\text{def}}{=} depth(\rho) - 1 \quad \text{if } a \in \{\overline{1}, \overline{2}\}
\end{aligned}$$

A forward path is a path that only mentions forward modalities.

We define a tree of types $\mathcal{T}$ as a tree whose nodes are types, $\mathcal{T}(\cdot) = t$, with at most two children, $\mathcal{T}\langle 1\rangle$ and $\mathcal{T}\langle 2\rangle$. The navigation in trees of types is trivially extended to forward paths. A tree of types is *consistent* iff for every forward path ρ and for every child a of $\mathcal{T}\langle\rho\rangle$, we have $\mathcal{T}\langle\rho\rangle(\cdot) = t$, $\mathcal{T}\langle\rho a\rangle(\cdot) = t'$ implies $\langle a\rangle \top \in t$, $\langle\overline{a}\rangle \top \in t'$, and $\Delta_a(t, t')$.

Given a consistent tree of types $\mathcal{T}$, we now define a dependency graph whose nodes are pairs of a forward path ρ and a formula in $t = \mathcal{T}\langle\rho\rangle(\cdot)$ or the negation of a formula in the complement of t. The directed edges of the graph are labeled with modalities consistent with the tree. This graph corresponds to what the algorithm ultimately builds, as every iteration discovers longer forward paths. For every (ρ, φ) in the nodes we build the following edges:

- $\varphi \in \Sigma(\psi) \cup \neg\Sigma(\psi) \cup \{\textcircled{S}, \neg\textcircled{S}, \langle a\rangle \top, \neg\langle a\rangle \top\}$: no edge

- $\rho = \epsilon$ and $\varphi = \langle \overline{a} \rangle \, \varphi'$ with $a \in \{1, 2\}$: no edge
- $\rho = \rho' a$ and $\varphi = \langle a' \rangle \, \varphi'$: let $t = \mathcal{T} \, \langle \rho \rangle \, (\cdot)$.

  We first consider the case where $a' \in \{1, 2\}$ and let $t' = \mathcal{T} \, \langle \rho a' \rangle \, (\cdot)$. As $\mathcal{T}$ is consistent, we have $\varphi' \dot{\in} t'$ hence there are T, F such that $\varphi' \dot{\in} t' \Longrightarrow (T, F)$ with T a subset of t', and F a subset of the complement of t'. For every $\varphi_T \in T$ we add an edge a' to $(\rho a', \varphi_T)$, and for every $\varphi_F \in F$ we add an edge a' to $(\rho a', \neg \varphi_F)$.

  We now consider the case where $a' \in \{\overline{1}, \overline{2}\}$ and first show that we have $a' = \overline{a}$. As $\mathcal{T}$ is consistent, we have $\langle \overline{a} \rangle \top$ in t. Moreover, as t is a tree type, it must contain $\langle a' \rangle \top$. As a' is a backward modality, it must be equal to $\overline{a}$ as at most one may be present. Hence we have $\rho' a a' = \rho'$ and we let $t' = \mathcal{T} \, \langle \rho' \rangle \, (\cdot)$. By consistency, we have $\varphi' \dot{\in} t'$, hence $\varphi' \dot{\in} t' \Longrightarrow (T, F)$ and we add edges as in the previous case: to (ρ', φ_T) and to $(\rho', \neg \varphi_F)$.
- $\rho = \rho' a$ and $\varphi = \neg \langle a' \rangle \, \varphi'$: let $t = \mathcal{T} \, \langle \rho \rangle \, (\cdot)$. If $\langle a' \rangle \top$ is not in t then no edge is added. Otherwise, we proceed as in the previous case. For downward modalities, we let $t' = \mathcal{T} \, \langle \rho a' \rangle \, (\cdot)$ and we compute $\varphi' \dot{\notin} t' \Longrightarrow (T, F)$, which we know to hold by consistency. We then add edges to $(\rho a', \varphi_T)$ and to $(\rho a', \neg \varphi_F)$ as before. For upward modalities, as we have $\langle a' \rangle \top$ in t, we must have $a' = \overline{a}$ and we let $t' = \mathcal{T} \, \langle \rho' \rangle \, (\cdot)$. We compute $\varphi' \dot{\notin} t' \Longrightarrow (T, F)$ and we add the edges to (ρ', φ_T) and to $(\rho', \neg \varphi_F)$ as before.

