High-Performance Complex Event Processing over Hierarchical Data[*]

BARZAN MOZAFARI, Massachusetts Institute of Technology
KAI ZENG, University of California, Los Angeles
LORIS D'ANTONI, University of Pennsylvania
CARLO ZANIOLO, University of California, Los Angeles

While complex event processing (CEP) constitutes a considerable portion of the so called Big Data analytics, current CEP systems can only process data having a simple structure, and are otherwise limited in their ability to efficiently support complex continuous queries on structured or semi-structured information. However, XML-like streams represent a very popular form of data exchange, comprising large portions of social network and RSS feeds, financial feeds, configuration files, and similar applications requiring advanced CEP queries. In this paper, we present the XSeq language and system that support CEP on XML streams, via an extension of XPath that is both powerful and amenable to an efficient implementation. Specifically, the XSeq language extends XPath with natural operators to express sequential and Kleene-* patterns over XML streams, while remaining highly amenable to efficient execution. In fact, XSeq is designed to take full advantage of the recently proposed Visibly Pushdown Automata (VPA), where higher expressive power can be achieved without compromising the computationally attractive properties of finite state automata. Besides the efficiency and expressivity benefits, the choice of VPA as the underlying model also enables XSeq go beyond XML streams and be easily applicable to any data with both sequential and hierarchical structures, including JSON messages, RNA sequences, and software traces. Therefore, we illustrate the XSeq's power for CEP applications through examples from different domains and provide formal results on its expressiveness and complexity. Finally, we present several optimization techniques for XSeq queries. Our extensive experiments indicate that XSeq brings outstanding performance to CEP applications: two orders of magnitude improvement is obtained over the same queries executed in general-purpose XML engines.

Categories and Subject Descriptors: H.2.3 [**Information Systems**]: DATABASE MANAGEMENT—*Languages, Query languages*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Complex Event Processing, Big Data Analytics, XML, JSON, Visibly Pushdown Automata

ACM Reference Format:
XXXX. ACM Trans. DB. Syst. 9, 4, Article 39 (March 2010), 39 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

---

[*] This manuscript is an extended version of a conference paper [Mozafari et al. 2012], now augmented with new applications (Sections 3.6, 3.8), formal semantics (Section 5), and proofs and complexity results (Sections 6.1, 6.2 and Appendix B).

This work was supported in part by NSF (Grant No. IIS 1118107).

Author's addresses: B. Mozafari, Computer Science and Artificial Intelligence Laboratory (CSAIL), MIT; K. Zeng, Computer Science Department, UCLA; L. D'Antoni, Department of Computer and Information Science, University of Pennsylvania. C. Zaniolo, Computer Science Department, UCLA;

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

```
  <result>{
for $t1 in doc("auction.xml")//Stock[@stock_symbol=‘DAGM’] return
<head>{$t1/@close}{
  for $t4 in $t1/following-sibling::Stock[@stock_symbol=‘DAGM’] where $t4/@close<=$t1/@close
   and (every $t2 in for $x in $t1/following-sibling::Stock[@stock_symbol=‘DAGM’]
                    where $x<<$t4 return $x satisfies $t2/@close<=$t1/@close and $t2/@close>=$t4/@close)
   and (every $t2 in for $x in $t1/following-sibling::Stock[@stock_symbol=‘DAGM’]
                    where $x<<$t4 return $x, $t3 in for $x in $t2/following-sibling::Stock[@stock_symbol=‘DAGM’]
                    where $x<<$t4 return $x satisfies $t2/@close>=$t3/@close and $t3/@close>=$t4/@close)
  return <bottom> {$t4/@close} </bottom>
}  </head>
}</result>
```

Fig. 1. A query in XPath 2.0/XQuery for a sequence of ‘falling price’ in Nasdaq’s XML.

1. INTRODUCTION

XPath is an important query language on its own merits and also because it serves as the kernel of other languages used in a wide range of applications, including XQuery, several graph languages [Strömbäck and Schmidt 2009], and OXPath for web information extraction [Furche et al. 2011]. Much work has also focused on the efficient support for XPath in the diverse computational environments required by these applications. In particular, finite state automata (FSA) have proven to be very effective at supporting XPath queries over XML streams [Koch 2009], and are also apt at providing superior scalability through the right mix of determinism versus non-determinism. In fact, numerous XML engines have been successfully built for efficient and continuous processing of XML streams [Chen et al. 2006; Peng and Chawathe 2003; Olteanu et al. 2003; Barton and et. al. 2003; Josifovski et al. 2005; Florescu and et. al. 2003; Diao et al. 2003]. All these systems support full or fragments of XPath or XQuery, and thus, naturally inherit the pros and cons of these languages. The simplicity of XPath and the generality of XQuery have made them very successful and effective for general-purpose applications. However, these languages lack explicit constructs for expressing Kleene-* and sequential patterns—a vital requirement in many CEP applications[1]. As a result, while the existing engines remain very effective in general-purpose applications over XML streams, their usability for CEP applications (that involve complex patterns) becomes highly limited as none of these engines provide any explicit sequencing/Kleene-* constructs over XML.

