Università degli Studi “G.D’Annunzio”
Dipartimento di Scienze

Modal logic and navigational XPath: an experimental comparison

Massimo Franceschet Enrico Zimuel

September 14, 2005

Technical Report no. R-2005-001 *Research Series*

Modal logic and navigational XPath: an experimental comparison

Massimo Franceschet Enrico Zimuel

Dipartimento di Scienze, Università "G. D'Annunzio"
Viale Pindaro, 42 – 65127 Pescara (Italy)
`m.franceschet@unich.it, enrico@zimuel.it`

September 14, 2005

Abstract. XPath is the core retrieval language of XQuery, the official query language for XML data. We empirically compare three query evaluation strategies for the navigational fragment of XPath known as Core XPath: a bottom-up algorithm based on model checking techniques for multi-modal logic, a first top-down procedure based on a technique to eliminate XPath filters, and a second top-down procedure that takes advantage of the pre/post plane representation of an XML tree. We implement the three methods and we benchmark the resulting XPath processors using a fragment of XPathMark, a recently proposed benchmark for XPath.

Keywords: *XPath query evaluation, model checking, benchmarking.*

Contents

1 Introduction

The *Extensible Markup Language* (XML) [14] is a popular representation language for semistructured data [1], which are data that do not necessarily possess a regular schema. The XML Path Language (XPath) [15] is a simple retrieval language for data represented in XML. In particular, XPath in the core retrieval fragment of the XML Query Language (XQuery) [16], the standard query language for XML.

XPath and modal logic are similar in many respects. Syntactically, the XPath language contains navigational axes that closely resemble modal logic *modalities*. Semantically, XPath queries are evaluated on XML trees, which are tree-shaped *Kripke structures* whose states (nodes) are labelled with XML tags. Finally, the query evaluation problem for XPath can be reinterpreted as a *model checking problem* for multi-modal logic.

XPath queries have the form $q[\alpha]$, where q is called *path* and α is called *filter*. A path is a sequence of axis steps and it is interpreted according to the following *query semantics*: it retrieves those nodes that are reachable from the current one through the axes used in q. A filter is similar to a *modal logic formula* and it is interpreted according to the standard *modal logic semantics*: it selects the current node if it satisfies the filter α. These two semantics are orthogonal, and they are mixed in the semantics of XPath. This orthogonality is the cause of the exponential complexity of a naive implementation of the semantics of XPath [7]. There exist two main strategies to avoid this exponential behaviour. The first translates the path q into a modal logic formula α_q and then applies modal logic semantics and methods. The second reduces the filter α to a query q_α and then uses query semantics and techniques. However, it is not clear which of the two contexts, either the modal logic context or the database one, is more appropriate for the implementation of efficient evaluation algorithms for XPath.

In this paper, we neatly isolate two evaluation strategies for the navigational fragment of XPath known as Core XPath [7]. The first algorithm, that we called BottomXPath, first translates a Core XPath query into a modal logic formula and then applies model checking procedures in order to retrieve the answer set of the original query. This algorithm works *bottom-up* with respect to the parse tree of the input query: it elaborates the sub-queries of the input query from the leaves of the parse tree up to the tree root. The second algorithm, that we called TopXPath, first replaces the filters present in the input query with query paths and then applies a node retrieval procedure in order to compute the answer set of the original query. This procedure works *top-down* with respect to the parse tree of the input query: it elaborates the sub-queries of the input query from the root of the parse tree down to the leaves of the tree. The XQuery formal semantics requires that the result of an XPath expression is a sequence of document nodes that is document sorted and duplicate free. The *document order* corresponds to the total order of nodes given by a preorder visit of the nodes of an XML tree. With reference to the time when the sorting of the XPath expression results is performed, we specify two versions of the top-down algorithm. The first version, that we named TopXPath1, does not care about the order of the nodes in the intermediate node sequences and it document sorts the final result only. The second version, that we named TopXPath2, maintains document sorted all the intermediate node sequences and hence it does not need to sort the final result. More importantly, it takes advantage of the hypothesis that the intermediate results are document sorted in order to speed-up the XPath axis evaluation. This happens by pruning the intermediate results as much as possible before starting each step evaluation.

A theoretical analysis of the worst-case asymptotic computational complexity of the outlined algorithms does not help in evaluating their real-life performance: all the procedures run in asymptotic worst-case *linear time* with respect to the product of the size of the XML tree and the length of the query. In order to better understand the computational differences between the proposed strategies, which is our main goal in this paper, we performed an *experimental analysis*. We implemented the algorithms in standard C language and we used a fragment of the XPath benchmark XPathMark [5] to assess the empirical complexity of the discussed strategies.

The rest of the paper is as follows. We survey related work in Section 2. In Section 3 we introduce XPath and relates it to modal logic. In Section 4 we describe an exponential-time algorithm that strictly follows the semantics of XPath, while in Sections 5 and 6 we describe the bottom-up and top-down evaluation strategies, respectively. In Section 7 we perform the experimental analysis of the proposed algorithms and we sum-up in Section 8.

2 Related work

As noticed by Gottlob et al. [6], many commercial engines implement XPath processing by adopting a naive exponential-time strategy even though the query processing problem for XPath admits a polynomial-time algorithm. Gottlob et al. [7] propose a bottom-up polynomial-time XPath processing algorithm for full XPath, which runs in linear time for Core XPath. Moreover, they discuss a general mechanism for translating the bottom-up algorithm into a top-down one. The relation between XPath query evaluation and model checking has been investigated in [2, 10], where the authors embed a fragment of Core XPath into temporal logic and use an existing model checker to solve the query evaluation problem. The idea of maintaining document sorted the intermediate answers in order to speed-up the axis evaluation has been proposed in [11], a work that is mostly inspired by the results in [8, 9]. However, none of these paper has empirically compared the different strategies for XPath query evaluation. This is our main task in this work. Our bottom-up procedure BottomXPath borrows from ideas in [2], while our first top-down algorithm

TopXPath1 has been inspired by the work in [6]. Finally, our second top-down algorithm TopXPath2 is an simplified version of the procedure proposed in [11].

3 XPath and modal logic

In this section we introduce XPath and relates it to modal logic.

3.1 XML path languages

Here we describe the syntax and the semantics of the navigational fragment of XPath that was called Core XPath in [6]. Moreover, we define an extension of Core XPath, namely Boolean XPath, that allows more freedom in the use of Boolean operators in the composition of queries.

Let Σ be a set of labels including the special one denoted by `*`. Let χ be the set of Core XPath axes, namely:

$$\begin{aligned}\chi \;=\; &\{\texttt{self}, \texttt{child}, \texttt{parent}, \texttt{descendant}, \texttt{ancestor}, \texttt{descendant-or-self},\\ &\texttt{ancestor-or-self}, \texttt{following-sibling}, \texttt{preceding-sibling},\\ &\texttt{following}, \texttt{preceding}\}\end{aligned}$$

We say that `child` is the inverse of `parent` and viceversa, `descendent` is the inverse of `ancestor` and viceversa, `descendant_or_self` is the inverse of `ancestor_or_self` and viceversa. Moreover, `following_sibling` is the inverse of `preceding_sibling` and viceversa, `following` is the inverse of `preceding` and viceversa, and finally `self` is the inverse of itself.

A *Core XPath query* is defined by the query clause of the following grammar:

$$\begin{array}{lcl}
\texttt{query} & = & /\texttt{path} \\
\texttt{path} & = & \texttt{step} \mid \texttt{step}/\texttt{path} \\
\texttt{step} & = & \texttt{axis} :: \texttt{a} \mid \texttt{axis} :: \texttt{a}[\texttt{filter}] \\
\texttt{filter} & = & \texttt{path} \mid \texttt{filter and filter} \mid \texttt{filter or filter} \mid \texttt{not}(\texttt{filter}) \mid (\texttt{filter}) \\
\texttt{axis} & \in & \chi \\
\texttt{a} & \in & \Sigma
\end{array}$$

The Boolean XPath language extends the Core XPath language with Boolean operators at path level. More precisely, a *Boolean XPath query* is defined by the query clause of the following grammar:

$$\begin{array}{lcl}
\texttt{query} & = & \texttt{path} \mid /\texttt{path} \mid \texttt{query and query} \mid \texttt{query or query} \mid \texttt{not}(\texttt{query}) \mid (\texttt{query}) \\
\texttt{path} & = & \texttt{step} \mid \texttt{path}/\texttt{path} \mid \texttt{path and path} \mid \texttt{path or path} \mid \texttt{not}(\texttt{path}) \mid (\texttt{path}) \\
\texttt{step} & = & \texttt{axis} :: \texttt{a} \mid \texttt{axis} :: \texttt{a}[\texttt{path}] \\
\texttt{axis} & \in & \chi \\
\texttt{a} & \in & \Sigma
\end{array}$$

Notice that each Core XPath query is a Boolean XPath query, but not viceversa. For instance, the Boolean XPath query $/\texttt{child} :: \texttt{a}/(\texttt{child} :: \texttt{b}\ \texttt{or}\ \texttt{child} :: \texttt{c})$ is not a Core XPath query. For a query q, we define its length, denoted by $length(q)$, as the sum of the number if Boolean operators and the number of *atomic* steps appearing in q. An atomic step has the form `axis::a`.

