A Decision Procedure for XPath Containment

Pierre Genevès, Nabil Layaïda

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Decision Procedure for XPath Containment

Pierre Genevès — Nabil Layaïda

N° 5867

Mars 2006

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--5867--FR+ENG

A Decision Procedure for XPath Containment

Pierre Genevès , Nabil Layaïda

Thème SYM — Systèmes symboliques
Projet WAM

Rapport de recherche n° 5867 — Mars 2006 — 41 pages

Abstract: XPath is the standard language for addressing parts of an XML document. We present a sound and complete decision procedure for containment of XPath queries. The considered XPath fragment covers most of the language features used in practice. Specifically, we show how XPath queries can be translated into equivalent formulas in monadic second-order logic. Using this translation, we construct an optimized logical formulation of the containment problem, which is decided using tree automata. When the containment relation does not hold between two XPath expressions, a counter-example XML tree is generated. We provide a complexity analysis together with practical experiments that illustrate the efficiency of the decision procedure for realistic scenarios.

Key-words: XML, XPath, queries, containment, logic, automata, MSO, WS2S

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Une procédure de décision pour l'inclusion des requêtes XPath

Résumé : XPath est le langage standard pour naviguer dans les documents XML et sélectionner un ensemble de noeuds. Nous présentons une procédure de décision correcte pour l'inclusion des requêtes XPath. Le fragment considéré de XPath couvre la plupart des constructions du langage utilisées en pratique. Nous montrons comment les expressions XPath peuvent être traduites en formules équivalentes en logique monadique du second ordre. En utilisant cette traduction, nous construisons une formulation logique optimisée du problème de l'inclusion, qui est décidée à l'aide d'automates d'arbres. Lorsque deux expressions XPath ne sont pas contenues l'une dans l'autre, un arbre XML contre-exemple est généré. Nous effectuons une analyse de complexité et menons des expérimentations qui illustrent l'efficacité de la procédure de décision pour des scénarios réalistes.

Mots-clés : XML, XPath, requêtes, inclusion, logique, automates, MSO, WS2S

1 Introduction

XPath is a simple language for querying an XML tree and returning a set of nodes. It is increasingly popular due to its expressive power and its compact syntax. These two 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 node selector in the transformations; in XML Schema to define keys; in XLink and XPointer to reference portions of XML data. XPath is also used in many applications such as update languages [38]; XML access control [19] and static analysis of transformations [41]. In all these applications and many others, solving the XPath containment problem is crucial.

The containment problem received a great research attention recently. The general formulation of the containment is as follows: given two XPath queries p_1 and p_2, check whether for any tree t, the results of the evaluation of p_1 are always contained in those of p_2. Other variants are also under scrutiny such as when t is additionally constained by an XML Schema or DTD. Fundamental questions such as the equivalence of two paths and the emptiness check of a path are both by-products of the containment.

Most of XPath containment applications such as query optimization, typechecking, key inference, are carried out statically. This allows for example to replace queries by more efficient specialized ones or to identify at compile time those that do not need to be evaluated at run time since they yield no results. This kind of analysis may help shifting the cost of enforcing runtime properties to compile time.

In the literature, much of the attention has been paid to classifying the containment problem for a simple XPath fragment in complexity classes. This allowed to identify subsets of this fragment for which deciding the containment can be done efficiently.

In this paper, our goal is to describe a sound, complete and efficient algorithm for a large XPath fragment supporting most of real-world use cases. We first briefly introduce the XPath language, and present the approach and outline of the paper.

1.1 Introduction to XPath

XPath [11] has been introduced by the W3C as the standard query language for retrieving information in XML documents. It allows to navigate in XML trees and return 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 it's "chapter" child elements and on to it's "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 that 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* "section" elements that have a least one child element named "citation". The situation becomes more interesting when combined with XPath's capability of searching along "axes" other than the shown "children of" axis. Indeed the above 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" sub-element. 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 nodes 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 [11].

Figure 2 gives the abstract syntax of the XPath fragment we consider in this paper. The fragment covers most features of XPath 1.0[1]. It includes all forward and reverse axes along with path composition, and boolean operators inside qualifiers (including negation), except data-value joins. The formal semantics of XPath is given in Section 3.1.

1.2 Approach and Outline

We propose the following staged approach for solving the containment problem between two XPath expressions:

1. translate each XPath query to an equivalent logical representation
2. express the containment problem as a formula in this logic
3. optimize the formula by taking advantage of specific peculiarities of the containment problem
4. solve the generated formula using an optimized solver
5. provide relevant examples and/or counter-examples of the truth status of the formula

[1] The fragment also includes two extensions from the forthcoming XPath 2.0 [7] language: qualified paths (e.g. $(p)[q]$) instead of XPath 1.0 qualified steps (e.g. $a::n[q]$) and path intersection ($p_1 \cap p_2$).

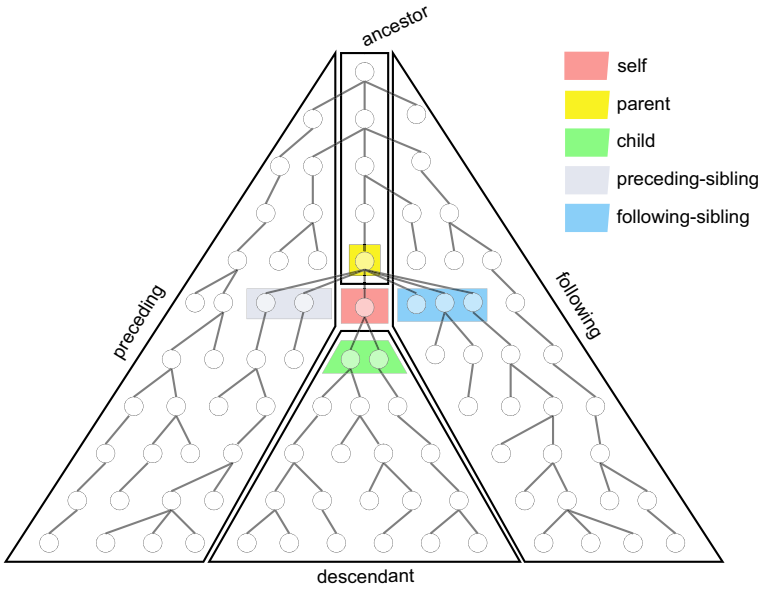

Figure 1: XPath Axes Partition from Context Node.

Expression	e	::=	$/p \mid p$
Path	p	::=	$p_1/p_2 \mid p[q] \mid e_1 \cup e_2 \mid e_1 \cap e_2 \mid (p) \mid a::n$
Qualifier	q	::=	q and $q \mid q$ or $q \mid$ not $q \mid e$
Axis	a	::=	child | descendant | self | descendant-or-self | following-sibling | following | parent | ancestor | ancestor-or-self | preceding-sibling | preceding
NodeTest	n	::=	$\sigma \mid *$

Figure 2: XPath Abstract Syntax.

Section 2 introduces the logic we use, Section 3 explains the translation of XPath queries to logical formulas, Section 4 presents the logical formulation of the containment problem. Complexity analysis and practical results are given in Section 5. Section 6 summarizes the related work before we conclude in Section 7.

2 A Logic for XML

In this section we introduce a specific variant of monadic second-order logic (MSO) as a formalism for representing XML instances.

2.1 Logical Description of Trees

An XML document can be seen as a finite ordered and labeled tree of unbounded depth and arity. Tree nodes are labeled with symbols taken from a finite[2] alphabet Σ. There is a straightforward isomorphism between sequences of unranked trees and binary trees [24, 33]. In order to describe it, we first define unranked trees as $\sigma(h)$ where $\sigma \in \Sigma$ and h is a hedge, i.e. a sequence of unranked trees, defined as follows:

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

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

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

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

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

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

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

For example, Figure 3 illustrates how the sample tree $r(a, b, c)$ is mapped to its binary representation $r(a(\epsilon, b(\epsilon, c(\epsilon, \epsilon))), \epsilon)$ 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.

[2] We present a technique for infinite alphabets in Section 2.6.

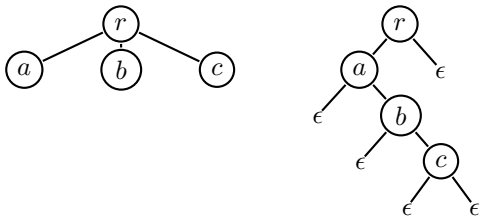

Figure 3: A n-ary Tree and its Binary Representation.

We define a *position* in a binary tree as a finite string over the alphabet $\{0, 1\}$ which identifies a node in the tree, like a path starting from the root. Each symbol of the string either corresponds to accessing the left child (0) or the right child (1) in the binary tree. Since a position in the tree uniquely identifies a node, and a node is uniquely identified by its position, nodes and positions are not distinguished.

A *characteristic function* of a set B is a function from A to {0,1}, where A is a superset of B. It returns 1 if and only if the element of A is also an element of B:

$$\begin{array}{l} B \subseteq A \\ f : A \rightarrow \{0, 1\} \\ \forall a \in A, f(a) = \left\{ \begin{array}{l} 1, \text{if } a \in B \\ 0, \text{if } a \notin B \end{array} \right. \end{array}$$

A *characteristic set* is a subset of a set A that contains all elements of A for which the characteristic function returns 1:

$$\begin{array}{l} X_f \subseteq A \\ X_f = \{a \in A \mid f(a) = 1\} \end{array}$$

In this paper, we consider characteristic sets which are subsets of the set of all positions in a tree. Such a characteristic set denotes where a particular property holds in a tree. Particular attention is paid to characteristic sets which tell us where a particular symbol occurs. Consider for instance the binary tree over the alphabet $\Sigma = \{r, a, b, c, \epsilon\}$ which is given on Figure 3. It is identified by its tuple representation $t_1 = (X_{f_r}, X_{f_a}, X_{f_b}, X_{f_c}, X_{f_\epsilon})$ where X_{f_σ} is the characteristic set of the symbol σ:

$$\begin{array}{ll} X_{f_r} = \{\epsilon\} & X_{f_c} = \{011\} \\ X_{f_a} = \{0\} & X_{f_\epsilon} = \{1, 00, 010, 0110, 0111\} \\ X_{f_b} = \{01\} & \end{array}$$

The set $X_{f_r} \cup X_{f_a} \cup X_{f_b} \cup X_{f_c} \cup X_{f_\epsilon}$ of all positions contained in characteristic sets defines a *shape*. A position p belongs to a characteristic set X_{f_σ} (also noted X_σ) if and only if the symbol σ occurs at p in the shape. Note that in the example of Figure 3, one and only one symbol occurs at each position. In the general case however, there is no restriction on the

content of characteristic sets. A given position can belong to several characteristic sets. In this case, several symbols may occur at a given position, and therefore we do not describe an instance anymore but a simple union type instead. On the opposite, a particular position may not be a member of any characteristic set. In this case, the overall structure contains a position which is not labeled by any symbol of the considered alphabet; therefore it is not a tree on this alphabet. We now introduce the logic that allows to capture additional constraints needed for shapes to conform to XML trees.

2.2 Introduction to WS2S

The logic we use is named WS2S which stands for *weak monadic second-order logic of two successors*. In this logic, first-order variables range over tree nodes. Second-order variables are interpreted as finite sets of tree nodes. A *weak* second order theory is one in which the set variables are allowed to range only over finite sets. *Weak* is enough for our application since XML documents have an unbounded depth but remain finite trees. *Monadic* means that quantification is only allowed over unary relations (sets), not over polyadic relations. *Monadic* allows quantification over set of nodes, which is powerful enough to model recursion in XPath queries (as we will see in Section 3.2). The *two successors* denote the left and right children of a node in the binary tree. They are sufficient to consider n-ary XML trees without loss of generality, owing to the mapping $\mathcal{B}$ presented in Section 2.1.