LEMMA 6.4. *The dependency graph of a consistent tree of types of a cycle-free formula is cycle free.*

LEMMA 6.5 (Soundness). *Let T be the result set of the algorithm. For any type $t \in T$ and any φ such that $\varphi \dot{\in} t$, then $[\![\varphi]\!]^0_\emptyset \neq \emptyset$.*

Proof outline: The proof (detailed in [16]) proceeds by induction on the number of steps of the algorithm. For every t in T^n and every witness tree $\mathcal{T}$ rooted at t built from X^n, we show that $\mathcal{T}$ is a consistent tree type and we build a focused tree f that is rooted (i.e. of the shape $(\sigma^\circ[tl], (\epsilon, Top, tl')))$.

We then proceed to show that f satisfies $\varphi_c(t)$ at level 0. To do so, we use a further induction on the dependency tree. □

LEMMA 6.6 (Completeness). *For a cycle-free closed formula $\varphi \in \mathcal{L}_\mu$, if $[\![\varphi]\!]_\emptyset \neq \emptyset$ then the algorithm terminates with a set of triples X such that $\texttt{FinalCheck}(\varphi, X)$.*

Proof outline: As the formula is satisfiable, we consider a smallest focused tree f satisfying it. We then use Lemma 4.2 to derive a finite satisfaction relation of φ that contains f. We then rely on this relation to build a run of the algorithm that produces a type with no backward modality implying the formula. □

LEMMA 6.7 (Complexity). *For $\psi \in \mathcal{L}_\mu$ the satisfiability problem $[\![\psi]\!]_\emptyset \neq \emptyset$ is decidable in time $2^{O(n)}$ where $n = |Lean(\psi)|$.*

7. Implementation Techniques

Our implementation relies on a symbolic representation of sets of ψ-types using Binary Decision Diagrams (BDDs) [5].

First, we observe that the implementation can avoid keeping track of every possible witnesses of each ψ-type. In fact, for a formula φ, we can test $[\![\varphi]\!]_\emptyset \neq \emptyset$ by testing the satisfiability of the (linear-size) "plunging" formula $\psi = \mu X.\varphi \vee \langle 1 \rangle \, X \vee \langle 2 \rangle \, X$ at the root of focused trees. That is, checking $[\![\psi]\!]^0_\emptyset \neq \emptyset$ while ensuring there is no unfulfilled upward eventuality at top level 0. One advantage of proceeding this way is that the implementation only need to deal with a current set of ψ-types at each step.

We now introduce a bit-vector representation of ψ-types. Types are complete in the sense that either a subformula or its negation must belong to a type. It is thus possible for a formula $\varphi \in \text{Lean}(\psi)$ to be represented using a single BDD variable. For $\text{Lean}(\psi) = \{\varphi_1, ..., \varphi_m\}$, we represent a subset $t \subseteq \text{Lean}(\psi)$ by a vector $\vec{t} = \langle t_1, ..., t_m \rangle \in \{0, 1\}^m$ such that $\varphi_i \in t$ iff $t_i = 1$. A BDD with m variables is then used to represent a set of such bit vectors.

We define auxiliary predicates for programs $a \in \{1, 2\}$:

- $\text{isparent}_a(\vec{t})$ is read "$\vec{t}$ is a parent for program a" and is true iff the bit for $\langle a \rangle \top$ is true in $\vec{t}$
- $\text{ischild}_a(\vec{t})$ is read "$\vec{t}$ is a child for program a" and is true iff the bit for $\langle \overline{a} \rangle \top$ is true in $\vec{t}$

For a set $T \subseteq 2^{\text{Lean}(\psi)}$, we note χ_T its corresponding characteristic function. Encoding $\chi_{\text{Types}(\psi)}$ is straightforward. The predicate $\text{status}_\varphi(\vec{t})$ is the equivalent of $\dot{\in}$ on the bit vector representation. We now construct the BDD of the relation Δ_a for $a \in \{1, 2\}$. This BDD relates all pairs $(\vec{x}, \vec{y})$ that are consistent w.r.t the program a, i.e., such that $\vec{y}$ supports all of $\vec{x}$'s $\langle a \rangle \, \varphi$ formulas, and vice-versa $\vec{x}$ supports all of $\vec{y}$'s $\langle \overline{a} \rangle \, \varphi$ formulas:

$$\Delta_a(\vec{x}, \vec{y}) \stackrel{\text{def}}{=} \bigwedge_{1 \leq i \leq m} \begin{cases} x_i \leftrightarrow \text{status}_\varphi(\vec{y}) & \text{if } \varphi_i = \langle a \rangle \, \varphi \\ y_i \leftrightarrow \text{status}_\varphi(\vec{x}) & \text{if } \varphi_i = \langle \overline{a} \rangle \, \varphi \\ \top & \text{otherwise} \end{cases}$$

For $a \in \{1, 2\}$, we define the set of witnessed vectors:

$$\chi_{\texttt{Wit}_a(T)}(\vec{x}) \stackrel{\text{def}}{=} \text{isparent}_a(\vec{x}) \rightarrow \exists \vec{y} \; [\; h(\vec{y}) \wedge \Delta_a(\vec{x}, \vec{y}) \;]$$

where $h(\vec{y}) = \chi_T(\vec{y}) \wedge \text{ischild}_a(\vec{y})$.