Our target in this paper is Core XPath, which is the core fragment of the standard XPath [15]. We will use Boolean XPath, which is *not* a fragment of the official XPath language, as an auxiliary language only. In particular, we will use Boolean XPath as an embedding language for the query filters in Section 6. However, it is worth noticing that all the algorithms and results in this paper easily extend to Boolean XPath language.

Both Core and Boolean XPath languages are interpreted over XML trees representing XML documents. Since in the present work we are only interested in the navigational power of XPath, we assume that the XML documents we work with do not contain attributes, namespaces, processing instructions, comments, and parsed character data. An XML tree is a rooted sibling-ordered tree $T = (N, R_{\downarrow}, R_{\rightarrow}, L)$, where:

- N is a set of nodes. We denote by *root* the root node of the tree. A tree node represents an element in the XML document;

- $R_{\downarrow}$ is a binary relation on N such that $(x, y) \in R_{\downarrow}$ iff y is a child of x;

- $R_{\rightarrow}$ is a (functional) binary relation on N such that $(x, y) \in R_{\rightarrow}$ iff y is the right sibling of x;

- L is a function from Σ to the power set of N such that, for $a \in \Sigma \setminus \{*\}$, $L(a)$ is the set of nodes that are labelled with tag a, and $L(*) = N$.

Given an XML tree T, a query q in the Boolean XPath language, and a context set $C \subseteq N$, the semantics of the Boolean XPath language (and hence of the Core XPath language as well) is given by a function $\sigma(T, q, C)$ returning a subset of N. The semantic function σ is inductively defined as follows:

$$
\begin{array}{lcl}
\sigma(T, \texttt{axis} :: \texttt{a}, C) & = & \{y \in N \mid \exists x \in C.\ (x,y) \in R^T_{\texttt{axis}} \wedge y \in L(\texttt{a})\} \\
\sigma(T, \texttt{axis} :: \texttt{a}[\texttt{path}], C) & = & \{y \in N \mid y \in \sigma(T, \texttt{axis} :: \texttt{a}, C) \wedge \sigma(T, \texttt{path}, \{y\}) \neq \emptyset\} \\
 & & \\
\sigma(T, \texttt{path}_1/\texttt{path}_2, C) & = & \sigma(T, \texttt{path}_2, \sigma(T, \texttt{path}_1, C)) \\
\sigma(T, \texttt{path}_1 \texttt{ and } \texttt{path}_2, C) & = & \sigma(T, \texttt{path}_1, C) \cap \sigma(T, \texttt{path}_2, C) \\
\sigma(T, \texttt{path}_1 \texttt{ or } \texttt{path}_2, C) & = & \sigma(T, \texttt{path}_1, C) \cup \sigma(T, \texttt{path}_2, C) \\
\sigma(T, \texttt{not}(\texttt{path}), C) & = & N \setminus \sigma(T, \texttt{path}, C) \\
 & & \\
\sigma(T, /\texttt{path}, C) & = & \sigma(T, \texttt{path}, \{root\}) \\
\sigma(T, \texttt{query}_1 \texttt{ and } \texttt{query}_2, C) & = & \sigma(T, \texttt{query}_1, C) \cap \sigma(T, \texttt{query}_2, C) \\
\sigma(T, \texttt{query}_1 \texttt{ or } \texttt{query}_2, C) & = & \sigma(T, \texttt{query}_1, C) \cup \sigma(T, \texttt{query}_2, C) \\
\sigma(T, \texttt{not}(\texttt{query}), C) & = & N \setminus \sigma(T, \texttt{query}, C)
\end{array}
$$

The relation $R^T_{\texttt{axis}}$ is a binary relation on N corresponding to the specified axis. Given a binary relation R, let R^+ be its transitive closure, R^* be its reflexive and transitive closure, and R^{-1} be its inverse. Moreover, $R_1 \circ R_2$ denotes the concatenation of R_1 and R_2. The relation $R_{\texttt{axis}}$ is formally defined as follows:

$$
\begin{array}{lcl}
R^T_{\texttt{self}} & = & \{(x,x) \mid x \in N\} \\
 & & \\
R^T_{\texttt{child}} & = & R_{\downarrow} \\
R^T_{\texttt{parent}} & = & (R^T_{\texttt{child}})^{-1} \\
 & & \\
R^T_{\texttt{descendant}} & = & (R_{\downarrow})^+ \\
R^T_{\texttt{ancestor}} & = & (R^T_{\texttt{descendant}})^{-1} \\
 & & \\
R^T_{\texttt{descendant-or-self}} & = & (R_{\downarrow})^* \\
R^T_{\texttt{ancestor-or-self}} & = & (R^T_{\texttt{descendant-or-self}})^{-1} \\
 & & \\
R^T_{\texttt{following-sibling}} & = & (R_{\rightarrow})^+ \\
R^T_{\texttt{preceding-sibling}} & = & (R^T_{\texttt{following-sibling}})^{-1} \\
 & & \\
R^T_{\texttt{following}} & = & R^T_{\texttt{ancestor-or-self}} \circ R^T_{\texttt{following-sibling}} \circ R^T_{\texttt{descendant-or-self}} \\
R^T_{\texttt{preceding}} & = & (R^T_{\texttt{following}})^{-1}
\end{array}
$$

Finally, the *answer set* of the query q with respect to the tree T is equal to $\sigma(T, q, N)$.

3.2 The connection between XPath and modal logic

Modal logic [3] extends propositional logic with modalities that, similarly to XPath axis, are used to browse the underlying relational structure. Let Σ be a set of proposition symbols. A *formula* in the (multi-) modal language is defined as follows:

$$\alpha = p \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \neg\alpha \mid \langle R_i \rangle \alpha$$

where $p \in \Sigma$ and $1 \leq i \leq c$ for some integer $c \geq 1$. We define $[R_i] = \neg\langle R_i \rangle \neg$. For a modal formula α, we define its length, denote by $length(\alpha)$, as the sum of the number of (Boolean and modal) operators plus the number of proposition symbols appearing in α. Moreover, let $sub(\alpha)$ as the set of subformulas of α. Notice that $|sub(\alpha)| = length(\alpha)$.

Modal formulas are interpreted over models. A model for modal logic is a triple $(W, \mathcal{R}, V)$, with $\mathcal{R} = \{R_1, \ldots, R_c\}$, where:

- W is a set elements which are called *states*;
- each $R_i \subseteq W \times W$ is a binary relation on W;
- V is a function from Σ to the power set of W such that, for $p \in \Sigma$, $L(p)$ is the set of states that are labelled with the proposition symbol p.

Given a modal formula α, a model $M = (W, \mathcal{R}, V)$, and a state $x \in W$, the semantics of modal logic is defined as follows:

$$\begin{array}{lll}
M, x \models p & \text{iff} & x \in V(p) \\
M, x \models \alpha \wedge \beta & \text{iff} & M, x \models \alpha \text{ and } M, x \models \beta \\
M, x \models \alpha \vee \beta & \text{iff} & M, x \models \alpha \text{ or } M, x \models \beta \\
M, x \models \neg\alpha & \text{iff} & M, x \not\models \alpha \\
M, x \models \langle R_i \rangle \alpha & \text{iff} & \text{there is } y \text{ such that } (x, y) \in R_i \text{ and } M, y \models \alpha
\end{array}$$

The *truth set* of a formula α with respect to a model M is the set $\{x \in W \mid M, x \models \alpha\}$.

The intimate relation between XPath and modal logic is explicated in the definition of Core XPath logic. *Core XPath logic* is an instance of multi-modal logic in which there is one modality for each axis in XPath. It is defined over a set of labels Σ including the special symbols denoted by `*` and `root`. A model for Core XPath logic is a relational structure corresponding to an XML tree. More precisely, given an XML tree $T = (N, R_{\downarrow}, R_{\rightarrow}, L)$, the corresponding model for Core XPath logic is $M_T = (N, \{R^T_{\mathtt{axis}}\}_{\mathtt{axis} \in \mathcal{X}}, L)$, where $L(\mathtt{root})$ is a singleton containing the root node of T. In Section 5 we will show how to embed Core XPath queries into

Core XPath formulas. Notice that $\langle\texttt{following}\rangle$ and $\langle\texttt{preceding}\rangle$ modalities are in fact redundant. Indeed:

$$\langle\texttt{following}\rangle\alpha \equiv \langle\texttt{ancestor-or-self}\rangle\langle\texttt{following-sibling}\rangle\langle\texttt{descendant-or-self}\rangle\alpha$$
$$\langle\texttt{preceding}\rangle\alpha \equiv \langle\texttt{ancestor-or-self}\rangle\langle\texttt{preceding-sibling}\rangle\langle\texttt{descendant-or-self}\rangle\alpha$$

We will use these equivalences in the following algorithms.