From a syntactic point of view, WS2S can be reduced to a simple core language, whose abstract syntax is:

$$\Phi \quad ::= \quad X \subseteq Y \mid X = Y - Z \mid X = Y.0 \mid X = Y.1 \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists X.\Phi$$

where X, Y, and Z denote arbitrary second-order variables. Other usual logical connectives can be derived from the core:

$$\begin{array}{lcl} \Phi_1 \vee \Phi_2 & \stackrel{\text{def}}{=} & \neg(\neg\Phi_1 \wedge \neg\Phi_2) \\ \Phi_1 \Rightarrow \Phi_2 & \stackrel{\text{def}}{=} & \neg\Phi_1 \vee \Phi_2 \\ \Phi_1 \Leftrightarrow \Phi_2 & \stackrel{\text{def}}{=} & \Phi_1 \wedge \Phi_2 \vee \neg\Phi_1 \wedge \neg\Phi_2 \end{array}$$

Universal quantification can be seen as a syntactic sugar:

$$\forall X.\Phi \stackrel{\text{def}}{=} \neg\exists X.\neg\Phi$$

Note that only second order variables appear in the core. This is because first order variables can be encoded as singleton second-order variables. We adopt a notation convention for simplifying the remaining part of the paper: first-order variables are noted in lowercase and second-order variables in uppercase.

2.3 Semantics of the Logic

Given a fixed main formula φ with k variables, we define its semantics inductively. Let a tuple representation $t = (X_1, ..., X_k) \in (\{0,1\}^*)^k$ be an interpretation of φ. We note $t(X)$

the interpretation X_i (such that $1 \leq i \leq k$) that t associates to the variable X occurring in φ. The semantics of φ is inductively defined relative to t. We use the notation $t \vDash \varphi$ (which is read: t satisfies φ) if the interpretation t makes φ true:

$$\begin{array}{lll} t \vDash X \subseteq Y & \text{iff} & t(X) \subseteq t(Y) \\ t \vDash X = Y - Z & \text{iff} & t(X) = t(Y) \setminus t(Z) \\ t \vDash X = Y.0 & \text{iff} & t(X) = \{p.0 \mid p \in t(Y)\} \\ t \vDash X = Y.1 & \text{iff} & t(X) = \{p.1 \mid p \in t(Y)\} \\ t \vDash \neg\varphi & \text{iff} & t \nvDash \varphi \\ t \vDash \varphi_1 \wedge \varphi_2 & \text{iff} & t \vDash \varphi_1 \text{ and } t \vDash \varphi_2 \\ t \vDash \exists X.\varphi & \text{iff} & \exists I \subseteq \{0,1\}^*, t[X \mapsto I] \vDash \varphi \end{array}$$

where the notation $t[X \mapsto I]$ denotes the tuple representation that interprets X as I and all other variables as t does. Note that the two successors of a particular position always exist in WS2S.

A formula φ naturally defines a language $\mathcal{L}(\varphi) = \{t \mid t \vDash \varphi\}$ over the alphabet $(\{0,1\}^*)^k$, where k is the number of variables of φ.

2.4 Decidability

A logic is *decidable* if an algorithm exists that determines for any formula its truth status: a formula can be valid (always true) or not valid; alternatively (and equivalently) the algorithm can classify formulas according to whether they are satisfiable (sometimes true) or unsatisfiable (always false).

It has been known since the 1960's that the class of regular tree languages is linked to decidability questions in formal logics. In particular, WS2S is decidable through the automaton-logic connection [40, 15], using tree automata. This follows and generalizes the results on the decidability of the weak monadic second-order logic of one successor (WS1S) using word-automata [10, 18]. Specifically, in 1960, Büchi proved that WS1S is as expressive as finite word-automata, and in 1968, Thatcher and Wright found out that there is an analogous correspondence for the extended case:

Theorem 2.4.1 *WS2S is as expressive as finite tree automata.*

The proof works in two directions. First, it is shown that a WS2S formula can be created such that it simulates a successful run of a tree-automaton. Second, for any given WS2S formula a corresponding tree automaton can be built. We explain and detail this second direction as it forms the theoretical basis of the WS2S decision procedure we use.

The correspondence of WS2S formulas and tree automata relies on a convenient representation that links the truth status of a formula with the recognition operated by an automaton. This representation is a matricial vision of the tuple representation described in Section 2.1. If we consider a tuple t, its matricial representation $\widetilde{t}$ is indexed by variables indices and positions in the tree. Entries of $\widetilde{t}$ correspond to values in $\mathbb{B} = \{0,1\}$ of characteristic functions: an entry $(v,p) = 1$ in $\widetilde{t}$ means that the position p belongs to the variable X_v.

Consider for instance the formula $\varphi = (\exists X.\exists Y.\ Y = Z.0 \wedge X = Z.1)$ which has three variables X, Y, and Z. A typical matrix looks like:

	ϵ	0	00	01	010	1
X	1	1	0	0	0	0
Y	0	1	0	1	0	0
Z	0	0	1	0	0	1

Note that this matrix is finite since we work with finite trees, and allows to capture trees of unbounded depth. As a counterpart, there is an infinite number of matrices that define the same interpretation: we can append any number of columns of zeros at the right end of the matrix (for positions after the end of the tree). Actually, we denote by $\widetilde{t}$ the minimum matrix, without such empty suffix. Rows of the matrix are called tracks and give the interpretation of each variable, which is defined as the finite set $\{p \mid$ the bit for position p in the X_i track is $1\}$.

Each column of the matrix is a bit vector that indicates the membership status of a node to the variables of the formula. The automaton recognizes all the interpretations (matrices) that satisfy the formula. A line by line reading of the matrix gives the interpretation of each variable (i.e. its associated set of positions), whereas an automaton processes the matrix column by column; it transits on each bit-vector.

2.5 From Formulas to Automata

Given a particular formula, a corresponding finite tree automaton (FTA) can be built in order to decide the truth status of the formula. We denote a FTA over an alphabet Σ of node labels by a tuple (Q, Q_f, Γ) where Q is the set of states, $Q_f \subseteq Q$ is a set of accepting states, and Γ is a set of transitions either of the form $\sigma \rightarrow q$ or of the form $\sigma(q, q') \rightarrow q''$, depending on the arity of the symbol $\sigma \in \Sigma$ (respectively a leaf or a binary constructor) and where q, q', q'' are automaton states belonging to Q.

We start from a formula φ with k second-order variables. As an interpretation of φ, consider a tuple representation $t = (X_1, ..., X_k) \in (\{0,1\}^*)^k$. We note $\mathcal{A}[\![\varphi]\!]$ the tree automaton that corresponds to φ. $\mathcal{A}[\![\varphi]\!]$ operates over the alphabet $\Sigma = \{0,1\}^k$, and can be seen as processing $\widetilde{t}$ column by column. Note however that there is an infinite number of matrices that defines the same interpretation. On one hand, any number columns of zeros can appear at the end of the matrix. On the other hand, a column of zeros can also appear for any position in the tree, before a non-empty column, denoting that this position is not a member of any interpretation. The automaton therefore faces a problem: when recognizing a column of zeros, knowing if the recognition should stop (because the end of the tree has been reached) or continue. In other terms, the automaton needs to know the maximal depth of the tree as an additional information in order to know when to stop. To this end, we introduce a new termination symbol $\perp$. From a matricial point of view, this symbol appears as a component of a bit-vector whenever this component will not be 1 anymore for the remaining bit-vectors to be processed. Technically, $\mathcal{A}[\![\varphi]\!]$ recognizes the tree representation $\overline{t}$ of t. $\overline{t}$ is obtained from t as follows:

$$t = (\{0\}, \{0,1\})$$

$\widetilde{t} =$

	ϵ	0	1
X	0	1	0
Y	0	1	1

$\overline{t} =$

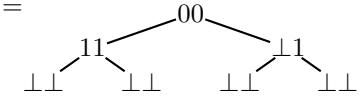

Figure 4: Representations of a Satisfying Interpretation of $X \subseteq Y$

1. the set of positions of $\overline{t}$ is the prefix-closure of $X_1 \cup ... \cup X_k$
2. leaves of $\overline{t}$ are labeled with $\bot^k$
3. binary constructors of the tree are labeled with an element of $\{\bot, 0, 1\}^k$ such that the i^{th} component of a position p in $\overline{t}$ is marked: 1 if and only if $p \in X_i$, 0 if and only if $p \notin X_i$ and some extension of p is in X_i, and $\bot$ otherwise

Note that in this tree representation, $\bot$ appears as a component of a node label whenever no descendant node has a 1 for the same component. For example, Figure 4 gives the tuple, the matrix, and the tree representation of a particular satisfying interpretation of the formula $X \subseteq Y$.

Theorem 2.5.1 *For every formula φ, there is an automaton $\mathcal{A}[\![\varphi]\!]$ such that:*

$$t \vDash \varphi \quad \equiv \quad \mathcal{A}[\![\varphi]\!] \textit{ accepts } \overline{t}$$

The automaton $\mathcal{A}[\![\varphi]\!]$ is calculated using an induction scheme. First, a basic bottom-up tree automaton corresponds to each atomic formula:

$$\mathcal{A}[\![X \subseteq Y]\!] = \left(\left\{ \begin{array}{ll} \bot\bot \to q, & \bot 0(q,q) \to q \\ \bot 1(q,q) \to q, & 00(q,q) \to q \\ 01(q,q) \to q, & 11(q,q) \to q \end{array} \right\}, \{q\} \right)$$

$$\mathcal{A}[\![X = Y - Z]\!] = \left(\left\{ \begin{array}{ll} \bot\bot\bot \to q, & \bot\bot 0(q,q) \to q, \\ \bot 0\bot(q,q) \to q, & \bot 00(q,q) \to q, \\ \bot 01(q,q) \to q, & \bot 11(q,q) \to q, \\ 0\bot\bot(q,q) \to q, & 0\bot 0(q,q) \to q, \\ 0\bot 1(q,q) \to q, & 00\bot(q,q) \to q, \\ 000(q,q) \to q, & 001(q,q) \to q, \\ 011(q,q) \to q, & 11\bot(q,q) \to q, \\ 110(q,q) \to q, & \end{array} \right\}, \{q\} \right)$$

$$\mathcal{A}[\![X = Y.0]\!] = \left(\left\{ \begin{array}{ll} \bot\bot \to q, & 00(q,q') \to q' \\ 00(q',q) \to q' & 01(q'',q) \to q' \\ 1\bot(q,q) \to q'' & 10(q,q) \to q'' \end{array} \right\}, \{q'\} \right)$$

$$\mathcal{A}[\![X = Y.1]\!] = \left(\left\{ \begin{array}{ll} \bot\bot \to q, & 00(q,q') \to q' \\ 00(q',q) \to q' & 01(q,q'') \to q' \\ 1\bot(q,q) \to q'' & 10(q,q) \to q'' \end{array} \right\}, \{q'\} \right)$$

Logical connectives are then translated into automata-theoretic operations, taking advantage of the closure properties of tree automata.