Then, the BDD of the fixpoint computation is initially set to the false constant, and the main function $\texttt{Upd}(\cdot)$ is implemented as:

$$\chi_{\texttt{Upd}(T)}(\vec{x}) \stackrel{\text{def}}{=} \chi_T(\vec{x}) \vee \left(\chi_{\text{Types}(\psi)}(\vec{x}) \wedge \bigwedge_{a \in \{1,2\}} \chi_{\texttt{Wit}_a(T)}(\vec{x}) \right)$$

Finally, the solver is implemented as iterations over the sets $\chi_{\texttt{Upd}(T)}$ until a fixpoint is reached. The final satisfiability condition consists in checking whether ψ is present in a ψ-type of this fixpoint with no unfulfilled upward eventuality.

We use two major techniques for further optimization. First, BDD relational products ($\exists \vec{y} \; [\; h(\vec{y}) \wedge \Delta_a(\vec{x}, \vec{y}) \;]$) are computed using conjunctive partitioning and early quantification [10]. Second, we observed that choosing a good initial order of $\text{Lean}(\psi)$ formulas does significantly improve performance. Experience has shown that the variable order determined by the breadth-first traversal of the formula ψ to solve, which keeps sister subformulas in close proximity, yields better results in practice.

8. Typing Applications and Experimental Results

For XPath expressions $e_1, ..., e_n$, we can formulate several decision problems in the presence of XML type expressions $T_1, ..., T_n$:

- XPath containment: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \neg E^{\rightarrow}[\![e_2]\!]_{[\![T_2]\!]}$ (if the formula is unsatisfiable then all nodes selected by e_1 under type constraint T_1 are selected by e_2 under type constraint T_2)
- XPath emptiness: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]}$
- XPath overlap: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge E^{\rightarrow}[\![e_2]\!]_{[\![T_2]\!]}$
- XPath coverage: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \bigwedge_{2 \leq i \leq n} \neg E^{\rightarrow}[\![e_i]\!]_{[\![T_i]\!]}$

Two problems are of special interest for XML type checking:

- Static type checking of an annotated XPath query: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \neg [\![T_2]\!]$ (if the formula is unsatisfiable then all

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//c/foll-sibling::d/e
e_4 a/b//d[prec-sibling::c]/e
e_5 a/c/following::d/e
e_6 a/b[//c]/following::d/e $\cap$ a/d[preceding::c]/e

e_7 *//switch[ancestor::head]//seq//audio[prec-sibling::video]

e_8 descendant::a[ancestor::a]
e_9 /descendant::*
e_{10} html/(head | body)
e_{11} html/head/descendant::*
e_{12} html/body/descendant::*

Figure 12. XPath Expressions Used in Experiments.

DTD	Symbols	Binary Type Variables
SMIL 1.0	19	11
XHTML 1.0 Strict	77	325

Table 1. Types Used in Experiments.

nodes selected by e_1 under type constraint T_1 are included in the type T_2.)

- XPath equivalence under type constraints: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \neg E^{\rightarrow}[\![e_2]\!]_{[\![T_2]\!]}$ and $\neg E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge E^{\rightarrow}[\![e_2]\!]_{[\![T_2]\!]}$ (This test can be used to check that the nodes selected after a modification of a type T_1 by T_2 and an XPath expression e_1 by e_2 are the same, typically when an input type changes and the corresponding XPath query has to change as well.)

As no third-party implementation we know of addresses reverse axes and recursion, we simply provide evidence that our approach is efficient. We carried out extensive tests[1] [16] , and present here only a representative sample that includes the most complex language features such as recursive forward and backward axes, intersection, large and very recursive types with a reasonable alphabet size. The tests use XPath expressions shown on Fig. 12 (where "//" is used as a shorthand for "/desc-or-self::*/") and XML types shown on Table 1. Table 2 presents some decision problems and corresponding performance results. Times reported in milliseconds correspond to the running time of the satisfiability solver without the (negligible) time spent for parsing and translating into $\mathcal{L}_\mu$.