To better illustrate the difficulty of expressing sequence queries in existing XML engines (that mostly support fragments of XPath/ XQuery), in Fig. 1 we have expressed a common query from stock analysis in XPath 2.0, where the user is interested in a sequence of stocks with falling prices[2]. As shown in this example, due to the lack of explicit constructs for sequencing and Kleene-* patterns, the query in XPath/ XQuery is very hard to write and understand for humans and is also difficult to optimize. We will return to this query in Section 3, and show that it can be easily expressed using simple sequential constructs (see Example 3.1 and Query 9). In fact, it is not a surprise that the general-purpose XML engines perform two orders of magnitude slower on these complex sequential queries than the same queries expressed and executed in XSeq (the language and system presented in this paper), whereby explicit constructs for

[1]There are several definitions of CEP applications [Brenna et al. 2009; Luckham 2001; Wu et al. 2006], but they commonly involve three requirements: (i) complex predicates (filtering, correlation), (ii) temporal/order/sequential patterns, and (iii) transforming the event(s) into more complex structures. In this paper we mainly focus on (i) and (ii) while achieving (iii) represents a direction for future research, e.g. by embedding our language (called XSeq) inside XSLT.

[2]In fact, in practice, stock queries tend to be much more complex, e.g. in a wedge pattern (`www.investopedia.com`), the user seeks an arbitrary number of falling and rising phases of a stock.

Kleene-* patterns and effective VPA-based optimizations allow for high-performance execution of CEP queries.

These limitations of XPath are not new, as several extensions of XPath have been previously proposed in the literature [ten Cate 2006; ten Cate and Marx 2007b; 2007a]. However, the efficient implementation of even these extensions (often referred to as Regular XPath) remained an open research challenge, which the papers proposing said extensions did not tackle (neither for stored data nor for data streams). In fact, the following was declared to be an important open problem since 2006 [ten Cate 2006]: *"Efficient algorithms for computing the transitive closure of XPath path expressions"*.

Fortunately, significant advances have been recently made in automata theory with the introduction of Visibly Pushdown Automata [Alur and Madhusudan 2004; 2006]. VPAs strike a balance between expressiveness and tractability: unlike pushdown automata (PDA), VPAs have all the appealing properties of FSA (a.k.a. word automata). For instance, VPAs enjoy higher expressiveness (than word automata) and more succinctness (than tree automata), while their decision complexity and closure properties are analogous to word automata, e.g., VPAs are closed under union, intersection, complementation, concatenation, and Kleene-*; their deterministic versions are as expressive as their non-deterministic counterparts; and membership, emptiness, language inclusion and equivalence are all decidable [Alur and Madhusudan 2004; 2006]. However unlike word automata, VPAs can model and query any *well-nested* data, such as XML, JSON files, RNA sequences, and software traces [Alur and Madhusudan 2006]. What these seemingly diverse set of formats have in common is their dual-structures: (i) they all have a sequential structures (e.g. there is a global order of the tags in a JSON or XML file based on the order that they appear in the document), (ii) they also have a hierarchical structure (when XML elements or JSON objects are enclosed in one another), but (iii) this hierarchical structure is well-nested, e.g. the open tags in the XML documents match with their corresponding close tags. Data with these properties can be formally modeled as *Nested Words* or *Visibly Pushdown Words* [Alur and Madhusudan 2004; 2006]. (We have included a brief background on nested words and VPAs in Appendix A.) Throughout this paper we refer to such formats as 'XML-like' data, but for the most part we focus on XML[3].

Although these new types of automata can bring major benefits in terms of expressive power, to the best of our knowledge, their optimization and efficient implementation in the context of XPath-based query languages have not been explored before. Hence, in this paper, we introduce the XSeq language which achieves new levels of expressive power supported by a very efficient implementation technology. XSeq extends XPath with powerful constructs that support (i) the specification of and search for complex sequential patterns over XML-like structures, and (ii) efficient implementation using the Kleene-* optimization technology and streaming Visibly Pushdown Automata (VPA).

Being able to compile complex pattern queries into equivalent VPAs has several key benefits. First, it allows for expressing complex queries that are common in CEP applications. Second, it allows for efficient stream processing algorithms. Finally, the closeness of VPAs under *union* operation creates the same opportunities for CEP systems (through combining their corresponding VPAs) that the closeness of NFAs (non-deterministic finite automata) created for publish-subscribe systems [Diao et al. 2003; Vagena et al. 2007; Laptev and Zaniolo 2012], where simultaneous processing of mas-

[3]Using XSeq to query other XML-like data (e.g. JSON, RNA, software traces) is straightforward and only involves introducing domain-specific interfaces on top of XSeq, e.g. see [Zeng et al. 2013] for a few examples of such interfaces.

sive number of queries becomes possible through merging the corresponding automata of the individual queries.