4 An exponential evaluation strategy

In this section we give a first implementation, called ShallowXPath, of a Core XPath query processor. The algorithm strictly follows the semantics of Core XPath given in Section 3.1, and, as we will show later, its complexity might be *exponential* in the length of the query. The procedure ShallowXPath inputs an XML tree T, a Core XPath query `q`, and a context set C. The tree T is represented as follows: each node is an object composed of a field *pre* containing the order of the node in a preorder visit of the tree, a field p containing a pointer to the parent of the node, or NIL if the node is the root, a field c containing a pointer to the first child of the node, or NIL if the node is a leaf, a field r containing a pointer to the right sibling of the node, or NIL if the node is the last sibling, a field l containing a pointer to the left sibling of the node, or NIL if the node is the first sibling, and a field *tag* containing to tag of the XML element that the node represents. The procedure ShallowXPath uses a sub-procedure ProcessStep in order to elaborate a single axis step. The latter calls an auxiliary procedure Descendant that retrieves all the descendant nodes of a given node that are labelled with a given tag. Moreover, ProcessStep invokes ProcessFilter whenever a filter must be evaluated. The latter recursively calls ShallowXPath. The pseudo-code is as follows.

ShallowXPath(T, `q`, C)
$R \leftarrow \emptyset$
`step` $\leftarrow$ *head*(`q`)
while `step` $\neq$ NIL **do**
  for all $x \in C$ **do**
    $R \leftarrow R \cup$ ProcessStep(T, `step`, x)
  end for
  $C \leftarrow R$
  `step` $\leftarrow$ *next*(`q`)
end while
return C

ProcessStep(T, `step`, x)
let `step` = `axis` :: `a`[`filter`]

```
R ← ∅
case
• axis = self
if a = * or tag[x] = a then
   R ← R ∪ {x}
end if
• axis = child
y ← c[x]
while y ≠ NIL do
   if a = * or tag[y] = a then
      R ← R ∪ {y}
   end if
   y ← r[y]
end while
• axis = parent
y ← p[x]
if y ≠ NIL then
   if a = * or tag[y] = a then
      R ← R ∪ {y}
   end if
end if
• axis = descendant
y ← c[x]
while y ≠ NIL do
   R ← R ∪ Descendants(y, a)
   y ← r[y]
end while
• axis = ancestor
y ← p[x]
while y ≠ NIL do
   if a = * or tag[y] = a then
      R ← R ∪ {y}
   end if
   y ← p[y]
end while
• axis = descendant-or-self
R ← Descendants(x, a)
• axis = ancestor-or-self
y ← x
while y ≠ NIL do
   if a = * or tag[y] = a then
```

```
      R ← R ∪ {y}
    end if
    y ← p[y]
end while
• axis = following-sibling
y ← r[x]
while y ≠ NIL do
    if a = ∗ or tag[y] = a then
        R ← R ∪ {y}
    end if
    y ← r[y]
end while
• axis = preceding-sibling
y ← l[x]
while y ≠ NIL do
    if a = ∗ or tag[y] = a then
        R ← R ∪ {y}
    end if
    y ← l[y]
end while
• axis = following
q ← ancestor-or-self::*/following-sibling::*/descendant-or-self::a
R ← ShallowXPath(T, q, {x})
• axis = preceding
q ← ancestor-or-self::*/preceding-sibling::*/descendant-or-self::a
R ← ShallowXPath(T, q, {x})
endcase
if filter ≠ NIL then
    for all x ∈ R do
        if not ProcessFilter(T, filter, x) then
            R ← R \ {x}
        end if
    end for
end if
return R
```

```
ProcessFilter(T, filter, x)
case
• filter = filter_1 and filter_2
return ProcessFilter(T, filter_1, x) and ProcessFilter(T, filter_2, x)
• filter = filter_1 or filter_2
```

return ProcessFilter(T, $\texttt{filter}_1$, x) **or** ProcessFilter(T, $\texttt{filter}_2$, x)
• $\texttt{filter} = \texttt{not filter}_1$
return **not** ProcessFilter(T, $\texttt{filter}_1$, x)
• $\texttt{filter} = \texttt{path}$
if ShallowXPath(T, $\texttt{path}$, $\{x\}$)) $\neq \emptyset$ **then**
  return TRUE
else
  return FALSE
end if
endcase

Descendants(x, $\texttt{a}$)
$R \leftarrow \emptyset$
$Q \leftarrow \emptyset$
if $x \neq$ NIL **then**
  $Enqueue(Q, x)$
end if
while $Q \neq \emptyset$ **do**
  $y \leftarrow Dequeue(Q)$
  if $\texttt{a} = *$ **or** $tag[y] = \texttt{a}$ **then**
    $R \leftarrow R \cup \{y\}$
  end if
  $y \leftarrow c[y]$
  while $y \neq$ NIL **do**
    $Enqueue(Q, y)$
    $y \leftarrow r[y]$
  end while
end while
return R

We claim that the complexity of ShallowXPath is exponential in the nesting degree of filter expressions in the query. Let $C(n, k, r)$ be the complexity of ShallowXPath on a tree of n nodes and a query of length k and of filter nesting degree r. We will show that $C(n, k, r) = O(k \cdot n^{2 \cdot r+2})$.

Let $k_1 = O(k)$ be the number of steps in the query which are not in a filter expression, and let $k_2 = O(k)$ be the maximum length of any filter in the query. For $r = 0$ (no filters are present in the query), we have that $C(n, k, r) = O(k \cdot n^2)$. For $r > 0$, we have that $C(n, k, r) = k_1 \cdot n \cdot f(n, k_2, r)$, where $f(n, k, r)$ is the complexity of ProcessStep on a tree of n nodes, a step of length k and of filter nesting degree r. Moreover, $f(n, k, r) = n + n \cdot g(n, k, r - 1)$, where $g(n, k, r)$ is the cost of ProcessFilter on a tree of n nodes, a filter of length k and of filter nesting degree r. Finally, $g(n,k,r) = k + C(n,k,r)$. The worst-case is a query of the form:

$$/\texttt{axis} :: \texttt{a}_1[\texttt{axis} : \texttt{a}_2[\texttt{axis} : \texttt{a}_3 \ldots [\texttt{axis} : \texttt{a}_\texttt{r}]] \ldots]$$

In such a case, $k_1 = 1$ and $k_2 = O(k)$. Thus:

$$\begin{array}{rcl} C(n,k,r) & = & n \cdot f(n,k,r) \\ & = & n \cdot (n + n \cdot g(n,k,r-1)) \\ & = & n \cdot (n + n \cdot (k + C(n,k,r-1))) \\ & = & O(n^2 \cdot k) + n^2 \cdot C(n,k,r-1) \\ & = & O(n^4 \cdot k) + n^4 \cdot C(n,k,r-2) \\ & = & \ldots \\ & = & O(n^{2 \cdot r} \cdot k) + n^{2 \cdot r} \cdot C(n,k,0) \\ & = & O(n^{2 \cdot r} \cdot k) + n^{2 \cdot r + 2} \cdot k \\ & = & O(k \cdot n^{2 \cdot r + 2}) \end{array}$$

Hence, the complexity of ShallowXPath is polynomial whenever the query has a bounded nesting degree of filters. However, if this degree is not bounded, then Shallow is very inefficient. In the following Sections 5 and 6 we will show how to avoid this exponential behaviour.

5 A bottom-up evaluation strategy

In this section we give an efficient bottom-up algorithm, called BottomXPath, to evaluate a Core XPath query. The algorithm is based on a technique that in the logic context is known as model checking [4]. The *model checking problem* is the following question: given a model M and a formula α, retrieve the truth set of α with respect to M. A *model checker* is an algorithm that solves the model checking problem.