Negation of a formula is handled through automaton complementation:

$$\mathcal{A}[\![\neg\varphi]\!] = \complement\mathcal{A}[\![\varphi]\!]$$

Complementation of a *complete* FTA simply consists in flipping accepting and rejecting states. Note that a FTA (Q, Q_f, Γ) is *complete* if and only if there is a transition $\sigma(q,q') \to q''$ for each $\sigma \in \Sigma$ and $(q,q',q'') \in Q^3$. Therefore, completing an automaton can be required before complementing it. For instance, the automaton $\mathcal{A}[\![X \subseteq Y]\!]$ given above is incomplete. A way to obtain $\mathcal{A}[\![\neg X \subseteq Y]\!]$ from it is to add a new state q', complete the transitions, and consider q' as the only accepting state of $\mathcal{A}[\![\neg X \subseteq Y]\!]$.

Conjunction in a formula is translated into intersection of automata:

$$\mathcal{A}[\![\varphi_1 \wedge \varphi_2]\!] = \mathcal{A}[\![\varphi_1]\!] \cap \mathcal{A}[\![\varphi_2]\!]$$

If we consider that $\mathcal{A}[\![\varphi_1]\!] = (Q_1, Q_{f_1}, \Gamma_1)$ and $\mathcal{A}[\![\varphi_2]\!] = (Q_2, Q_{f_2}, \Gamma_2)$, obtaining $\mathcal{A}[\![\varphi_1]\!] \cap \mathcal{A}[\![\varphi_2]\!]$ basically consists in calculating a product automaton:

$$\mathcal{A}[\![\varphi_1]\!] \cap \mathcal{A}[\![\varphi_2]\!] = (Q_1 \times Q_2, Q_{f_1} \times Q_{f_2}, \Gamma)$$

where:

$$\Gamma = \left\{ \sigma((q_1,q_2),(q_1',q_2')) \to (q_1'',q_2'') \middle| \begin{array}{l} \sigma(q_1,q_1') \to q_1'' \in \Gamma_1 \\ \sigma(q_2,q_2') \to q_2'' \in \Gamma_2 \end{array} \right\}$$

Existential quantification relies on projection and determinization of tree automata. The automaton $\mathcal{A}[\![\exists X.\varphi]\!]$ is derived from $\mathcal{A}[\![\varphi]\!]$ by projection. This means the alphabet of $\mathcal{A}[\![\exists X.\varphi]\!]$ has to be one element smaller than the alphabet of $\mathcal{A}[\![\varphi]\!]$. In every tuple of $\mathcal{A}[\![\varphi]\!]$ the X component is removed, so that its size is decreased by one. The rest of the automaton remains the same. Intuitively, $\mathcal{A}[\![\exists X.\varphi]\!]$ acts as $\mathcal{A}[\![\varphi]\!]$ except it is allowed to guess the bits for X. The automaton $\mathcal{A}[\![\exists X.\varphi]\!]$ may be non-deterministic even if $\mathcal{A}[\![\varphi]\!]$ was not [13], that is why determinization is required.

As a result, for every formula φ it is possible to build an automaton $\mathcal{A}[\![\varphi]\!]$ in this manner, which defines the same language as φ:

$$\mathcal{L}(\mathcal{A}[\![\varphi]\!]) = \mathcal{L}(\varphi)$$

Analyzing the automaton $\mathcal{A}[\![\varphi]\!]$ allows to decide the truth status of the formula φ:

- if $\mathcal{L}(\mathcal{A}[\![\varphi]\!]) = \emptyset$ then φ is unsatisfiable;
- else φ is satisfiable. If $\mathcal{L}(\complement\mathcal{A}[\![\varphi]\!]) = \emptyset$ then φ is always satisfiable (valid).

Possessing the full automaton corresponding to a formula is of great value, since we can use it to produce examples and counter-examples of the truth status of the formula. We can generate a relevant example (or counter-example) by looking for an accepting run of the automaton (or its complement). In practice, our implementation relies on the MONA solver [27] that implements this WS2S decision procedure along with various optimizations.

2.6 XML Tree Represention

We have seen in Section 2.1 how shapes can be defined using characteristic sets. A shape is basically a second order variable, interpreted as a set of nodes, for which particular properties hold. Using WS2S, we now express additional requirements needed in order for a shape X to represent an XML tree.

The first requirements are structural. First, in order to be a tree, the shape must be prefix-closed, that is, for any position in the tree, any prefix of this position is also in the tree:

$$\text{PrefixClosed}(X) \stackrel{\text{def}}{=} \forall x.\forall y.((y = x.1 \ \vee \ y = x.0) \wedge y \in X) \Rightarrow x \in X$$

This ensures the shape is fully connected.

Second, let us define the following predicate for the root of X:

$$\text{IsRoot}(X, x) \stackrel{\text{def}}{=} x \in X \wedge \neg(\exists z.z \in X \wedge (x = z.1 \vee x = z.0))$$

In order to be a tree and not a hedge, X must have only one root with no sibling:

$$\text{SingleRoot}(X) \stackrel{\text{def}}{=} \forall x.\text{IsRoot}(X, x) \Rightarrow x.1 \notin X$$

Then, the labeling of the tree must be consistent with XML. We want to tolerate that the same symbol may appear at several locations in the tree with different arities: either as

a binary constructor or as a leaf. However, one and only one symbol is associated with a position in the shape. We may at first want to consider that the set of characteristic sets forms a partition:

$$\begin{array}{lcl} \text{Partition}(X, X_1, ..., X_n) & \stackrel{\text{def}}{=} & X = \bigcup_{i=1}^{n} X_i \wedge \text{Disjoint}(X_1, ..., X_n) \\ \text{Disjoint}(X_1, ..., X_n) & \stackrel{\text{def}}{=} & \bigwedge_{i \neq j} X_i \cap X_j = \emptyset \end{array}$$

but this would prevent us from considering an actual XML tree which is labeled with symbols taken from an infinite alphabet. Actually, the problem comes from declaring $X = \bigcup_{i=1}^{n} X_i$ that prevents any other symbol to occur in the tree. Instead, if we only specify that the characteristic sets must be disjoint, then we allow a position in the tree not to be a member of any of the considered characteristic sets. That is how we emulate the labeling from an infinite alphabet. As a result, we encode an XML tree (that we want non-empty in order not to get degenerated results) in the following way:

$$\begin{array}{lcl} \text{XMLTree}(X, X_1, ..., X_n) & \stackrel{\text{def}}{=} & \text{PrefixClosed}(X) \\ & \wedge & \text{SingleRoot}(X) \\ & \wedge & \text{Disjoint}(X_1, ..., X_n) \\ & \wedge & X \neq \emptyset \end{array}$$

where X is the tree and X_i the characteristic sets. Figure 5 introduces how we formulate this in MONA Syntax [27], for the case of two characteristic sets of interest named `Xbook` and `Xcitation`. The only difference is that the shape X is declared as a global free variable named `$` together with associated restrictions, instead of being passed as a parameter to predicates. In MONA syntax, "`var2`" is the keyword for declaring a free second-order variable; "`all1`" is the universal quantifier for first-order variables; and "&" and "|" respectively stand for the "$\wedge$" and "$\vee$" connectives.

3 Logical Interpretation of XPath Queries

We explain in this section how an XPath expression can be translated into an equivalent WS2S formula. We first recall XPath denotational semantics then introduce our logical interpretation of an XPath query. This representation basically consists in considering a query as a relation that connects two tree nodes: the context node from which the query is applied, and the result node.

3.1 Denotational Semantics

In the classical denotational semantics of paths, first given in [42], the evaluation of a path returns a set of nodes. Figure 6 presents the denotational semantics of our XPath fragment. The formal semantics functions $\mathcal{S}_e$ and $\mathcal{S}_p$ define the set of nodes returned by expressions and paths, starting from a context node x in the tree. The function $\mathcal{S}_q$ defines the semantics

```
ws2s;
# Data Model
var2 $ where ~empty($)
         & (all1 x : all1 y : ((y=x.1 | y=x.0) & (y in $)) => x in $)
         & all1 r : (r in $ & ~(ex1 z : z in $ & (r=z.1 | r=z.0)))
             => r.1 notin $;

# Characteristic sets
var2 Xbook, Xcitation;

# Partition
((all1 x : x in Xbook =>x notin Xcitation)
&(all1 x : x in Xcitation =>x notin Xbook));
```

Figure 5: A WS2S Formula Describing a Sample XML Tree in MONA Syntax.

of qualifiers that basically state the existence or absence of one or more paths from a context node x. The semantics of paths uses the navigational semantics of axes shown on Figure 7. Navigation performed by axes (as illustrated on a sample XML tree by Figure 1) relies on a few primitives over the XML data model:

- *root*() returns the root of the tree;
- *children*(x) returns the set of nodes which are children of the node x;
- *parent*(x) returns the parent node of the node x;
- the relation $\ll$ defines the ordering: $x \ll y$ holds if and only if the node x is before the node y in the depth-first traversal order of the n-ary XML tree;
- and *name*() returns the XML labeling of a node in a tree.

3.2 Navigation and Recursion

As a first step toward a WS2S encoding of XPath expressions, we need to express the navigational primitives over binary trees. Considering binary trees involves recursion for modeling the usual child relation on unranked trees (c.f. Figure 8). Recursion is not available as a basic construct of WS2S. We define recursion in monadic second-order logic via a transitive closure formulated using second-order quantification.

We begin by defining the following-sibling relation in WS2S. Let us consider a second-order variable F as the set of nodes of interest. We define the following-sibling relation as an induction scheme. The base case just captures that the immediate right successor of x is effectively its first following sibling:

$$(x.1 \in F)$$

$$
\begin{array}{lcl}
\mathcal{S}_e & : & \mathit{Expression} \longrightarrow \mathit{Node} \longrightarrow \mathrm{Set}(\mathit{Node}) \\
\mathcal{S}_e[\![/p]\!]_x & = & \mathcal{S}_p[\![p]\!]_{root()} \\
\mathcal{S}_e[\![p]\!]_x & = & \mathcal{S}_p[\![p]\!]_x \\
 & & \\
\mathcal{S}_p & : & \mathit{Path} \longrightarrow \mathit{Node} \longrightarrow \mathrm{Set}(\mathit{Node}) \\
\mathcal{S}_p[\![p_1/p_2]\!]_x & = & \{x_2 \mid x_1 \in \mathcal{S}_p[\![p_1]\!]_x \wedge x_2 \in \mathcal{S}_p[\![p_2]\!]_{x_1}\} \\
\mathcal{S}_p[\![p[q]]\!]_x & = & \{x_1 \mid x_1 \in \mathcal{S}_p[\![p]\!]_x \wedge \mathcal{S}_q[\![q]\!]_{x_1}\} \\
\mathcal{S}_p[\![e_1 \cup e_2]\!]_x & = & \mathcal{S}_e[\![e_1]\!]_x \cup \mathcal{S}_e[\![e_2]\!]_x \\
\mathcal{S}_p[\![e_1 \cap e_2]\!]_x & = & \{x_1 \mid x_1 \in \mathcal{S}_e[\![e_1]\!]_x \wedge x_1 \in \mathcal{S}_e[\![e_2]\!]_x\} \\
\mathcal{S}_p[\![(p)]\!]_x & = & \mathcal{S}_p[\![p]\!]_x \\
\mathcal{S}_p[\![\mathrm{a}::\sigma]\!]_x & = & \{x_1 \mid x_1 \in \mathcal{S}_a[\![\mathrm{a}]\!]_x \wedge \mathit{name}(x_1) = \sigma\} \\
\mathcal{S}_p[\![\mathrm{a}::*]\!]_x & = & \{x_1 \mid x_1 \in \mathcal{S}_a[\![\mathrm{a}]\!]_x\} \\
 & & \\
\mathcal{S}_q & : & \mathit{Qualifier} \longrightarrow \mathit{Node} \longrightarrow \mathit{Boolean} \\
\mathcal{S}_q[\![q_1 \text{ and } q_2]\!]_x & = & \mathcal{S}_q[\![q_1]\!]_x \wedge \mathcal{S}_q[\![q_2]\!]_x \\
\mathcal{S}_q[\![q_1 \text{ or } q_2]\!]_x & = & \mathcal{S}_q[\![q_1]\!]_x \vee \mathcal{S}_q[\![q_2]\!]_x \\
\mathcal{S}_q[\![\text{not } q]\!]_x & = & \neg\, \mathcal{S}_q[\![q]\!]_x \\
\mathcal{S}_q[\![e]\!]_x & = & \mathcal{S}_e[\![e]\!]_x \neq \emptyset
\end{array}
$$