The first XPath containment instance was first formulated in [24] as an example for which the proposed tree pattern homomorphism technique is incomplete. The e_8 example shows that the official XHTML DTD does not syntactically prohibit the nesting of anchors. For the XHTML case, we observe that the time needed is more important, but it remains practically relevant, especially for static analysis operations performed only at compile-time.

9. Related Work

The XPath containment problem has attracted a lot of research attention in the database community, where the focus was given to the study of the impact of different XPath features on the containment complexity (see [28] for an overview). The complexity of XPath satisfiability in the presence of DTDs also is extensively studied in [3]. From these results, we know that XPath containment with or without type constraints ranges from EXPTIME to undecidable.

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

XPath Decision Problem	XML Type	Time (ms)
$e_1 \subseteq e_2$ and $e_2 \not\subseteq e_1$	none	353
$e_4 \subseteq e_3$ and $e_4 \subseteq e_3$	none	45
$e_6 \subseteq e_5$ and $e_5 \not\subseteq e_6$	none	41
e_7 is satisfiable	SMIL 1.0	157
e_8 is satisfiable	XHTML 1.0	2630
$e_9 \subseteq (e_{10} \cup e_{11} \cup e_{12})$	XHTML 1.0	2872

Table 2. Some Decision Problems and Corresponding Results.

Most formalisms used in the context of XML are related to one of the two logics used for unranked trees: first-order logic (FO), and Monadic Second Order Logic (MSO). FO and relatives are frequently used for query languages since they nicely capture their navigational features [2]. In a attempt to reach more expressive power, the work found in [1] proposes a variant of Propositional Dynamic Logic (PDL) with an EXPTIME complexity. MSO, specifically the weak monadic second-order logic of two successors (WS2S) [9], is one of the most expressive decidable logic used when both regular types and queries [2] are under consideration. WS2S satisfiability is known to be non-elementary. A drawback of the WS2S decision procedure is that it requires the full construction and complementation of tree automata.

Some temporal and fixpoint logics closely related to MSO have been introduced and allow to avoid explicit automata construction. The propositional modal μ-calculus introduced in [22] has been shown to be as expressive as nondeterministic tree automata [11]. Since it is trivially closed under negation, it constitutes a good alternative for studying MSO-related problems. Moreover, it has been extended with converse programs in [31]. The best known complexity for the resulting logic is obtained through reduction to the emptiness problem of alternating tree automaton which is in $2^{O(n^4 \cdot \log n)}$, where n corresponds to the length of a formula [18]. Unfortunately the logic lacks the finite model property. From [31], we know that WS2S is exactly as expressive as the alternation-free fragment (AFMC) of the propositional modal μ-calculus. Furthermore, the AFMC subsumes all early logics such as CTL and PDL (see [2] for a complete survey on tree logics).

The goal of the research presented so far is limited to establishing new theoretical properties and complexity bounds. Our research differs in that we seek precise complexity bounds, efficient implementation techniques, and concrete design that may be directly applied to the type checking of XPath queries under regular tree types.

In this line of research, some experimental results based on WS2S, through the Mona tool [21], have recently been reported for XPath containment [15]. However, for static analysis purposes, the explosiveness of the approach is very difficult to control due to the non-elementary complexity. Closer to our contribution, the recent work found in [29] provides a decision procedure for the AFMC with converse whose time complexity is $2^{O(n \cdot \log n)}$. However, models of the logic are Kripke structures (infinite graphs). Enforcing the finite tree model property can be done at the syntactic level [29], and this has been further developed in the XML setting in [14]. Nevertheless, the drawback of this approach is that the AFMC decision procedure requires expensive cycle-detection for rejecting infinite derivation paths for least fixpoint formulas. The present work shows how this can be avoided for finite trees. As a consequence, the resulting performance is much more attractive. In an earlier work on XML type checking, a logic for finite trees was presented [30], but the logic is not closed under negation.

In [7], a technique is presented for statically ensuring correctness of paths. The approach only deals with emptiness of XPath

expressions without reverse axes, whereas our approach solves the more general problem of containment, including reverse axes.

The work [25] presents an approximated technique that is able to statically detect errors in XSLT stylesheets. Their approach could certainly benefit from using our exact algorithm instead of their conservative approximation. The XDuce [19], CDuce [4], and XStatic [13] languages support pattern-matching through regular expression types but not XPath. A survey on existing research on statically type checking XML transformations can be found in [25].

10. Conclusion

The main result of our paper is a sound and complete algorithm for the satisfiability of decision problems involving regular tree types and XPath queries with a tighter $2^{O(n)}$ complexity in the length of a formula. Our approach is based on a sub-logic of the alternation-free modal μ-calculus with converse for finite trees.