We start by embedding Core XPath queries into Core XPath formulas. We first define a translation ω from XPath filter expressions into Core XPath formulas. A *filter expression* in XPath is defined by the filter clause of the Core XPath grammar given in Section 3.1. The function ω is as follows (if `filter` is empty in the first two clauses below, then the corresponding conjunct is missing):

$$\begin{array}{rcl} \omega(\texttt{axis} :: \texttt{a}[\texttt{filter}]) & = & \langle\texttt{axis}\rangle(\texttt{a} \wedge \omega(\texttt{filter})) \\ \omega(\texttt{axis} :: \texttt{a}[\texttt{filter}]/\texttt{path}) & = & \langle\texttt{axis}\rangle(\texttt{a} \wedge \omega(\texttt{filter}) \wedge \omega(\texttt{path})) \\ \omega(\texttt{filter}_1 \texttt{ and } \texttt{filter}_2) & = & \omega(\texttt{filter}_1) \wedge \omega(\texttt{filter}_2) \\ \omega(\texttt{filter}_1 \texttt{ or } \texttt{filter}_2) & = & \omega(\texttt{filter}_1) \vee \omega(\texttt{filter}_2) \\ \omega(\texttt{not}(\texttt{filter})) & = & \neg\omega(\texttt{filter}) \end{array}$$

We now define a translation τ from Core XPath queries into Core XPath formulas as follows (if `filter` is empty in the below clauses, then the corresponding conjunct is missing):

$$\begin{aligned}
\tau(/\texttt{axis} :: \texttt{a}[\texttt{filter}]) &= \texttt{a} \wedge \omega(\texttt{filter}) \wedge \langle \texttt{axis}^{-1} \rangle\ \texttt{root} \\
\tau(\texttt{path}/\texttt{axis} :: \texttt{a}[\texttt{filter}]) &= \texttt{a} \wedge \omega(\texttt{filter}) \wedge \langle \texttt{axis}^{-1} \rangle\ \tau(\texttt{path})
\end{aligned}$$

where $\texttt{axis}^{-1}$ is the inverse of `axis`. Notice that the length of $\tau(q)$ is *linear* in the length of q. We have the following:

Theorem 5.1 *Let q be a Core XPath query and T be an XML tree. Then, the answer set of q with respect to T is the truth set of $\tau(q)$ with respect to M_T.*

By virtue of Theorem 5.1, the answer set for a Core XPath query equals to the truth set for the corresponding Core XPath formula. Hence, we can solve the query evaluation problem in terms of the model checking problem by using a model checker as a query processor. The algorithm is as follows. Let T be an XML tree and q be a Core XPath query:

- build the model M_T corresponding to the tree T;
- translate q into a modal formula $\tau(q)$;
- compute the truth set of $\tau(q)$ with respect to M_T using a model checker for modal logic.

The complexity of the outlined method is the following. First, notice that, for any axis different from `self`, `child`, and `parent`, the cardinality of the relation $R^T_{\texttt{axis}}$ belonging to M_T might be quadratic in the number n of nodes of the XML tree. Hence, computing the model M_T costs $O(n^2)$. Translating the query q costs $O(k)$, where k is the length of q. The size of $\tau(q) = O(k)$. Model checking for modal logic costs is $O(k \cdot (n + m))$, where k is the length of the formula, n is the number of states of the model, and m is the biggest cardinality of any reachability relation in the model. Since, in our case, $m = O(n^2)$, the overall complexity of the above algorithm is $O(k \cdot n^2)$, hence quadratic in the number of nodes of the XML tree.

In the following, we give an alternative model checking algorithm for Core XPath logic that runs in time $O(k \cdot n)$. BottomXPath is a bottom-up model checker for Core XPath logic. It inputs an XML tree T (and not a multi-modal model) and a Core XPath formula α. The algorithm is similar to a model checker for the temporal logic CTL (see, e.g., [4]); instead of CTL temporal operators, BottomXPath checks XPath axes. BottomXPath uses a subprocedure EvalAxis. The latter inputs a tree T, and axis `axis` and a formula β. For each node $x \in N$, the procedure EvalAxis labels x

with $\langle \texttt{axis} \rangle \beta$ if, and only if, there exists a node $y \in N$ reachable from x trough the relation induced by `axis` such that y is labelled with β. EvalAxis takes advantage of a Boolean matrix A, where rows represent formulas and columns represent nodes, in order to label nodes with formulas that are true at them. Moreover, it uses the auxiliary procedure LabelDescendants in order to label the descendant nodes of a given node with a given formula. In the following code we assume that $\langle \texttt{following} \rangle$ and $\langle \texttt{preceding} \rangle$ modalities in α has been replaced as shown in Section 3.2.

```
BottomXPath(T, α)
for all β ∈ sub(α) do
   for all x ∈ N do
      A(β, x) ← 0
   end for
end for
for all i ← 1 to length(α) do
   for all β ∈ sub(α) such that length(β) = i do
      case
      • β = root
      A(β, root) ← 1
      • β = ∗
      for all x ∈ N do
         A(β, x) ← 1
      end for
      • β ∈ Σ \ {root, ∗}
      for all x ∈ L(β) do
         A(β, x) ← 1
      end for
      • β = β1 ∧ β2
      for all x ∈ N do
         if (A(β1, x) = 1  and  A(β2, x) = 1) then
            A(β, x) ← 1
         end if
      end for
      • β = β1 ∨ β2
      for all x ∈ N do
         if (A(β1, x) = 1  or  A(β2, x) = 1) then
            A(β, x) ← 1
         end if
      end for
      • β = ¬β1
      for all x ∈ N do
```

if $A(\beta_1, x) = 0$ **then**
$A(\beta, x) \leftarrow 1$
end if
end for
• $\beta = \langle \texttt{axis} \rangle \beta_1$
EvalAxis$(T, \texttt{axis}, \beta_1)$
endcase
end for
end for
$R \leftarrow \emptyset$
for all $x \in N$ **do**
if $A(\alpha, x) = 1$ **then**
$R \leftarrow R \cup \{x\}$
end if
end for
return R

EvalAxis$(T, \texttt{axis}, \beta)$
case
• $\texttt{axis} = \texttt{self}$
for all $x \in N$ **do**
if $A(\beta, x) = 1$ **then**
$A(\langle \texttt{self} \rangle \beta, x) \leftarrow 1$
end if
end for
• $\texttt{axis} = \texttt{child}$
for all $x \in N$ **do**
$y \leftarrow c[x]$
$found \leftarrow$ FALSE
while $y \neq$ NIL **and not** $found$ **do**
if $A(\beta, y) = 1$ **then**
$A(\langle \texttt{child} \rangle \beta, x) \leftarrow 1$
$found \leftarrow$ TRUE
end if
$y \leftarrow r[y]$
end while
end for
• $\texttt{axis} = \texttt{parent}$
for all $x \in N$ **do**
$y \leftarrow p[x]$
if $y \neq$ NIL **and** $A(\beta, y) = 1$ **then**

```
        A(⟨parent⟩β, x) ← 1
    end if
end for
• axis = descendant
for all x ∈ N do
    if A(β, x) = 1 then
        y ← p[x]
        while y ≠ NIL and A(⟨descendant⟩β, y) = 0 do
            A(⟨descendant⟩β, y) ← 1
            y ← p[y]
        end while
    end if
end for
• axis = ancestor
for all x ∈ N do
    if A(β, x) = 1 and A(⟨ancestor⟩β, x) = 0 then
        y ← c[x]
        while y ≠ NIL do
            LabelDescendant(⟨ancestor⟩β, y)
            y ← r[y]
        end while
    end if
end for
• axis = descendant-or-self
for all x ∈ N do
    if A(β, x) = 1 then
        y ← x
        while y ≠ NIL and A(⟨descendant⟩β, y) = 0 do
            A(⟨descendant⟩β, y) ← 1
            y ← p[y]
        end while
    end if
end for
• axis = ancestor-or-self
for all x ∈ N do
    if A(β, x) = 1 and A(⟨ancestor⟩β, x) = 0 then
        LabelDescendant(⟨ancestor⟩β, x)
    end if
end for
• axis = following-sibling
for all x ∈ N do
```

```
  if A(β, x) = 1 then
    y ← l[x]
    while y ≠ NIL and A(⟨ following-sibling ⟩β, y) = 0 do
      A(⟨ following-sibling ⟩β, y) ← 1
      y ← l[y]
    end while
  end if
end for
• axis = preceding-sibling
for all x ∈ N do
  if A(β, x) = 1 then
    y ← r[x]
    while y ≠ NIL and A(⟨ preceding-sibling ⟩β, y) = 0 do
      A(⟨ preceding-sibling ⟩β, y) ← 1
      y ← r[y]
    end while
  end if
end for
endcase
```

```
LabelDescendants(α, x)
Q ← ∅
if x ≠ NIL and A(α, x) = 0 then
  Enqueue(Q, x)
end if
while Q ≠ ∅ do
  y ← Dequeue(Q)
  A(α, y) ← 1
  y ← c[y]
  while y ≠ NIL do
    if A(α, y) = 0 then
      Enqueue(Q, y)
    end if
    y ← r[y]
  end while
end while
```

The computational complexity of EvalAxis is *linear* in the number of nodes of the tree T. The cost of BottomXPath is hence $O(k \cdot n)$, thus linear is the product of the length of the query and the size of the XML tree.

The whole bottom-up evaluation algorithm for Core XPath is as follows:

1. translate q into $\tau(q)$ and replace the modalities $\langle \texttt{following} \rangle$ and $\langle \texttt{preceding} \rangle$ appearing in $\tau(q)$ obtaining a formula α_q;

2. run BottomXPath on T and α_q;

3. sort, in document order, the result of BottomXPath.

The complexity of the translation step is $O(k)$ and the call to BottomXPath costs $O(k \cdot n)$. Since nodes are integers from 1 to n, we can use a linear-time sorting algorithm like CountingSort to sort the result. Hence, the overall complexity for the bottom-up evaluation of q on T is $O(k \cdot n)$.