Figure 6: Denotational Semantics of Expressions, Paths and Qualifiers.

$$
\begin{array}{lcl}
\mathcal{S}_a & : & \mathit{Axis} \longrightarrow \mathit{Node} \longrightarrow \mathrm{Set}(\mathit{Node}) \\
\mathcal{S}_a[\![\mathrm{child}]\!]_x & = & \mathit{children}(x) \\
\mathcal{S}_a[\![\mathrm{parent}]\!]_x & = & \mathit{parent}(x) \\
\mathcal{S}_a[\![\mathrm{descendant}]\!]_x & = & \mathit{children}^+(x) \\
\mathcal{S}_a[\![\mathrm{ancestor}]\!]_x & = & \mathit{parent}^+(x) \\
\mathcal{S}_a[\![\mathrm{self}]\!]_x & = & \{x\} \\
\mathcal{S}_a[\![\mathrm{descendant\text{-}or\text{-}self}]\!]_x & = & \mathcal{S}_a[\![\mathrm{descendant}]\!]_x \cup \mathcal{S}_a[\![\mathrm{self}]\!]_x \\
\mathcal{S}_a[\![\mathrm{ancestor\text{-}or\text{-}self}]\!]_x & = & \mathcal{S}_a[\![\mathrm{ancestor}]\!]_x \cup \mathcal{S}_a[\![\mathrm{self}]\!]_x \\
\mathcal{S}_a[\![\mathrm{preceding}]\!]_x & = & \{y \mid y \ll x\} - \mathcal{S}_a[\![\mathrm{ancestor}]\!]_x \\
\mathcal{S}_a[\![\mathrm{following}]\!]_x & = & \{y \mid x \ll y\} - \mathcal{S}_a[\![\mathrm{descendant}]\!]_x \\
\mathcal{S}_a[\![\mathrm{following\text{-}sibling}]\!]_x & = & \{y \mid y \in \mathit{child}(\mathit{parent}(x)) \wedge x \ll y\} \\
\mathcal{S}_a[\![\mathrm{preceding\text{-}sibling}]\!]_x & = & \{y \mid y \in \mathit{child}(\mathit{parent}(x)) \wedge y \ll x\}
\end{array}
$$

Figure 7: Denotational Semantics of Axes.

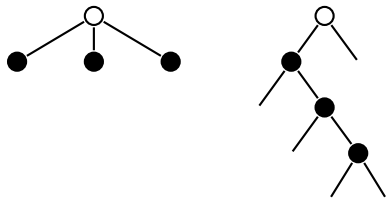

Figure 8: Children in the Unranked and Binary Cases.

Then the induction step states that the immediate right successor of every position in F is also among the following siblings, and formulates this as a transitive closure:

$$\forall z.(z \in F \Rightarrow z.1 \in F)$$

We are now about to formulate the global requirement for a node y to be itself one of the following siblings of x. The node y must belong to the set F which is closed under the following-sibling relation starting from $x.1$:

$$(x.1 \in F \wedge \forall z.z \in F \Rightarrow z.1 \in F) \Rightarrow y \in F$$

Note that this formula is satisfied for multiple sets F. For instance, the set of all tree nodes satisfies this implication. Actually, we are only interested in the smallest set F for which the formula holds: the set which contains all and only all following siblings. A way to express this is to introduce a universal quantification over F. Indeed, ranging over all such set of nodes notably takes into account the particular case where F is minimal, i.e. the set we are interested in. If the global formula holds for every F, y is also in the minimal set that contains only the following siblings of x. Therefore, we define the XPath "following-sibling" axis as the WS2S predicate:

$$\begin{aligned} \text{followingsibling}(X, x, y) \quad \stackrel{\text{def}}{=} \quad & \forall F.F \subseteq X \Rightarrow \\ & ((x.1 \in F \wedge \forall z.z \in F \Rightarrow z.1 \in F) \Rightarrow y \in F) \end{aligned}$$

that expresses the requirements for a node y to be a following sibling of a node x in the tree X. XPath "descendant" axis can be modeled in the same manner. The set D of interest is initialized with the left child of the context node, and is closed under both successor relations:

$$\begin{aligned} \text{descendant}(X, x, y) \quad \stackrel{\text{def}}{=} \quad & \forall D.D \subseteq X \Rightarrow \\ & (x.0 \in D \wedge \forall z.(z \in D \Rightarrow z.1 \in D \wedge z.0 \in D) \Rightarrow y \in D) \end{aligned}$$

Considering these two relations as navigational primitives, we can build more complex ones out of them:

$$\begin{array}{lcl}
\text{child}(X,x,y) & \stackrel{\text{def}}{=} & y = x.0 \ \vee \ \text{followingsibling}(X, x.0, y) \\
\text{following}(X,x,y) & \stackrel{\text{def}}{=} & \exists z.z \in X \wedge z.1 \in X \wedge \text{ancestor}(X,x,z) \\
 & & \wedge \ \text{descendant}(X, z.1, y) \\
\text{self}(X,x,y) & \stackrel{\text{def}}{=} & x = y \\
\text{descendantorself}(X,x,y) & \stackrel{\text{def}}{=} & \text{self}(X,x,y) \ \vee \ \text{descendant}(X,x,y)
\end{array}$$

Eventually, the other XPath axes are defined as syntactic sugars by taking advantage of XPath symmetry:

$$\begin{array}{lcl}
\text{ancestor}(X,x,y) & \stackrel{\text{def}}{=} & \text{descendant}(X,y,x) \\
\text{parent}(X,x,y) & \stackrel{\text{def}}{=} & \text{child}(X,y,x) \\
\text{precedingsibling}(X,x,y) & \stackrel{\text{def}}{=} & \text{followingsibling}(X,y,x) \\
\text{ancestororself}(X,x,y) & \stackrel{\text{def}}{=} & \text{descendantorself}(X,y,x) \\
\text{preceding}(X,x,y) & \stackrel{\text{def}}{=} & \text{following}(X,y,x)
\end{array}$$

3.3 Logical Composition of Steps

This section describes how path composition operators are translated into logical connectives. The translation $\mathcal{W}_e$ is formally specified as a "derivor" shown on Figure 9 and written $\mathcal{W}_e[\![e]\!]_x^y$ where:

- the parameter e (surrounded by special "syntax" braces $[\![]\!]$) is the source language parameter that is rewritten;
- the additional parameters x and y are respectively the context and the result node of the query.

The compilation of an XPath expression to WS2S relies on $\mathcal{W}_p$ in charge of translating paths into formulas, and the dual derivor $\mathcal{W}_q$ for translating qualifiers into formulas. The basic principle is that $\mathcal{W}_p[\![p]\!]_x^y$ holds for all pairs x, y of nodes such that y is accessed from x through the path p. Similarly, $\mathcal{W}_q[\![q]\!]_x$ holds for all nodes x such that the qualifier q is satisfied from the context node x.

The interpretation of path composition $\mathcal{W}_p[\![p_1/p_2]\!]_x^y$ consists in checking the existence of an intermediate node that connects the two paths, and therefore requires a new fresh variable to be inserted. The same holds for $\mathcal{W}_e[\![/p]\!]_x^y$ that restarts from the root to interpret p, whatever the current context node x is.

Paths can occur inside qualifiers therefore $\mathcal{W}_e$, $\mathcal{W}_p$ and $\mathcal{W}_q$ are mutually recursive. Since the interpretations of paths and qualifiers are respectively dyadic and monadic formulas, the translation of a path inside a qualifier $\mathcal{W}_q[\![e]\!]_x$ requires the insertion of a new fresh variable whose only purpose consists in testing the existence of the path.

$$
\begin{array}{lcl}
\mathcal{W}_e & : & \mathit{Expression} \longrightarrow \mathit{Node} \longrightarrow \mathit{Node} \longrightarrow \mathit{WS2S} \\
\mathcal{W}_e[\![/p]\!]_x^y & = & \exists z.\mathit{isroot}(z) \wedge \mathcal{W}_p[\![p]\!]_z^y \\
\mathcal{W}_e[\![p]\!]_x^y & = & \mathcal{W}_p[\![p]\!]_x^y \\
 & & \\
\mathcal{W}_p & : & \mathit{Path} \longrightarrow \mathit{Node} \longrightarrow \mathit{Node} \longrightarrow \mathit{WS2S} \\
\mathcal{W}_p[\![p_1/p_2]\!]_x^y & = & \exists z.\mathcal{W}_p[\![p_1]\!]_x^z \wedge \mathcal{W}_p[\![p_2]\!]_z^y \\
\mathcal{W}_p[\![p[q]]\!]_x^y & = & \mathcal{W}_p[\![p]\!]_x^y \wedge \mathcal{W}_q[\![q]\!]_y \\
\mathcal{W}_p[\![e_1 \cup e_2]\!]_x^y & = & \mathcal{W}_e[\![e_1]\!]_x^y \vee \mathcal{W}_e[\![e_2]\!]_x^y \\
\mathcal{W}_p[\![e_1 \cap e_2]\!]_x^y & = & \mathcal{W}_e[\![e_1]\!]_x^y \wedge \mathcal{W}_e[\![e_2]\!]_x^y \\
\mathcal{W}_p[\![(p)]\!]_x^y & = & \mathcal{W}_p[\![p]\!]_x^y \\
\mathcal{W}_p[\![a::\sigma]\!]_x^y & = & a(x,y) \wedge y \in X_\sigma \\
\mathcal{W}_p[\![a::*]\!]_x^y & = & a(x,y) \\
 & & \\
\mathcal{W}_q & : & \mathit{Qualifier} \longrightarrow \mathit{Node} \longrightarrow \mathit{WS2S} \\
\mathcal{W}_q[\![q_1 \text{ and } q_2]\!]_x & = & \mathcal{W}_q[\![q_1]\!]_x \wedge \mathcal{W}_q[\![q_2]\!]_x \\
\mathcal{W}_q[\![q_1 \text{ or } q_2]\!]_x & = & \mathcal{W}_q[\![q_1]\!]_x \vee \mathcal{W}_q[\![q_2]\!]_x \\
\mathcal{W}_q[\![\text{not } q]\!]_x & = & \neg\, \mathcal{W}_q[\![q]\!]_x \\
\mathcal{W}_q[\![e]\!]_x & = & \exists y.\mathcal{W}_e[\![e]\!]_x^y
\end{array}
$$

Figure 9: Translating XPath into WS2S.

Eventually, the translation of steps relies on the logical definition of axes: $a(x, y)$ denotes the WS2S predicate defining the XPath axis a, as described in Section 3.2. For instance, Figure 10 presents the WS2S translation of the XPath expression:

child::book/descendant::citation[parent::section]

4 The XPath Containment Problem

We are now ready to formulate the XPath containment problem in terms of a logical formula.