Our proof method reveals deep connections between this logic and XPath decision problems. First, the translations of XML regular tree types and a large XPath fragment are cycle-free and linear in the size of the corresponding formulas in the logic. Second, on finite trees, since both operators are equivalent, the logic with a single fixpoint operator is closed under negation. This allows to address key XPath decision problems such as containment.

Finally, there are a number of interesting directions for further research that build on ideas developed here: extending XPath to restricted data values comparisons that preserves this complexity, for instance data values on a finite domain, and integrating related work on counting [8] to our logic. We also plan on continuing to improve the performance of our implementation.

Acknowledgments

We would like to thank Giorgio Ghelli for his helpful comments on earlier drafts and Benjamin C. Pierce for his useful suggestions.

References

[1] L. Afanasiev, P. Blackburn, I. Dimitriou, B. Gaiffe, E. Goris, M. Marx, and M. de Rijke. PDL for ordered trees. *Journal of Applied Non-Classical Logics*, 15(2):115–135, 2005.

[2] P. Barceló and L. Libkin. Temporal logics over unranked trees. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 31–40, New York, NY, USA, 2005.

[3] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *PODS '05: Proceedings of the twenty-fourth ACM Symposium on Principles of Database Systems*, pages 25–36, New York, NY, USA, 2005. ACM Press.

[4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *ICFP '03: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 51–63, New York, NY, USA, 2003. ACM Press.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, 1986.

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

[7] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Static analysis for path correctness of XML queries. *J. Funct. Program.*, 16(4-5):621–661, 2006.

[8] S. Dal Zilio, D. Lugiez, and C. Meyssonnier. A logic you can count on. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 135–146, New York, NY, USA, 2004. ACM Press.

[9] J. Doner. Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 4:406–451, 1970.

[10] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[11] E. A. Emerson and C. S. Jutla. Tree automata, μ-calculus and determinacy. In *Proceedings of the 32nd annual Symposium on Foundations of Computer Science*, pages 368–377, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[12] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 587–598, New York, NY, USA, 2004. ACM Press.

[13] V. Gapeyev and B. C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany*, 2003. A preliminary version was presented at FOOL '03.

[14] P. Genevès and N. Layaïda. A system for the static analysis of XPath. *ACM Trans. Inf. Syst.*, 24(4):475–502, 2006.

[15] P. Genevès and N. Layaïda. Deciding XPath containement with MSO. *Data & Knowledge Engineering, to appear*, 2007.

[16] P. Genevès, N. Layaïda, and A. Schmitt. A satisfiability solver for XML and XPath, June 2006. http://wam.inrialpes.fr/xml.

[17] P. Genevès and J.-Y. Vion-Dury. Logic-based XPath optimization. In *DocEng '04: Proceedings of the 2004 ACM Symposium on Document Engineering*, pages 211–219, NY, USA, 2004. ACM Press.

[18] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata logics, and infinite games: a guide to current research*. Springer-Verlag, New York, NY, USA, 2002.

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

[20] G. P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[21] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA 1.4, January 2001. http://www.brics.dk/mona/.

[22] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[23] M. Y. Levin and B. C. Pierce. Type-based optimization for regular patterns. In *DBPL '05: Proceedings of the 10th International Symposium on Database Programming Languages*, volume 3774 of *LNCS*, London, UK, August 2005. Springer-Verlag.

[24] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[25] A. Møller and M. I. Schwartzbach. The design space of type checkers for XML transformation languages. In *Proc. Tenth International Conference on Database Theory, ICDT '05*, volume 3363 of *LNCS*, pages 17–36. Springer-Verlag, January 2005.

[26] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, 2005.

[27] G. Pan, U. Sattler, and M. Y. Vardi. BDD-based decision procedures for the modal logic K. *Journal of Applied Non-classical Logics*, 16(1-2):169–208, 2006.

[28] T. Schwentick. XPath query containment. *SIGMOD Record*, 33(1):101–109, 2004.

[29] Y. Tanabe, K. Takahashi, M. Yamamoto, A. Tozawa, and M. Hagiya. A decision procedure for the alternation-free two-way modal μ-calculus. In *In TABLEAUX 2005*, volume 3702 of *LNCS*, pages 277–291, London, UK, September 2005. Springer-Verlag.

[30] A. Tozawa. On binary tree logic for XML and its satisfiability test. In *PPL '04: Informal Proceedings of the Sixth JSSST Workshop on Programming and Programming Languages*, 2004.

[31] M. Y. Vardi. Reasoning about the past with two-way automata. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 628–641, London, UK, 1998. Springer-Verlag.