6 A top-down evaluation strategy

In this section we give two efficient top-down algorithms, called TopXPath1 and TopXPath2, to evaluate Core XPath queries. Both the algorithms first replace the filters present in the input query with query paths and then apply a node retrieval procedure in order to compute the answer set of the original query.

We first show how to get rid of filters. The *inverting translation* ι inputs a filter expression in the Core XPath language and returns its inverse in the Boolean XPath language. It is defined as follows:

$$\begin{array}{lcl}
\iota(\texttt{axis} :: \texttt{a}) & = & \texttt{self} :: \texttt{a}/\texttt{axis}^{-1} :: * \\
\iota(\texttt{axis} :: \texttt{a}[\texttt{filter}]) & = & \iota(\texttt{filter})/\iota(\texttt{axis} :: \texttt{a}) \\
\iota(\texttt{step}/\texttt{path}) & = & \iota(\texttt{path})/\iota(\texttt{step}) \\
\iota(\texttt{filter}_1 \texttt{ and } \texttt{filter}_2) & = & \iota(\texttt{filter}_1) \texttt{ and } \iota(\texttt{filter}_2) \\
\iota(\texttt{filter}_1 \texttt{ or } \texttt{filter}_2) & = & \iota(\texttt{filter}_1) \texttt{ or } \iota(\texttt{filter}_2) \\
\iota(\texttt{not}(\texttt{filter})) & = & \texttt{not}(\iota(\texttt{filter}))
\end{array}$$

Notice that $\iota(q)$ is a query without filters in the Boolean XPath language. We now define a translation υ from Core XPath queries into Boolean XPath queries *without filters*:

$$\begin{array}{lcl}
\upsilon(/\texttt{axis} :: \texttt{a}) & = & /\texttt{axis} :: \texttt{a} \\
\upsilon(/\texttt{axis} :: \texttt{a}[\texttt{filter}]) & = & (/\texttt{axis} :: \texttt{a} \texttt{ and } \iota(\texttt{filter})) \\
\upsilon(\texttt{step}/\texttt{path}) & = & \upsilon(\texttt{step})/\upsilon(\texttt{path})
\end{array}$$

Notice that length of $\upsilon(q)$ is linear in the length of q. We have the following:

Theorem 6.1 *Let T be an XML tree and q be a Core XPath query. Then, q and $\upsilon(q)$ have the same answer set.*

6.1 A first top-down algorithm

In this section we propose a first top-down strategy, called TopXPath1, to evaluate Core XPath queries. With respect to the data structure described in Section 4, we assume here that an additional field called *count* is added to the object representation of each node of the tree. The new field is used to record whether the node has been visited or not during a step evaluation. TopXPath1 does not care about the order of the nodes in the intermediate context sets and it sorts the final result only. TopXPath1 inputs an XML tree T, a Boolean XPath query without filters q, and a context set C. It uses a sub-procedure ProcessPath1 to elaborate query paths, which in turn calls a procedure ProcessStep1 to evaluate query steps. In particular, the procedure ProcessStep1$(T, \texttt{axis}, \texttt{a}, C)$ elaborates the step $\texttt{axis} :: \texttt{a}$ on the tree T with context set C, according to the XPath semantics. In order to avoid to walk on the same node twice, the procedure checks the *count* field of the node's object. This filed is initialized to 0 for each node when TopXPath1 starts. The global variable k is also initialized to 0 and it is incremented by one at each step evaluation performed with ProcessStep1. When a node is visited during a step evaluation, its *count* field is assigned to the value contained in k. Hence, during the k-th step evaluation, all nodes that has been already visited in that step evaluation have their count field set to k, while the count field of the unexplored nodes is less than k. This method avoids the costly resetting of the count field at each step evaluation. Finally, ProcessStep1 uses an auxiliary procedure RetrieveDescendants to retrieve the descendant nodes of a given node that are labelled with a given tag. The pseudo-code is as follows.

TopXPath1$(T, \texttt{q}, C)$
$k \leftarrow 0$
for $x \in N$ **do**
  $count[x] \leftarrow 0$
end for
case
• $\texttt{q} = \texttt{query}_1 \texttt{ and } \texttt{query}_2$
return TopXPath1$(T, \texttt{query}_1, C) \cap$ TopXPath1$(T, \texttt{query}_2, C)$
• $\texttt{q} = \texttt{query}_1 \texttt{ or } \texttt{query}_2$
return TopXPath1$(T, \texttt{query}_1, C) \cup$ TopXPath1$(T, \texttt{query}_2, C)$
• $\texttt{q} = \texttt{not(query)}$
return $N \setminus$ TopXPath1$(T, \texttt{query}, C)$
• $\texttt{q} = \texttt{/path}$
return ProcessPath1$(T, \texttt{path}, \{root(T)\})$
• $\texttt{q} = \texttt{path}$
return ProcessPath1$(T, \texttt{path}, N)$
endcase

```
ProcessPath1(T, p, C)
case
• p = path₁ and path₂
return ProcessPath1(T, path₁, C) ∩ ProcessPath1(T, path₂, C)
• p = path₁ or path₂
return ProcessPath1(T, path₁, C) ∪ ProcessPath1(T, path₂, C)
• p = not(path)
return N \ ProcessPath1(T, path, C)
• p = step/path
return ProcessPath1(T, path, ProcessPath1(T, step, C))
• p = axis :: a
return ProcessStep1(T, axis, a, C)
endcase
```

```
ProcessStep1(T, axis, a, C)
k ← k + 1
R ← ∅
case
• axis = self
for x ∈ C do
   if a = ∗ or tag[x] = a then
      R ← R ∪ {x}
   end if
end for
• axis = child
for x ∈ C do
   y ← c[x]
   while y ≠ NIL do
      if a = ∗ or tag[y] = a then
         R ← R ∪ {y}
      end if
      y ← r[y]
   end while
end for
• axis = parent
for x ∈ C do
   y ← p[x]
   if y ≠ NIL and count[y] < k then
      count[y] ← k
      if a = ∗ or tag[y] = a then
         R ← R ∪ {y}
```

end if
end if
end for
• `axis` = `descendant`
for $x \in C$ **do**
if $count[x] < k$ **then**
$count[x] \leftarrow k$
$y \leftarrow c[x]$
while $y \neq \textsc{nil}$ **do**
$R \leftarrow R \cup \text{RetrieveDescendants}(y, \texttt{a})$
$y \leftarrow r[y]$
end while
end if
end for
• `axis` = `ancestor`
for $x \in C$ **do**
$y \leftarrow p[x]$
while $y \neq \textsc{nil}$ **and** $count[y] < k$ **do**
$count[y] \leftarrow k$
if $\texttt{a} = *$ **or** $tag[y] = \texttt{a}$ **then**
$R \leftarrow R \cup \{y\}$
end if
$y \leftarrow p[y]$
end while
end for
• `axis` = `descendant-or-self`
for $x \in C$ **do**
if $count[x] < k$ **then**
$R \leftarrow R \cup \text{RetrieveDescendants}(x, a)$
end if
end for
• `axis` = `ancestor-or-self`
for $x \in C$ **do**
$y \leftarrow x$
while $y \neq \textsc{nil}$ **and** $count[y] < k$ **do**
$count[y] \leftarrow k$
if $\texttt{a} = *$ **or** $tag[y] = \texttt{a}$ **then**
$R \leftarrow R \cup \{y\}$
end if
$y \leftarrow p[y]$
end while

```
end for
• axis = following-sibling
for x ∈ C do
   y ← r[x]
   while y ≠ NIL and count[y] < k do
      count[y] ← k
      if a = ∗ or tag[y] = a then
         R ← R ∪ {y}
      end if
      y ← r[y]
   end while
end for
• axis = preceding-sibling
for x ∈ C do
   y ← l[x]
   while y ≠ NIL and count[y] < k do
      count[y] ← k
      if a = ∗ or tag[y] = a then
         R ← R ∪ {y}
      end if
      y ← l[y]
   end while
end for
• axis = following
q ← ancestor-or-self::*/following-sibling::*/descendant-or-self::a
R ← ProcessPath1(T, q, C)
• axis = preceding
q ← ancestor-or-self::*/preceding-sibling::*/descendant-or-self::a
R ← ProcessPath1(T, q, C)
endcase
return R
```

```
RetrieveDescendants(x, a)
R ← ∅
Q ← ∅
if x ≠ NIL and count[x] < k then
   count[x] ← k
   Enqueue(Q, x)
end if
while Q ≠ ∅ do
   y ← Dequeue(Q)
```

```
  if a = * or tag[y] = a then
    R ← R ∪ {y}
  end if
  y ← c[y]
  while y ≠ NIL do
    if count[y] < k then
      count[y] ← k
      Enqueue(Q, y)
    end if
    y ← r[y]
  end while
end while
return R
```

In the worst-case, the procedure ProcessStep1 visits the entire tree and hence its cost is $O(n)$, where n is the number of nodes of the tree. The evaluation algorithm TopXPath1 runs in linear time with respect to the product of the size of the XML tree and the length of the query. The whole top-down evaluation algorithm for Core XPath is as follows:

1. translate q into $\upsilon(q)$;
2. run TopXPath1 on T and $\upsilon(q)$;
3. sort, in document order, the result of TopXPath1.