4.1 Logical Formulation

Given two XPath expressions e_1 and e_2, we build the WS2S formula corresponding to checking their containment in two steps. We first translate each XPath expression into a WS2S logical relation that connects two nodes in the tree, as presented in Section 3.3. Then we have to unify the data model. Each translation yields a set of characteristic sets. We build the union of them, so that characteristic sets that correspond to symbols used in both expressions are identified.

From a logical point of view, $e_1 \subseteq e_2$ means that each pair of nodes (x, y) such that x and y are connected by the logical relation corresponding to e_1 is similarly connected by the

```
# Translated XPath expression:
# child::book/descendant::citation[parent::section]
ws2s;
# Data Model
var2 $ where ~empty($)
         & (all1 x : all1 y : ((y=x.1 | y=x.0) & (y in $)) => x in $)
         & all1 r : (r in $ & ~(ex1 z : z in $ & (r=z.1 | r=z.0)))
              => r.1 notin $;

# Characteristic sets
var2 Xbook, Xcitation, Xsection;

# Partition
((all1 x : x in Xbook =>x notin Xcitation & x notin Xsection)&
(all1 x : x in Xcitation =>x notin Xbook & x notin Xsection)&
(all1 x : x in Xsection =>x notin Xbook & x notin Xcitation));

# Query (parameters are context and result nodes)
pred xpath1 (var1 x, var1 y)= ex1 x1 : child(x,x1) & x1 in Xbook
                              & descendant(x1,y) & y in Xcitation
                              & ex1 x2 : parent(y,x2) & x2 in Xsection;
```

Figure 10: WS2S Translation of a Sample XPath in MONA Syntax.

logical relation obtained from e_2:

$$\forall x.\ \forall y.\ \mathcal{W}_e[\![e_1]\!]_x^y \Rightarrow \mathcal{W}_e[\![e_2]\!]_x^y \tag{1}$$

The containment relation holds between expressions e_1 and e_2 if and only if the WS2S formula (1) is satisfied for all trees. With respect to the notations of Section 2.6, the containment between expressions e_1 and e_2 is thus formulated as:

$$\forall X.\ \text{XMLTree}(X, X_1, ..., X_n) \Rightarrow (\forall x \in X.\ \forall y \in X.\ \mathcal{W}_e[\![e_1]\!]_x^y \Rightarrow \mathcal{W}_e[\![e_2]\!]_x^y)$$

where the X_i are members of the union of all characteristic sets detected for each expression. Consider for instance the two XPath expressions:

$$\begin{array}{lcl} e_1 & = & \text{child::book/descendant::citation[parent::section]} \\ e_2 & = & \text{descendant::citation[ancestor::book and ancestor::section]} \end{array}$$

Figure 11 presents the generated WS2S formula for checking containment between e_1 and e_2, in MONA syntax. The formula is determined valid (which means $e_1 \subseteq e_2$) in less than 0.2 seconds, the time spent to build the corresponding automaton and analyze it. If we reciprocally check the containment between e_2 and e_1, the formula is satisfiable, which means $e_2 \not\subseteq e_1$. The generated counter-example XML tree is shown on Figure 12. The total running time of the decision procedure is less than 0.9 seconds, including the generation of the counter-example.

4.2 Soundness and Completeness

Soundness and completeness of our decision procedure are ensured by construction. Indeed, if we restart from the initial definition of the containment problem: provided a XML tree, checking containment betwen two XPath e_1 and e_2 consists in determining if the following proposition holds:

$$\forall x, \mathcal{S}_e[\![e_1]\!]_x \subseteq \mathcal{S}_e[\![e_2]\!]_x \tag{2}$$

By definition, (2) is logically equivalent to:

$$\forall x, \forall y, y \in \mathcal{S}_e[\![e_1]\!]_x \Rightarrow y \in \mathcal{S}_e[\![e_2]\!]_x \tag{3}$$

Then the last step remaining to prove is the equivalence between (3) and (1). To this end, we need to prove that our compilation of XPath expressions into WS2S formulas preserves XPath denotational semantics, which means:

Theorem 4.2.1 *The logical translation of XPath expressions is equivalent to XPath denotational semantics:*

$$\mathcal{W}_p[\![e]\!]_x^y \equiv y \in \mathcal{S}_p[\![e]\!]_x \tag{4}$$

```
ws2s;
# Checking XPath Containment between
# 'child::book/descendant::citation[parent::section]' and
# 'descendant::citation[ancestor::book and ancestor::section]'

# Data Model
var2 $ where ~empty($)
        & (all1 x : all1 y : ((y=x.1 | y=x.0) & (y in $)) => x in $)
        & all1 r : (r in $ & ~(ex1 z : z in $ & (r=z.1 | r=z.0)))
            => r.1 notin $;

# Characteristic sets
var2 Xbook, Xcitation, Xsection;

# Queries (parameters are context and result nodes)
pred xpath1 (var1 x, var1 y)= ex1 x1 : child(x,x1) & x1 in Xbook
                              & descendant(x1,y) & y in Xcitation
                              & ex1 x2 : parent(y,x2) & x2 in Xsection;
pred xpath2 (var1 x, var1 y)= descendant(x,y) & y in Xcitation
                              & ex1 x1 : (ancestor(y,x1) & x1 in Xbook)
                              & ex1 x2 : (ancestor(y,x2)
                              & x2 in Xsection);
# Problem formulation
((all1 x : x in Xbook =>x notin Xcitation & x notin Xsection)&
(all1 x : x in Xcitation =>x notin Xbook & x notin Xsection)&
(all1 x : x in Xsection =>x notin Xbook & x notin Xcitation))
=>
(all1 x : all1 y : (xpath1(x,y)=> xpath2(x,y)));
```

Figure 11: WS2S Formula for Containment of two XPath Expressions in MONA Syntax.

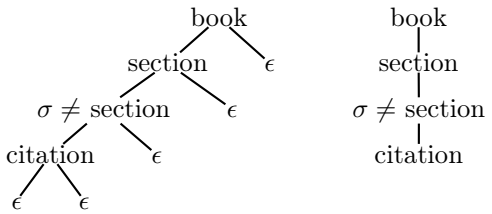

Figure 12: Binary and n-ary Representations of a Counter-Example XML Tree for which $e_2 \not\subseteq e_1$.

Proof (Sketch)
The proof is achieved using induction over the structure of paths. Since the definition of paths and qualifiers is cross-recursive, we use a mutual induction scheme. The scheme relies on the dual property for qualifiers that also needs to be proved:

$$\forall p, \forall x, (\mathcal{S}_q[\![q]\!]_x \equiv \mathcal{W}_q[\![q]\!]_x) \tag{5}$$

Specifically (4) is proved by taking (5) as assumption, and reciprocally (5) is proved under (4) as assumption. Both equivalences (4) and (5) are proved inductively for each compositional layer. The idea basically consists in associating corresponding logical connectives to each set-theoretic composition operator used in the denotational semantics. XPath qualifier constructs trivially correspond to logical WS2S connectives. Path constructs involves set-theoretic union and intersection operations which are respectively mapped to logical disjunction and conjunction. Two path constructs: p_1/p_2 and $p[q]$ require specific attention in the sense their denotational semantics introduce particular compositions over sets of nodes. We recall them below:

$$\begin{array}{lcl} \mathcal{S}_p[\![p_1/p_2]\!]_x & = & \{x_2 \mid x_1 \in \mathcal{S}_p[\![p_1]\!]_x \wedge x_2 \in \mathcal{S}_p[\![p_2]\!]_{x_1}\} \\ \mathcal{S}_p[\![p[q]]\!]_x & = & \{x_1 \mid x_1 \in \mathcal{S}_p[\![p]\!]_x \wedge \mathcal{S}_q[\![q]\!]_{x_1}\} \end{array}$$

We introduce auxiliary lemmas to clarify how these constructs are mapped to WS2S. The XPath construct p_1/p_2 is generalized as a function *product*(), whereas the XPath construct $p[q]$ is generalized as *filter*():

$$\begin{array}{lcl} \mathit{product}() & : & \mathrm{Set}(\mathit{Node}) \to (\mathit{Node} \to \mathrm{Set}(\mathit{Node})) \to \mathrm{Set}(\mathit{Node}) \\ \mathit{filter}() & : & \mathrm{Set}(\mathit{Node}) \to (\mathit{Node} \to \mathit{Boolean}) \to \mathrm{Set}(\mathit{Node}) \end{array}$$

product() is characterized by the lemmas (6) and (7), in which y and z are nodes, and S is a set of nodes. These lemmas abstract over XPath navigational functionalities performed by axes by letting f denoting a function that returns a set of nodes provided a current node:

$$\forall y, \forall z, \forall S, \forall f : \mathit{Node} \to \mathrm{Set}(\mathit{Node}), z \in S \Rightarrow y \in (f\,z) \Rightarrow y \in \mathit{product}(S, f) \tag{6}$$

$$\forall y, \forall S, \forall f : \mathit{Node} \to \mathrm{Set}(\mathit{Node}), y \in \mathit{product}(S, f) \Rightarrow \exists z, z \in S \wedge y \in (f\,z). \tag{7}$$

The function *filter*() is in turn characterized by the following lemma:

$$\forall y, \forall g : \mathit{Node} \to \mathit{Boolean}, y \in \mathit{filter}(S, g) \Rightarrow y \in S \tag{8}$$

The auxiliary lemmas (6), (7), and (8) are also proved by induction. Developping the proof in constructive logic involves the (trivial) decidability of set-theoretic inclusion and of the denotational semantics of qualifiers. The full formal proof is detailed in [22]. It has been mechanically checked by the machine using the Coq formal proof management system [25].

5 Complexity Analysis and Practical Results

In this section, we review the global complexity of our approach and its implications in practice. Our method basically consists in building the formula from the input queries and deciding it. The translation of an XPath query to its logical representation is linear in the size of the input query. Indeed, each expression is decomposed then translated inductively in one pass without any duplication, as shown by the formal definition of $\mathcal{W}_e$ in Section 3.3. The second step is the decision procedure, which, compared to the translation, represents the major part of the cost.

As introduced earlier, a WS2S formula is decided throughout the logic-automaton connection described in Sections 2.4 and 2.5. This translation from logical formulas to tree automata, while effective, is unfortunately *non-elementary*[3]. Indeed, WS1S is known to have a unbounded stack of exponentials as worst case lower bound [31, 36], and WS2S is at least quadratically more difficult to work with [26].

Nevertheless, recent works on MSO solvers - especially those using BDDs techniques [9] such as MONA [27] - suggest that in particular practical cases the explosiveness of this technique can be effectively controlled. Our goal in the remaining part of this section is to show that our WS2S formulation and its associated decision procedure give rather efficient results in practice.We first describe what makes deciding WS2S non-elementary. We then introduce a subsequent problem-specific optimization, before presenting practical results obtained for deciding the containment of XPath expressions using the MONA solver.

5.1 Sources of complexity for a WS2S Decision Procedure

Two factors have a major impact on the cost of a WS2S decision procedure:

1. the number of second-order variables in the formula
2. the number of states of the corresponding automaton (automaton size)

The number of second-order variables determines the alphabet size. More precisely, a formula with k variables is decided by an automaton operating on the alphabet $\Sigma = \{0,1\}^k$. Representing the transition function δ of such an automaton can be prohibitive. Indeed, in the worst case, the representation of a complete FTA requires $2^k \cdot |Q|^3$ transitions where Q is the set of states of the automaton. A direct encoding with classical FTA such as the one described in Section 2.5 would lead to an impracticable algorithm. Modern logical solvers

[3]We recall that the term *elementary* introduced by Grzegorczyk [23] 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 has shown that every elementary function in one argument is bounded by $\lambda n.tower(n,c)$ for some constant c. Hence, the term *non-elementary* refers to a function that grows faster than any such function.

represent transition functions using BDDs [9] that can lead to exponential improvements [27, 39].