If k is the length of q and n is the number of nodes of T, the complexity of the translation step is $O(k)$ and the call to TopXPath1 costs $O(k \cdot n)$. Since nodes are integers from 1 to n, we can use a linear-time sorting algorithm like CountingSort to sort the result. Hence, the overall complexity for the evaluation of q on T with the top-down method described in this section is $O(k \cdot n)$, as for the bottom-up strategy proposed in Section 5.

6.2 A second top-down algorithm

In this section we propose a second top-down strategy, called TopXPath2, to evaluate Core XPath queries. With respect to the data structure described in Section 4, we assume that two additional fields are added to the object representation of each node of the tree: a field called *count* that, as in TopXPath1, is used to record whether the node has been visited or not during a step evaluation, and a field called *post* containing the order of the node in a postorder visit of the tree. TopXPath2 uses a sub-procedure ProcessPath2 which in turns calls ProcessStep2, as done for TopXPath1. ProcessStep2 uses an auxiliary procedure Children to retrieve the children nodes of a given node that are labelled with a given tag, and Descendants to retrieve the descendant nodes of a given node that are labelled with a given tag. Moreover, it uses the following auxiliary list procedures, where C and L are double-linked lists and x is a node:

- NewList(), that initializes a new list;
- DelFirst(C), that deletes and returns the first element of C;
- DelLast(C), that deletes and returns the last element of C;
- AddAfter(C, x), that appends x to C;
- AddListAfter(C, L), that appends L to C;
- AddBefore(C, x), that adds x in front of C;
- AddListBefore(C, L), that adds L in front of C;
- First(C) that returns the first element of C;
- Last(C) that returns the last element of C.

All these procedures can be implemented in constant time. TopXPath2 differs from TopXPath1 since it maintains document sorted the intermediate context sets. Moreover, it exploits the sorted contexts to speed-up the XPath axis evaluation by pruning the context sets as much as possible before starting each step evaluation. By maintaining both the preorder and the postorder ranks for each node, TopXPath2 implicitly represents an XML tree as a bi-dimensional plane, called the *pre/post plane* in [8]. Each node x is encoded by the point $(pre(x), post(x))$. A nice feature of this encoding is that, for each node x, the top-right (respectively, bottom-left) quadrant of x contains all the following (respectively, preceding) nodes of x, and the bottom-right (respectively, top-left) quadrant of x contains all the descendant (respectively, ancestor) nodes of x. Hence, given two arbitrary nodes x and y, we can check in constant time the relative position of y with respect to x. As an example, consider the cases of `following` and `preceding` axes. By exploiting the pre/post plane properties, the context set can always be reduced to a singleton (see code lines 157–160 and 173). Finally, TopXpath2 takes advantage, when necessary, of the counting technique described in Section 6.1 to avoid the exploration of the same tree zones twice. The pseudo-code of ProcessStep2 is as follows.

1: ProcessStep2(T, $\texttt{axis}, \texttt{a}, C$)
2: $k \leftarrow k + 1$
3: $R \leftarrow \emptyset$

```
case
• axis = self
L ← NewList()
while C ≠ ∅ do
   x ← DelFirst(C)
   if a = ∗ or tag[x] = a then
      AddAfter(L, x)
   end if
end while
return L
• axis = child
L ← NewList()
S ← NewList()
while C ≠ ∅ do
   x ← First(C)
   if S = ∅ then
      AddListBefore(S, Children(x, a))
      DelFirst(C)
   else
      if pre[First(S)] ≤ pre[x] then
         AddAfter(L, DelFirst(S))
      else
         AddListBefore(S, Children(x, a))
         DelFirst(C)
      end if
   end if
end while
if S ≠ ∅ then
   AddListAfter(L, S)
end if
return L
• axis = parent
L ← NewList()
while C ≠ ∅ do
   x ← DelFirst(C)
   y ← p[x]
   if count[y] < k then
      if a = ∗ or tag[y] = a then
         AddAfter(L, y)
      end if
      count[y] ← k
```

```
    end if
end while
return L
• axis = descendant
L ← NewList()
while C ≠ ∅ do
    x ← DelFirst(C)
    while C ≠ ∅ and post[First(C)] < post[x] do
        DelFirst(C)
    end while
    y ← c[x]
    while y ≠ NIL do
        AddListAfter(L, Descendants(y, a))
        y ← r[y]
    end while
end while
return L
• axis = ancestor
L ← NewList()
while C ≠ ∅ do
    x ← DelFirst(C)
    S ← NewList()
    y ← p[x]
    while y ≠ NIL and count[y] < k do
        if a = * or tag[y] = a then
            AddBefore(S, y)
        end if
        count[y] ← k
        y ← p[y]
    end while
    AddListAfter(L, S)
end while
return L
• axis = descendant-or-self
L ← NewList()
while C ≠ ∅ do
    x ← DelFirst(C)
    while C ≠ ∅ and post[First(C)] < post[x] do
        DelFirst(C)
    end while
    AddListAfter(L, Descendants(x, a))
```

```
end while
return L
• axis = ancestor-or-self
L ← NewList()
while C ≠ ∅ do
   x ← DelFirst(C)
   S ← NewList()
   y ← x
   while y ≠ NIL and count[y] < k do
      if a = ∗ or tag[y] = a then
         AddBefore(S, y)
      end if
      count[y] ← k
      y ← p[y]
   end while
   AddListAfter(L, S)
end while
return L
• axis = following-sibling
L ← NewList()
H ← NewList()
while C ≠ ∅ do
   S ← NewList()
   x ← DelFirst(C)
   y ← r[x]
   while y ≠ NIL and count[y] < k do
      if a = ∗ or tag[y] = a then
         if C ≠ ∅ and post[First(C)] < post[y] then
            AddAfter(S, y)
         else
            while H ≠ ∅ and pre[First(H)] < pre[y] do
               AddAfter(L, DelFirst(H))
            end while
            AddAfter(L, y)
         end if
      end if
      count[y] ← k
      y ← r[y]
   end while
   AddListBefore(H, S)
end while
```

```
AddListAfter(L, H)
return L
• axis = preceding-sibling
L ← NewList()
H ← NewList()
while C ≠ ∅ do
   S ← NewList()
   x ← DelLast(C)
   y ← l[x]
   while y ≠ NIL and count[y] < k do
      if a = * or tag[y] = a then
         if C ≠ ∅ and pre[Last(C)] > pre[y] then
            AddBefore(S, y)
         else
            while H ≠ ∅ and pre[Last(H)] > pre[y] do
               AddBefore(L, DelLast(H))
            end while
            AddBefore(L, y)
         end if
      end if
      count[y] ← k
      y ← l[y]
   end while
   AddListAfter(H, S)
end while
AddListBefore(L, H)
return L
• axis = following
L ← NewList()
if C ≠ ∅ then
   x ← DelFirst(C)
   while C ≠ ∅ and post[First(C)] < post[x] do
      x ← DelFirst(C)
   end while
   while x ≠ NIL do
      y ← r[x]
      while y ≠ NIL do
         AddListAfter(L, Descendants(y, a))
         y ← r[y]
      end while
      x ← p[x]
```

end while
end if
return L
• $\texttt{axis} = \texttt{preceding}$
$L \leftarrow \text{NewList}()$
$x \leftarrow \text{Last}(C)$
while $x \neq \text{NIL}$ **do**
$M \leftarrow \text{NewList}()$
$y \leftarrow l[x]$
while $y \neq \text{NIL}$ **do**
$\text{AddListAfter}(M, \text{Descendants}(y, \texttt{a}))$
$y \leftarrow l[y]$
end while
$\text{AddListBefore}(L, M)$
$x \leftarrow p[x]$
end while
return L
endcase

$\text{Children}(x, \texttt{a})$
$L \leftarrow \text{NewList}()$
$y \leftarrow c[x]$
while $y \neq \text{NIL}$ **do**
if $\texttt{a} = *$ **or** $tag[y] = \texttt{a}$ **then**
$\text{AddAfter}(L, y)$
end if
$y \leftarrow r[y]$
end while
return L

$\text{Descendants}(x, \texttt{a})$
$L \leftarrow \text{NewList}()$
$S \leftarrow \text{NewList}()$
while $x \neq \text{NIL}$ **do**
if $\texttt{a} = *$ **or** $tag[x] = \texttt{a}$ **then**
$\text{AddAfter}(L, x)$
end if
$x \leftarrow c[x]$
$\text{AddBefore}(S, x)$
end while
while $S \neq \emptyset$ **do**

$x \leftarrow r[\mathrm{DelFirst}(S)]$
while $x \neq$ NIL **do**
  if $\mathtt{a} = *$ **or** $tag[x] = \mathtt{a}$ **then**
    $\mathrm{AddAfter}(L, x)$
  end if
  $\mathrm{AddBefore}(S, x)$
  $x \leftarrow c[x]$
end while
end while
return L

In the worst-case, the procedure ProcessStep2 visits the entire tree and hence its cost is $O(n)$, where n is the number of nodes of the tree. The evaluation algorithm TopXPath2 runs in linear time with respect to the product of the size of the XML tree and the length of the query. The whole top-down evaluation algorithm for Core XPath is as follows:

1. translate q into $\upsilon(q)$;
2. run TopXPath2 on T and $\upsilon(q)$.

If k is the length of q and n is the number of nodes of T, the complexity of the translation step is $O(k)$ and the call to TopXPath2 costs $O(k \cdot n)$. Since TopXPath2 maintains sorted the context sets, the result of TopXPath2 is already sorted. Hence, the overall complexity for the evaluation of q on T with the top-down method described in this section is $O(k \cdot n)$, as for BottomXPath and TopXPath1.

7 Experimental analysis

All the three algorithms proposed in Sections 5 and 6, namely BottomXPath, TopXPath1 and TopXPath2, have the same asymptotic worst-case complexity. In order to better understand the computational differences between the proposed strategies, we performed an *experimental analysis*. We implemented the algorithms in standard C language and we used a fragment of the XPath benchmark XPathMark [5] to assess the empirical complexity of the discussed strategies. In this section, we report about this analysis. The source code (released under the GNU General Public License), the executable programs (for Gnu/Linux systems), and additional experimental data and plots (including a comparison with XQuery processor Saxon [12]) are available at `http://www.zimuel.it/xpath`.

Our experiments were run on an AMD Sempron 1.7 GHz, with 1 GB RAM, running Debian Gnu/Linux version 2.6.10. All the times are response CPU times in seconds. We ran tests using a variety of XML documents and XPath queries.

The documents are generated using the XML benchmarking program XMark [13]. XMark generated documents are modeled after a database as deployed by an Internet auction site, a typical e-commerce application. They allow for the formulation of queries that both feel natural and present concise challenges. Moreover, the generated documents make the behavior of queries predictable. XMark provides an accurate scaling of the XML document size using a user defined scaling factor f. The numbers are calibrated to match a total XML document size of approximately 100 MB when f assume the value 1.0. We used the following scaling factors:

$$(0.001, 0.002, 0.004, 0.008, 0.016, 0.032, 0.064, 0.128, 0.256, 0.512, 1)$$

corresponding to the following sizes (in MB):

$$(0.116, 0.212, 0.468, 0.909, 1.891, 3.751, 7.303, 15.044, 29.887, 59.489, 116.517)$$

As for the benchmark queries, we used a navigational fragment of XPathMark [5]. XPathMark is a benchmark for XPath consisting of a set of queries covering all aspects of XPath 1.0. These queries have been designed for XML documents generated under XMark. The benchmark queries we used in this paper are the following:

Q1 *The keywords in annotations of closed auctions*

```
/child::site/child::closed_auctions/child::closed_auction
/child::annotation/child::description/child::parlist
/child::listitem/child::text/child::keyword
```

Q2 *All the keywords*

```
/descendant::keyword
```

Q3 *The keywords in a paragraph item*

```
/descendant-or-self::listitem/descendant-or-self::keyword
```

Q4 *The (either North or South) American items*

```
/child::site/child::regions/child::*/child::item
[parent::namerica or parent::samerica]
```

Q5 *The paragraph items containing a keyword*

```
/descendant::keyword/ancestor::listitem
```

Q6 *The mail containing a keyword*

```
/descendant::keyword/ancestor-or-self::mail
```

Q7 *The last bidder of all open auctions*

```
/child::site/child::open_auctions/child::open_auction
```

```
/child::bidder[not(following-sibling::bidder)]
```

Q8 *The first bidder of all open auctions*

```
/child::site/child::open_auctions/child::open_auction
/child::bidder[not(preceding-sibling::bidder)]
```

Q9 *The last item of the document*

```
/child::site/child::regions/child::*/child::item[not(following::item)]
```

Q10 *The first item of the document*

```
/child::site/child::regions/child::*/child::item[not(preceding::item)]
```

Q11 *People having an address and either a phone or a homepage*

```
/child::site/child::people/child::person
[child::address and (child::phone or child::homepage)]
```

Q12 *People having no homepage*

```
/child::site/child::people/child::person[not(child::homepage)]
```

We are interested into the evaluation of the efficiency and of the data scalability of the three algorithms proposed in this paper. To perform this evaluation, we took advantage of the following standard measures:

- Given a query q and a document d, the *query response time* is the time taken by an algorithm to give the answer for the query q on the document d including all the phases of the elaboration (parsing of the document, processing the query, serialization of the results, etc).
- Given a query q and a document d, the *query response speed* is defined as the size of the document d divided by the response time for query q and document d. The measure unit is, for instance, MB/sec.
- Given a query q and two documents d_1 and d_2, where the size of d_2 is bigger than the size of d_1, the *data scalability factor* is defined as v_1/v_2, where v_1 is the query response speed of q on d_1 and v_2 is the query response speed of q on d_2.

The response time for a query gives an indication of how fast is a query processor to give the answer, while the data scalability factor is useful to test how a query processor performs when the size of the XML data increases. In particular, if the scalability factor is lower than 1, that is $v_1 < v_2$, then we have a positive speed acceleration when moving from document d_1 to document d_2. In this case, we say that the scalability is *sub-linear*. If the scalability factor is higher then 1, that is $v_1 > v_2$, then we have a negative speed acceleration when moving from document d_1

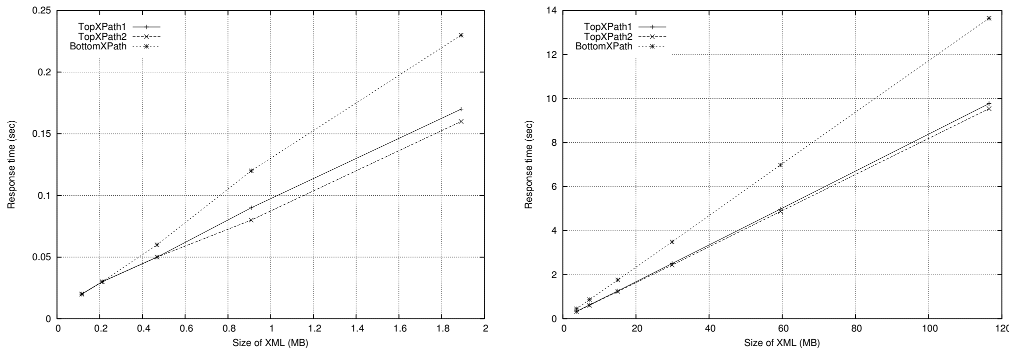

Figure 1. *Benchmark response times*

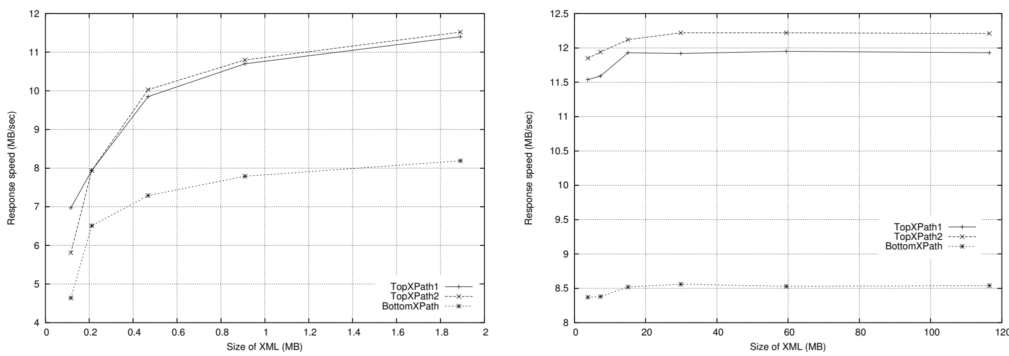

Figure 2. *Benchmark response speeds*

to document d_2. In this case, we say that the scalability is *super-linear*. Finally, if the scalability factor is equal to 1, that is $v_1 = v_2$, then the speed is constant when moving from document d_1 to document d_2. In this case, we say that scalability is *linear*. Usually, sub-linear and linear scalability are good properties of a query processor.

These measures can be aggregated along two directions, the document and the query one. Let us fix a document d and vary the query in the benchmark set. One can compute the average of the response times of all the benchmark queries on document d. This measure, that we call the *benchmark response time* for d, is depicted in Figure 1, where we vary the size of the document on the x-axis (the left side plot is for documents from scaling factor 0.001 up to 0.016, and the right side plot is for bigger documents from scaling factor 0.032 up to 1). The *benchmark response speed* for d is the size of d divided by the benchmark response time for d. This is illustrated in Figure 2. The *benchmark data scalability factor* is defined as above in terms of the benchmark response speed. This is shown in Figure 3.