As seen in Section 2.5, automaton construction is performed inductively by composing automata corresponding to each sub-formula. During this process, the number of states of intermediate automata may grow significantly. Automaton size depends on the nature of the automata-theoretic operation applied and the sizes of automata constructed so far. Each operation on tree automata particularly affects the size of the resulting automaton:

- Automata intersection causes a quadratic increase in automaton size in the worst case, as well as all binary WS2S connectors ($\wedge$, $\vee$, $\Rightarrow$) that involve automata products [28].
- In our case, since MONA works with deterministic complete automata, automata complementation corresponding to WS2S negation is a linear-time algorithm that consists in flipping accepting and rejecting states.
- The major source of complexity originates from automata determinization which may cause an exponential increase of the number of states in the worst case [13]. Logical quantification involves automaton projection (c.f. Section 2.5) which may result in a non-deterministic automaton, thus involving determinization. Hopefully, a succession of quantifications of the same type can be combined as a single projection followed by a single determinization. However, any alternation of second-order quantifiers requires a determinization, thus possibly causing an exponential increase of the automaton size.

As a consequence, the number of states of the final automaton corresponding to a formula with n quantifier alternations is in the worst case a tower of exponentials of height $c \cdot n$ where c is some constant, and this is a lower bound [37]. This bound may sound discouraging. Fortunately, the worst-case scenario which corresponds to complex formulas, is not likely to occur for the containment in practice. Additionally, we describe in the following section a significant optimization that takes advantage of XPath peculiarities for combating automaton size explosion.

5.2 Optimization Based on Guided Tree Automata

A major source of complexity arises from the translation of composed paths. Each translation of the form $\mathcal{W}_p[\![p_1/p_2]\!]^y_x$ introduces an existentially quantified first-order variable which ranges over all possible tree positions (c.f. Figure 3.3).

The idea in this section is to take advantage of XPath navigational peculiarities for attempting to reduce the scope associated to such variables. XPath navigates the tree step by step: each step selects a set of nodes which is in turn used to select a new one by the next step. The interpretation of a variable inserted during the translation of p_1/p_2 corresponds to the intermediate node which is a result of p_1 and the context node of p_2. The truth status of the formula is determined by the existence of such an intermediate node at a particular position in the tree. If we can determine regions in the tree in which such a node may appear from those where it cannot appear, we gain valuable positional knowledge that can be used

```
e1(x,y) = ex1 x1 : isroot(x1) & x1 in $
 & ex1 x2 : child(x1,x2) &  x2 in Xbook & descendant(x2,y)
 & y in $ & ex1 x3 : child(y,x3) & x3 in Xcitation;
```

Figure 13: WS2S Translation of e_3 in MONA Syntax.

to reduce the variable scope. Actually, we need to identify the region in the tree (or even some larger approximation) in which the node must be located in order for the formula to be satisfied. XPath sequential structure of steps makes it possible to exploit such positional knowledge. Indeed, consider for instance the expression:

$$e_3 = \text{/child::book/descendant::*[child::citation]}$$

e_3 navigates from the document root through its "book" children elements and then selects all descendant nodes provided they have at least one child named "citation". Several conditions must be satisfied by a tree t_1 in order to yield a result for e_3:

- t_1 must have at least one "book" element as a child of the root;
- t_1 must have at least one element that must be a descendant of the "book" element;
- for this node to be selected it must have at least one child named "citation".

This is made explicit by the logical translation $\mathcal{W}_e[\![e_3]\!]_x^y$ in MONA syntax shown on Figure 13. In this translation, `x1`, `x2` and `x3` denote the respective positions of the root node, a "book" child, and a "citation" child of the selected position `y`. These variables actually only range over a particular set of positions in the tree. By definition, the root can only appear at depth level 0, the "book" element can only occur at level 1 and its descendants occur at any depth level l greater or equals to 2. Eventually, the "citation" element should occur at level $l + 1$. This is because each step introduces its particular positional constraint which can be propagated to the next steps.

The idea of taking advantage of positional knowledge is even more general. Theoretically, normal bottom-up FTA are sufficient for deciding validity of a WS2S formula (as presented in Section 2.4). However composition of such automata is particularly sensitive to state space explosion, as presented in Section 5.1. Guided tree automata (GTA) [8] have been introduced in order to combat such state space explosion by following the divide to conquer approach. A GTA is just an ordinary FTA equipped with an additional deterministic top-down tree automaton called the guide. The latter is introduced to take advantage of positional knowledge, and used for partitioning the FTA state space into independent subspaces. Top-down deterministic automata are strictly less powerful than ordinary (bottom-up or non-deterministic top-down) FTA [13]. However, this is not a problem since the guide is only intended to provide additional auxiliary information used for optimization purposes. As a consequence, the more precise is the guide, the more efficient is the decision procedure, but an approximation is sufficient. The guide basically splits the state space of the FTA in

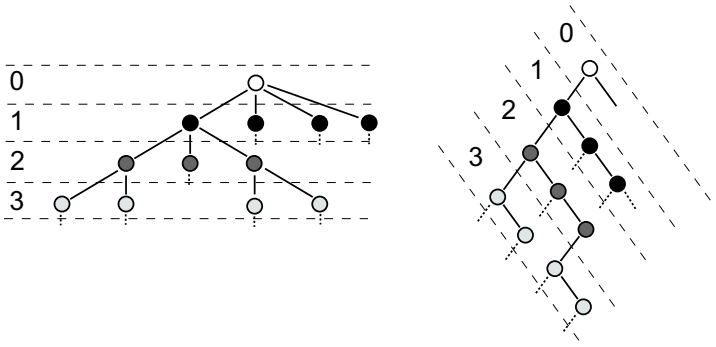

Figure 14: Depth Levels in the Unranked and Binary Cases.

independent subsets. Therefore the transition relation of the bottom-up automaton is split into a family of transition functions, one for each state space name. In our case, a state space name corresponds to a particular depth level or a set of depth levels. GTA can be composed in the same way than ordinary FTA as explained in Section 2.4. A GTA can be seen as an ordinary tree automaton, where the state space has been factorized according to the guide. A GTA with only one state space is just an ordinary tree automaton. A detailed description of GTA can be found in [8]. GTA-based optimization may lead to exponential improvements of the decision procedure [17].

In our case we introduce a tree partitioning based on the depth levels, which is depicted by Figure 14 for a n-ary sample tree and its binary counterpart. Based on this partitioning, we define a positional constraint associated to each node variable as a set of depth levels. Indeed, a node refereed by an XPath can occur at several depth levels since some axes involve transitive closure (c.f. Section 3.1). Moreover, the set of depth levels can even be infinite since XPath offers recursion in unbounded trees.

The computation of sets of depth levels is calculated by the function L_e shown on Figure 15, and written $L_e[\![e]\!](N)$ where e is the XPath expression to be analyzed and N is the set of positional constraints corresponding to the context node from which e is applied. Again, our algorithm proceeds inductively on the structure of XPath expressions. XPath steps are base cases for which the set of levels is effectively calculated from the previous one. Transitive closure axes such as "descendant" turn the set of depth levels into a infinite one, even if the previous was finite. Path composition basically propagate the level calculations by combining with the base cases. Note that an important precision can be gained with absolute XPath expressions. In this case, the initial set of depth levels is the singleton $\{0\}$ as opposed to relative XPath expressions for which the context node is not known and the initial set of depth levels is subsequently $\mathbb{N}$.

The optimized compilation of XPath expressions to WS2S formulas is given on Figure 16. $\mathcal{W}'_e$, $\mathcal{W}'_p$ and $\mathcal{W}'_q$ are respective optimized versions of $\mathcal{W}_e$, $\mathcal{W}_p$ and $\mathcal{W}_q$, which convey a set of depth levels as an additional parameter passed to L_e and L_p. These functions compute the

$$
\begin{array}{lcl}
L_e & : & \mathit{Expression} \longrightarrow \mathrm{Set}(\mathit{Integer}) \longrightarrow \mathrm{Set}(\mathit{Integer}) \\
L_e[\![/p]\!](N) & = & L_p[\![p]\!](\{0\}) \\
L_e[\![p]\!](N) & = & L_p[\![p]\!](\mathbb{N}) \\
 & & \\
L_p & : & \mathit{Path} \longrightarrow \mathrm{Set}(\mathit{Integer}) \longrightarrow \mathrm{Set}(\mathit{Integer}) \\
L_p[\![p_1/p_2]\!](N) & = & L_p[\![p_2]\!](L_p[\![p_1]\!](N)) \\
L_p[\![p[q]]\!](N) & = & L_p[\![p]\!](N) \\
L_p[\![e_1 \dashv e_2]\!](N) & = & L_e[\![e_1]\!](N) \cup L_e[\![e_2]\!](N) \\
L_p[\![e_1 \cap e_2]\!](N) & = & L_e[\![e_1]\!](N) \cap L_e[\![e_2]\!](N) \\
L_p[\![(p)]\!](N) & = & L_p[\![p]\!](N) \\
L_p[\![\text{self::}n]\!](N) & = & N \\
L_p[\![\text{child::}n]\!](N) & = & \{n+1 \mid n \in N\} \\
L_p[\![\text{parent::}n]\!](N) & = & \{n-1 \mid n \in N\} \\
L_p[\![\text{descendant::}n]\!](N) & = & \{n' \mid n \in N \wedge n' > n\} \\
L_p[\![\text{descendant-or-self::}n]\!](N) & = & \{n' \mid n \in N \wedge n' >= n\} \\
L_p[\![\text{ancestor::}n]\!](N) & = & \{n' \mid n \in N \wedge n' >= 0 \wedge n' < n\} \\
L_p[\![\text{ancestor-or-self::}n]\!](N) & = & \{n' \mid n \in N \wedge n' >= 0 \wedge n' <= n\} \\
L_p[\![\text{following::}n]\!](N) & = & \mathbb{N} - \{0\} \\
L_p[\![\text{preceding::}n]\!](N) & = & \mathbb{N} - \{0\} \\
L_p[\![\text{following-sibling::}n]\!](N) & = & N \\
L_p[\![\text{preceding-sibling::}n]\!](N) & = & N
\end{array}
$$

Figure 15: Computation of the Depth Levels of Nodes Selected by a Path.

$$
\begin{array}{lcl}
\mathcal{W}'_e & : & \mathit{Expression} \rightarrow \mathit{Node} \rightarrow \mathit{Node} \rightarrow \mathrm{Set}(\mathit{Integer}) \rightarrow \mathit{WS2S} \\
\mathcal{W}'_e[\![/p]\!](x,y,N) & = & \exists z\ [\{0\}]\ .\mathit{isroot}(z) \wedge \mathcal{W}'_p[\![p]\!](z,y,\{0\}) \\
\mathcal{W}'_e[\![p]\!](x,y,N) & = & \mathcal{W}'_p[\![p]\!](x,y,\mathbb{N}) \\
 & & \\
\mathcal{W}'_p & : & \mathit{Path} \rightarrow \mathit{Node} \rightarrow \mathit{Node} \rightarrow \mathrm{Set}(\mathit{Integer}) \rightarrow \mathit{WS2S} \\
\mathcal{W}'_p[\![p_1/p_2]\!](x,y,N) & = & \exists z\ [L_p[\![p_1]\!](N)]\ .\mathcal{W}'_p[\![p_1]\!](x,z,N) \wedge \mathcal{W}'_p[\![p_2]\!](z,y,N) \\
\mathcal{W}'_p[\![p[q]]\!](x,y,N) & = & \mathcal{W}'_p[\![p]\!](x,y,N) \wedge \mathcal{W}'_q[\![q]\!](y,N) \\
\mathcal{W}'_p[\![e_1 \dashv e_2]\!](x,y,N) & = & \mathcal{W}'_e[\![e_1]\!](x,y,N) \vee \mathcal{W}'_e[\![e_2]\!](x,y,N) \\
\mathcal{W}'_p[\![e_1 \cap e_2]\!](x,y,N) & = & \mathcal{W}'_e[\![e_1]\!](x,y,N) \wedge \mathcal{W}'_e[\![e_2]\!](x,y,N) \\
\mathcal{W}'_p[\![(p)]\!](x,y,N) & = & \mathcal{W}'_p[\![p]\!](x,y,N) \\
\mathcal{W}'_p[\![a::\sigma]\!](x,y,N) & = & a(x,y) \wedge y \in X_\sigma \\
\mathcal{W}'_p[\![a::*]\!](x,y,N) & = & a(x,y) \\
 & & \\
\mathcal{W}'_q & : & \mathit{Qualifier} \rightarrow \mathit{Node} \rightarrow \mathrm{Set}(\mathit{Integer}) \rightarrow \mathit{WS2S} \\
\mathcal{W}'_q[\![q_1 \text{ and } q_2]\!](x,N) & = & \mathcal{W}'_q[\![q_1]\!](x,N) \wedge \mathcal{W}'_q[\![q_2]\!](x,N) \\
\mathcal{W}'_q[\![q_1 \text{ or } q_2]\!](x,N) & = & \mathcal{W}'_q[\![q_1]\!](x,N) \vee \mathcal{W}'_q[\![q_2]\!](x,N) \\
\mathcal{W}'_q[\![\text{not } q]\!](x,N) & = & \neg\, \mathcal{W}'_q[\![q]\!](x,N) \\
\mathcal{W}'_q[\![e]\!](x,N) & = & \exists y\ [L_e[\![e]\!](N)]\ .\mathcal{W}'_e[\![e]\!](x,y,N)
\end{array}
$$

Figure 16: Translation of XPath Expressions to WS2S Formulas with Restricted Variable Scopes.

```
guide l0 -> (l1, epsilon),
      l1 -> (l2, l1),
      l2 -> (l3, l2),
      l3 -> (lothers, l3),
      lothers -> (lothers, lothers),
      epsilon -> (epsilon, epsilon);

e1(x,y)= ex1 [l0] x1 : (isroot(x) & x=x1 & x in $)
& ex1 [l1] x2 : child(x1,x2) & x2 in Xbook & descendant(x2,y)
& y in $ & ex1 [l3, lothers] x3 : child(y,x3) & x3 in Xcitation;
```

Figure 17: Optimized WS2S Translation of e_3 in MONA Syntax.

restrictions on variable scope that are inserted by $\mathcal{W}'_p$ and $\mathcal{W}'_q$. We denote by "$\exists z\ [D]$ " the fact that the existentially quantified first-order variable z is restricted to appear at a depth level among the set of depth levels D. In practice, L_e and L_p can be merged into $\mathcal{W}'_e$ and can be implemented in a single pass over the XPath expression. Thus the translation and the depth level computation remain linear in the size of the query.

MONA provides an implementation of GTA. The application of the previous algorithm to e_3 leads to the logical formulation shown on Figure 17 in MONA syntax.

The guide obtained in this translation means that the root is labeled with "l0"; its left and right successor nodes are labeled with "l1" and "epsilon" respectively. The "epsilon" is a dummy state space reflecting the fact that the underlying shape is a tree and not a hedge. No variable is associated with this state space. The "lothers" state space represents any tree node occurring at a depth level greater than 3. Such a state space is associated with variables whose scope is of unbounded depth. The size of the guide depends on the maximum depth level found among the computed restrictions. Formally, a guide for a maximum depth level n is a top-down deterministic tree automaton with $\{q_0, ..., q_{n+1}\} \cup \{q_\epsilon\}$ as set of states, q_0 as the single initial state, and the following set of transitions:

$$\begin{array}{ll} & \{q_0 \rightarrow (q_1, q_\epsilon)\} \\ \cup & \{q_i \rightarrow (q_{i+1}, q_i) \mid i \in [1...n]\} \\ \cup & \{q_{n+1} \rightarrow (q_{n+1}, q_{n+1})\} \\ \cup & \{q_\epsilon \rightarrow (q_\epsilon, q_\epsilon)\} \end{array}$$

where q_i ($i \in [0...n]$) denotes the state space name corresponding to the depth level i, and q_{n+1} represents all depth levels greater or equal to $n+1$. For formulating the XPath containment, the guide is computed from the two XPath expressions. Specifically, the deepest (and thus the most precise) guide is chosen as the guide for both expressions.

Eventually, each variable is restricted with a list of state spaces that represents the regions in the tree where its valuation must be searched. For instance, "`ex1 [l1] x2`" means the scope of the variable `x2` is limited to tree nodes occurring at depth level 1.

This optimization is useful for any kind of XPath expressions: absolute or relative. More precise restrictions can be computed for absolute XPath expressions (for which the initial set of depth levels is the singleton $\{0\}$).

5.3 Practical Experiments

The objective of this section aims at testing the practical performance of our method. To this end, we carried out several testing scenarios of our implementation. First, we used an XPath benchmark [21] whose goal is to cover XPath features by gathering a significant variety of XPath expressions met in real-world applications. This first test series consists in 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 the others. Comparing two queries Q_i and Q_j may yield to 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 this by $Q_i \subset Q_j$ or alternatively by $Q_j \supset Q_i$
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 on Figure 18, and results together with total running times of the decision procedure are summarized on Figure 19. Several tests are expensive in the case of the FTA-based decision procedure. "N/A" denotes that the procedure did not complete

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)]

Figure 18: XPath Queries Taken from the XPathmark Benchmark.

within reasonable time and space bounds for comparing Q_4 and Q_2. The optimized decision procedure gives better and comparable results for both $\subseteq$ and $\supseteq$ tests. Obtained results show that all tests are solved in less than 0.5 seconds. This reflects the fact that XPath expressions used in real-world scenarios tend not to be very complex.

The second test series consists in comparing expressions taken from research papers on the containment of XPath expressions. Some have been used to test proposed techniques (such as the tree pattern homomorphisms [32]). They have also been used to show that checking XPath containment in general may become very hard. Figure 20 presents the expressions we collected and shows associated results.

Finally, Figure 21 presents the results of a third test series including examples with intersection, and axes such as "following" and "preceding", not present in the previous series.

These experiments have been conducted on a Pentium 4, 3 Ghz, with 1Gb of RAM, running Linux. Our implementation has been developed in JAVA and controls the C++ implementation of the MONA solver.

6 Related Work

Extensive research has been conducted on XPath query containment. Different fragments of the XPath language have been studied. Among them, a core XPath fragment is frequently used. This fragment isolates the "child" axis noted "/" (and included in all fragments), the "descendant" axis (often noted "//" in the literature[4]), branching "[ ]", and wildcard "*" as the most important features, and is denoted by $\mathrm{XP}^{\{*,//,[]\}}$. Decidability of containment for $\mathrm{XP}^{\{*,//,[]\}}$ can be obtained by a translation to datalog with recursion. While containment is undecidable for general datalog with recursion, it has been shown using chase techniques, that the datalog fragment needed for $\mathrm{XP}^{\{*,//,[]\}}$ has a decidable containment problem [43]. More specifically, containment for $\mathrm{XP}^{\{*,//,[]\}}$ is coNP-complete [32]. The containment mapping technique relies on a polynomial time tree homomorphism algorithm, which gives a sufficient but not necessary condition for containment of $\mathrm{XP}^{\{*,//,[]\}}$ in general.

[4] Actually $p_1//p_2$ stands for p_1/descendant-or-self::node()/p_2 in the XPath standard formal semantics [16]

Relation	FTA Time (s)		GTA Time (s)	
	$\subseteq$	$\supseteq$	$\subseteq$	$\supseteq$
$Q_1 \not\subset Q_2$	0.12	19.19	0.28	0.28
$Q_1 \not\subset Q_3$	0.07	0.06	0.10	0.10
$Q_1 \not\subset Q_4$	0.52	4.76	0.11	0.11
$Q_1 \supset Q_5$	0.07	0.06	0.08	0.08
$Q_1 \not\subset Q_6$	0.10	0.07	0.13	0.12
$Q_1 \not\subset Q_7$	0.10	0.07	0.12	0.13
$Q_1 \supset Q_8$	0.07	0.07	0.08	0.09
$Q_1 \not\subset Q_9$	0.08	0.11	0.11	0.11
$Q_2 \subset Q_3$	0.10	7.23	0.30	0.30
$Q_2 \subset Q_4$	0.11	N/A	0.31	0.31
$Q_2 \not\subset Q_5$	20.04	0.13	0.29	0.29
$Q_2 \not\subset Q_6$	15.25	2.33	0.35	0.35
$Q_2 \not\subset Q_7$	19.93	10.49	0.37	0.36
$Q_2 \not\subset Q_8$	19.79	0.12	0.29	0.29
$Q_2 \not\subset Q_9$	19.76	0.17	0.30	0.31
$Q_3 \supset Q_4$	0.04	0.04	0.04	0.04
$Q_3 \not\subset Q_5$	0.06	0.07	0.11	0.11
$Q_3 \not\subset Q_6$	0.05	0.05	0.06	0.07
$Q_3 \not\subset Q_7$	0.05	0.05	0.06	0.07
$Q_3 \not\subset Q_8$	0.08	0.08	0.11	0.11
$Q_3 \not\subset Q_9$	0.08	0.11	0.13	0.12
$Q_4 \not\subset Q_5$	7.13	0.63	0.11	0.12
$Q_4 \not\subset Q_6$	0.05	0.05	0.07	0.07
$Q_4 \not\subset Q_7$	0.05	0.05	0.07	0.07
$Q_4 \not\subset Q_8$	7.26	0.64	0.12	0.11
$Q_4 \not\subset Q_9$	6.45	0.76	0.13	0.13
$Q_5 \not\subset Q_6$	0.11	0.07	0.13	0.12
$Q_5 \not\subset Q_7$	0.12	0.08	0.13	0.14
$Q_5 \equiv Q_8$	0.06	0.07	0.07	0.09
$Q_5 \not\subset Q_9$	0.08	0.12	0.12	0.11
$Q_6 \not\subset Q_7$	0.05	0.05	0.07	0.06
$Q_6 \not\subset Q_8$	0.08	0.12	0.13	0.13
$Q_6 \not\subset Q_9$	0.09	0.24	0.14	0.15
$Q_7 \not\subset Q_8$	0.08	0.13	0.13	0.13
$Q_7 \not\subset Q_9$	0.09	0.24	0.15	0.15
$Q_8 \not\subset Q_9$	0.13	0.09	0.14	0.14

Figure 19: Results and Total Running Times of the Decision Procedure.