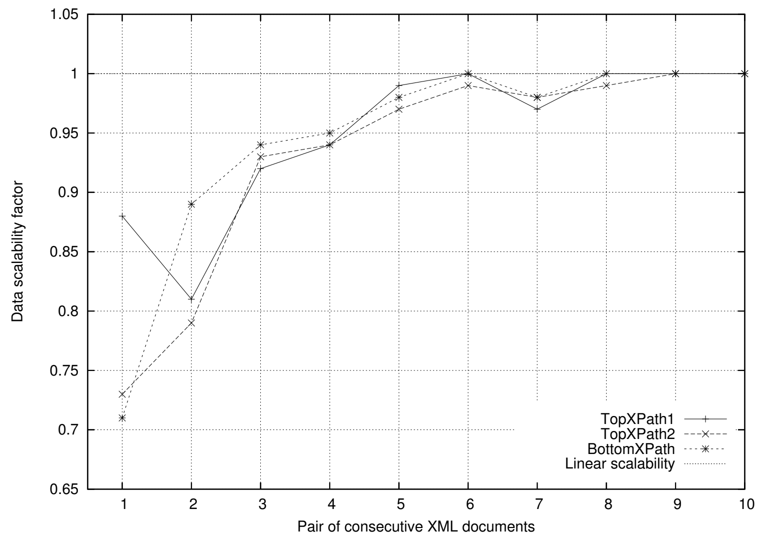

Figure 3. *Benchmark data scalability factors*

Moreover, let us now fix a query q and vary the document in the chosen document series. The *average query response speed* for q is the average of the response speeds for q over the document series (see Figure 4), while the *average data scalability factor* for q is the average of the data scalability factors for q over the documents series (see Figure 5). Finally, the *average benchmark response speed* is the benchmark response speed averaged over the document series, and the *average benchmark scalability factor* is the benchmark scalability factor averaged over the document series. These last two measures are scalar values and they give an immediate indication about the efficiency and the data scalability of the XPath engine. The average benchmark response speeds we obtained for the implemented processors are: 10.79 MB/sec for TopXPath1, 10.70 MB/sec for TopXPath2, and 7.76 MB/sec for BottomXPath. The average benchmark scalability factors are: 0.93 MB/sec for TopXPath1, 0.95 MB/sec for TopXPath2, and 0.95 MB/sec for BottomXPath.

In the following we analyze the outcomes of our experimental evaluation:

- The response times of the two top-down strategies are very close, with TopXPath2 slightly faster than TopXPath1. This tells us that the approach of maintaining the context sequences document sorted at any time does not pay off in terms of response time.

- The top-down strategy is more efficient than the bottom-up one (about 30% faster, and the difference increases as the size of the data increases). The gap is bigger in the case of queries like Q1 and Q4 that do not need to explore big

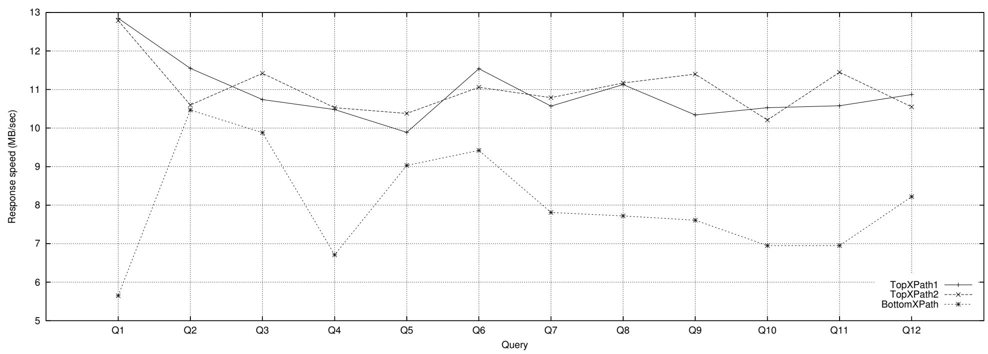

Figure 4. *The average query response speeds*

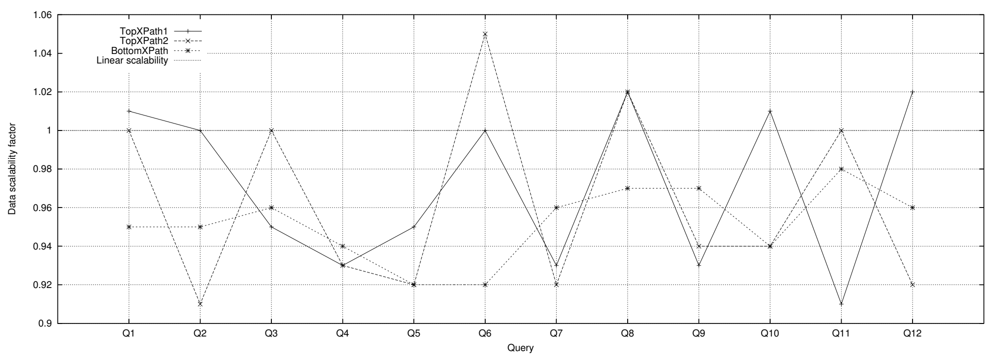

Figure 5. *The average data scalability factors*

portions of the document tree in order to compute the query answer, while the response times of the two strategies are similar in the case of queries like Q2 that need to visit the entire document. This phenomenon can be explained as follows: the bottom-up algorithm visits the entire document tree for each sub-query of the main query, while the top-down procedure explores only the tree zones that are relevant for the evaluation of the query.

- All the three XPath processors scale-up linearly (or even sub-linearly on small data) with respect to the size of the XML data. This confirms the linear-time complexity of the implemented algorithms.

8 Conclusion and future work

We implemented three evaluation strategies for the navigational fragment of XPath and we benchmarked the resulting XPath processors using a fragment of XPathMark, a recently proposed benchmark for XPath. The main outcomes of our investigation are (i) a top-down evaluation approach is faster than a bottom-up one, and (ii) the celebrated pre/post plane optimizations for XPath query evaluation are essentially as good as a foxy visit of the tree modeling the XML document.

It is worth pointing out that a bottom-up strategy outputs much more information than a top-down strategy. In particular, the bottom-up model checking-based procedure computes the answer set for *each* sub-query of the input query, while the top-down routine retrieves only those nodes belonging to the answer of the input query. This feature of the bottom-up approach may in fact become a benefit whenever the ansewer set for the sub-queries of the input query is relevant. Consider for instance a query processor that is queried many times possibly by different users. It is not unlikely that similar queries are posed at different times. In such a case, a bottom-up strategy may easily reuse the results computed for common sub-queries (in a dynamic programming fashion), while a top-down strategy must re-compute the result for each new query from scratch.

As a future work, we would like to compare the performace of the bottom-up and top-down approaches in a multi-query environment. Moreover, we intend to extend to developed evaluation system with different evaluation strategies, e.g., automata-based approaches. Another goal is to increase the supported language, e.g., with text(), id(), and position() XPath functions. On the modal logic side, we would like to investigate *top-down strategies* to solve the model checking problem for modal and temporal logics.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, 2000.

[2] L. Afanasiev, M. Franceschet, M. de Rijke, and M. Marx. CTL model checking for processing simple XPath queries. In *Proceedings of the International Symposium on Temporal Representation and Reasoning (TIME)*, pages 117–124. IEEE Computer Society Press, 2004.

[3] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic.* Cambridge University Press, 2001.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.

[5] M. Franceschet. XPathMark: an XPath benchmark for the XMark generated data. In *Proceedings of the International XML Database Symposium (XSym)*, volume 3671 of *LNCS*, pages 129–143, 2005. `http://www.science.uva.nl/~francesc/xpathmark`.

[6] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 95–106, 2002.

[7] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems (TODS)*, 30(2):444–491, 2005.

[8] T. Grust. Accelerating XPath location steps. In *ACM SIGMOD International Conference on Management of Data*, pages 109–120, 2002.

[9] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 524–525. Morgan Kaufmann Publishers, 2003.

[10] P. Hartel. A trace semantics for positive Core XPath. In *Proceedings of the International Symposium on Temporal Representation and Reasoning (TIME)*, pages 103–112. IEEE Computer Society Press, 2005.

[11] J. Hidders and P. Michiels. Efficient XPath axis evaluation for DOM data structures. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, pages 54–63, 2004.

[12] M. H. Kay. Saxon. An XSLT and XQuery processor. `http://saxon.sourceforge.net`, 2005.

[13] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 974–985, 2002. `http://monetdb.cwi.nl/xml/`.

[14] World Wide Web Consortium. Extensible Markup Language (XML). `http://www.w3.org/XML`, 1998.

[15] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`, 1999.

[16] World Wide Web Consortium. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery`, 2005.