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/e][b/f][c]
E_7 a[b/e][b/f]

E_8 /descendant::editor[parent::journal]
E_9 /descendant-or-self::journal/child::editor

Relation	FTA Time (s)		GTA Time (s)	
	$\subseteq$	$\supseteq$	$\subseteq$	$\supseteq$
$E_1 \subset E_2$	1.49	1.56	0.79	0.71
$E_3 \supset E_4$	1.64	0.07	0.83	0.22
$E_3 \supset E_5$	1.67	0.08	0.76	0.24
$E_4 \supset E_5$	0.31	0.08	0.36	0.24
$E_6 \subset E_7$	0.07	0.17	0.15	0.22
$E_8 \equiv E_9$	0.04	0.04	0.04	0.04

Figure 20: Results on XPath Containment Examples Found in Research Papers.

E_{10} a/b//c/following-sibling::d/e
E_{11} a//d[preceding-sibling::c]/e
E_{12} //a//b//c/following-sibling::d/e
E_{13} //b[ancestor::a]//*[preceding-sibling::c]/e
E_{14} /b[preceding::a]//following::c
E_{15} /a/b//following::c
E_{16} a/b[//c]/following::d/e
E_{17} a//d[preceding::c]/e
E_{18} a/b//d[preceding-sibling::c]/e
E_{19} a/c/following::d/e
E_{20} a/d[preceding::c]/e
E_{21} a/b[//c]/following::d/e $\cap$ a/d[preceding::c]/e
E_{22} a/c/following::d/e $\cap$ a/d[preceding::c]/e

Relation	FTA Time (s)		GTA Time (s)	
	$\subseteq$	$\supseteq$	$\subseteq$	$\supseteq$
$E_{10} \subset E_{11}$	0.14	0.18	0.21	0.12
$E_{12} \subset E_{13}$	0.11	0.39	0.27	0.31
$E_{14} \subset E_{15}$	0.08	0.33	0.12	0.13
$E_{16} \subset E_{17}$	0.14	0.19	0.21	0.28
$E_{18} \equiv E_{10}$	0.09	0.09	0.18	0.19
$E_{19} \not\sim E_{20}$	0.22	0.76	0.31	0.48
$E_{21} \subset E_{19}$	0.15	0.17	0.22	0.23
$E_{22} \not\sim E_{16}$	0.09	0.24	0.18	0.21

Figure 21: Results on Examples Including "following" and "preceding" Axes.

If any of the three constructs "*", "//", or "[ ]" is dropped then query containment is in PTIME [32]. In particular, containment for $XP^{\{//,[\,]\}}$ is shown to be in PTIME in [2], and [43] noted that containment for $XP^{\{//,*\}}$ is also in PTIME.

Authors of [34] show that containment for $XP^{\{*,//,[\,],|\}}$, while coNP-complete for an infinite alphabet, is in PSPACE for a finite alphabet. They also show that containment for $XP^{\{//,|\}}$ is complete for PSPACE.

A summary of complexity results for various XPath fragments, classified with respect to complexity classes can be found in [35].

Characterizations of the expressive power of these language in terms of both logics and tree patterns are given in [5]. This work also studies structural properties such as closure properties focusing on the ability to perform basic boolean operations while remaining in the same fragment.

A different but nevertheless related problem concerns XPath containment in presence of constraints. [14] considers XPath containment in the presence of DTDs and simple XPath integrity constraints (SXICS). They obtain that this problem is undecidable in general and in the presence of bounded SXICs and DTDs. Additionally, the containment problem is shown to be in EXPTIME for the fragments $XP^{\{//,[\,]\}}$, $XP^{\{//,[\,],|\}}$, $XP^{\{//,|\}}$ in the presence of DTDs [44].

From a theoretical perspective, the connection between XPath and formal logics is actively studied [30, 4, 3]. In particular, [30] characterizes a subset of XPath in terms of extensions of Computational Tree Logic (CTL) [12], which is equivalent to first order logic (FO) over tree structures [29, 3] and whose satisfiability is in EXPTIME. Authors of [32] first observed that a fragment of XPath can be embedded in CTL. However, regular tree languages are not fully captured by such FO variants [6]. The work found in [1] proposes a variant of Propositional Dynamic Logic (PDL) [20] with a similar EXPTIME complexity for reasoning about ordered trees, but whose exact expressive power is still under study.

Compared to all these previous works, the XPath fragment we consider is far more complete and much more realistic. If the connection between XPath and MSO has already been mentioned [4, 3], it has not been developed nor exploited yet. Note that our implementation is, in our knowledge, the only known fully implemented working system for deciding the containment between two realistic XPath expressions. WS2S has a high worst-case complexity, which indicates probable blow-ups for large and complex WS2S formulas. This does not preclude a useful and practical decision procedure for XPath containment.

7 Conclusion

In this paper, we proposed a new logical approach for the XPath containment problem. XPath queries are translated in a decidable logic called WS2S. XPath containment is formulated in terms of a WS2S formula, then decided using tree automata. This paper makes several contributions.

First, we propose a specific variant of MSO, namely WS2S, as a logic for modeling XML instances and XPath queries. The automaton-logic connection has not been fully

investigated in the context of XML. We believe that this work is a significant step in this direction which has not revealed its full potential yet. As a valuable outcome, we show how an XPath expression can be translated into an equivalent formula in monadic second-order logic.

We propose a sound and complete decision procedure for XPath containment. It consists in building the full automaton corresponding to the XPath containment problem. An additional benefit of this technique is to allow generation of tree examples and counter-examples of the truth status of the formula. We believe this makes our method of special interest for many applications including debuggers, or enhancing reporting during static analysis stages.

We show how containment can be effectively decided for a large XPath fragment that includes union, intersection, path composition and boolean connectives together with all XPath axes, branching, and wildcards. This fragment is far more complete than other fragments addressed in previous studies.

Eventually, we propose an optimization method based on guided tree automata that takes advantage of XPath peculiarities to speed up the decision procedure. The global proposed approach has been implemented. Although the worst-case complexity of WS2S is non-elementary, we provide practical experiments and detailed results that corroborate our claim that this decision procedure is efficient for real-world XPath expressions.

One direction of future work is to search for tree automata guides that produce a finer-grained partition of the automaton state space, in order to enhance the scalability of the decision procedure. Another direction consists in studying the containment problem under regular type constraints such as DTDs or XML-Schemas. This requires to express type constraints in WS2S and combine them efficiently with the containment formula.

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

[3] 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. IEEE Computer Society.

[4] 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.

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

[6] M. Benedikt and L. Segoufin. Regular tree languages definable in FO. In *STACS '05: Proceedings of the 22nd Annual Symposium on Theoretical Aspects of Computer Science*, volume 3404 of *LNCS*, pages 327–339, London, UK, 2005. Springer-Verlag.

[7] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0, W3C working draft, September 2005. http://www.w3.org/TR/2005/WD-xpath20-20050915/.

[8] M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for guided tree automata. In *WIA '96: Revised Papers from the First International Workshop on Implementing Automata*, volume 1260 of *LNCS*, pages 6–25, London, UK, 1997. Springer-Verlag.

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

[10] J. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[11] 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.

[12] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, volume 131 of *LNCS*, pages 52–71, London, UK, 1981. Springer-Verlag.

[13] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 1997. release October, 1st 2002.

[14] A. Deutsch and V. Tannen. Containment of regular path expressions under integrity constraints. In *KRDB '01: Proceedings of the 8th International Workshop on Knowledge Representation meets Databases*, volume 45 of *CEUR Workshop Proceedings*, pages 1–11, ceur-ws.org, 2001. CEUR.

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

[16] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics, W3C working draft, September 2005. http://www.w3.org/TR/2005/WD-xquery-semantics-20050915/.

[17] J. Elgaard, A. Møller, and M. I. Schwartzbach. Compile-time debugging of C programs working on trees. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, volume 1782 of *LNCS*, pages 119–134, London, UK, 2000. Springer-Verlag.

[18] C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98:21–52, 1961.

[19] 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.

[20] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[21] M. Franceschet. XPathMark - an XPath benchmark for XMark generated data. In *XSYM '05: Proceedings of The Third International Symposium on Database and XML Technologies*, volume 3671 of *LNCS*, pages 129–143, London, UK, 2005. Springer-Verlag.

[22] P. Genevès and J.-Y. Vion-Dury. XPath formal semantics and beyond: A Coq-based approach. In *TPHOLs '04: Emerging Trends Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*, pages 181–198, Salt Lake City, Utah, United States, August 2004. University Of Utah.

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

[24] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for xml. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2005.

[25] G. Huet, G. Kahn, and C. Paulin-Mohring. *The* Coq *Proof Assistant - A tutorial - Version 8.0*. INRIA, April 2004.

[26] N. Klarlund. Mona & Fido: The logic-automaton connection in practice. In *CSL '97: Selected Papers from the 11th International Workshop on Computer Science Logic*, volume 1414 of *LNCS*, pages 311–326, London, UK, 1998. Springer-Verlag.

[27] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[28] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *CIAA '00: Revised Papers from the 5th International Conference on Implementation and Application of Automata*, volume 2088 of *LNCS*, pages 182–194, London, UK, 2001. Springer-Verlag.

[29] M. Marx. Conditional XPath, the first order complete XPath dialect. In *PODS '04: Proceedings of the twenty-third ACM Symposium on Principles of Database Systems*, pages 13–22, New York, NY, USA, 2004. ACM Press.

[30] M. Marx. XPath with conditional axis relations. In *Proceedings of the 9th International Conference on Extending Database Technology*, volume 2992 of *LNCS*, pages 477–494, London, UK, January 2004. Springer-Verlag.

[31] A. Meyer. Weak monadic second-order theory of successor is not elementary-recursive. In R. Parikh, editor, *Proceedings of Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154, New York, NY, USA, 1975. Springer-Verlag.

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

[33] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[34] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, volume 2572 of *LNCS*, pages 315–329, London, UK, 2003. Springer-Verlag.

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

[36] L. Stockmeyer. The complexity of decision problems in automata theory and logic. Technical Report MAC-TR-133, Project MAC, M.I.T, Cambridge, Massachusetts, United States, 1974.

[37] L. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *STOC '73: Proceedings of the 5th ACM symposium on Theory of computing*, pages 1–9, New York, NY, USA, 1973. ACM Press.

[38] G. Sur, J. Hammer, and J. Siméon. Updatex - an XQuery-based language for processing updates in XML. In *PLAN-X 2004: Proceedings of the International Workshop on Programming Language Technologies for XML, Venice, Italy*, volume NS-03-4 of *BRICS Notes Series*, pages 40–53, Aarhus, Denmark, January 2004. BRICS.

[39] Y. Tanabe, K. Takahashi, M. Yamamoto, A. Tozawa, and M. Hagiya. A decision procedure for the alternation-free two-way modal μ-calculus. In *TABLEAUX '05: Proceedings of the 14th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, volume 3702 of *LNCS*, pages 277–291, London, UK, September 2005. Springer-Verlag.

[40] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.

[41] A. Tozawa. Towards static type checking for XSLT. In *DocEng '01: Proceedings of the 2001 ACM Symposium on Document Engineering*, pages 18–27, New York, NY, USA, 2001. ACM Press.

[42] P. Wadler. Two semantics for XPath. Internal Technical Note of the W3C XSL Working Group, http://homepages.inf.ed.ac.uk/wadler/papers/xpath-semantics/xpath-semantics.pdf, January 2000.

[43] P. T. Wood. On the equivalence of XML patterns. In *CL '00: Proceedings of the First International Conference on Computational Logic*, volume 1861 of *LNCS*, pages 1152–1166, London, UK, 2000. Springer-Verlag.

[44] P. T. Wood. Containment for XPath fragments under DTD constraints. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, volume 2572 of *LNCS*, pages 300–314, London, UK, August 2003. Springer-Verlag.

Contents

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399