Contributions. In summary, we make the following contributions:

(1) The design of XSeq, a powerful and user-friendly query language for CEP over XML streams or stored sequences.
(2) An efficient implementation for XSeq based on VPA-based query plans, and several compile-time and run-time optimizations.
(3) Formal results on the expressiveness of XSeq, and the complexity of its query evaluation and query containment.
(4) An extensive empirical evaluation of XSeq system, using several well-known benchmarks, datasets and engines.
(5) Our XSeq engine can also be seen as the first optimization and implementation for several of the previously proposed languages that are subsumed in XSeq but were never implemented (e.g. Regular XPath [ten Cate 2006], Regular XPath(W) [ten Cate and Segoufin 2008] and Regular XPath$^{\approx}$ [ten Cate and Marx 2007b]).

Paper Organization. We present the main constructs of our language in Section 2 using simple examples. The generality and versatility of XSeq for expressing CEP queries are illustrated in Section 3 where several well-known queries are discussed. Our query execution and optimization techniques are presented in Section 4. In order to study the expressiveness and complexity of our language, we first provide formal semantics for XSeq in Section 5, which is followed by our formal results in Section 6, including the translation of XSeq queries into VPAs, their MSO-completeness and their query evaluation and query containment complexities. Our XSeq engine is empirically evaluated in Section 7, which is followed by an overview of the related work in Section 8. Finally, we conclude in Section 9. For completeness, we have also included a brief background on VPAs in Appendix A.

2. XSEQ QUERY LANGUAGE

In this section, we briefly introduce the query language supported by our CEP system, called XSeq. The simplified syntax of XSeq is given in Fig. 2 which suffices for the sake of this presentation. Below we explain the semantics of XSeq via simple examples. We defer the formal semantics to Section 5.

Inherited Constructs from Core XPath. The navigational fragments of XPath 1.0 and 2.0 are called, respectively, Core XPath 1.0 [ten Cate and Marx 2007b] and Core XPath 2.0 [ten Cate and Marx 2007a]. The semantics of these common constructs are similar to XPath (e.g., axes, attributes). Other syntactic constructs of XPath (e.g., the $following$ axis) can be easily expressed in terms of these main constructs (see [ten Cate and Marx 2007a]). In XSeq there are two new axes to express the *immediately following* [4] notion, namely $first\_child$ and $immediate\_following\_sibling$, which are described later on. Some of the axes in XSeq have shorthands:

[4] XSeq does not have analogous operators for immediately preceding since backward axes of XPath are rarely used in practice.

$$\begin{array}{rcl}
\text{XSeqQuery} & \leftarrow & [\mathit{'return'}\ \text{Output}\ \mathit{'from'}]\ \text{Pattern} \\
& & [\mathit{'where'}\ \text{Condition}]\ [\mathit{'partition\ by'}\ \text{Pattern}] \\
\text{Output} & \leftarrow & \mathit{Operand}\ [','\ \text{Output}] \\
\text{Pattern} & \leftarrow & [\mathit{'doc()'}]\ \text{PathExpr} \\
\text{PathExpr} & \leftarrow & \text{Step} \\
& & \mid \text{PathExprDefinition} \\
& & \mid \text{PathExpr}\ \text{PathExpr} \\
& & \mid '('\ \text{PathExpr}\ ')'\ '*' \\
& & \mid \text{PathExpr}\ \mathit{'union'}\ \text{PathExpr} \\
& & \mid \text{PathExpr}\ \mathit{'intersect'}\ \text{PathExpr} \\
\text{PathExprDefinition} & \leftarrow & '('\ \mathit{Variable}\ ':'\ \text{PathExpr}\ ')' \\
\text{Step} & \leftarrow & \text{Axis}\ \text{NameTest}\ \text{Predicate}\ * \\
\text{Axis} & \leftarrow & \text{AxisSpecifier}\ '::'\ \mid \text{AbbreviatedAxisSpecifier} \\
\text{AxisSpecifier} & \leftarrow & \mathit{'self'} \mid \mathit{'child'} \mid \mathit{'parent'} \mid \mathit{'descendant'} \mid \mathit{'ancestor'} \\
& & \mid \mathit{'attribute'} \mid \mathit{'following\_sibling'} \mid \mathit{'preceding\_sibling'} \\
& & \mid \mathit{'first\_child'} \mid \mathit{'immediate\_following\_sibling'} \\
\text{AbbreviatedAxisSepcifier} & \leftarrow & '\cdot' \mid '/' \mid '//' \mid '@' \mid '\backslash' \mid '/\backslash' \\
\text{NameTest} & \leftarrow & \mathit{QName} \mid '*' \mid \mathit{Variable} \mid \text{KindTest} \\
\text{KindTest} & \leftarrow & \mathit{'node()'} \mid \mathit{'text()'} \\
\text{Predicate} & \leftarrow & '['\ (\text{Pattern} \mid \text{Condition})\ ']' \\
\text{Condition} & \leftarrow & \mathit{BoolExpr} \\
\text{Operand} & \leftarrow & \mathit{Constant} \mid \text{Alias}\ \text{PlainStep}\ *\ (\text{AttributeStep} \mid \text{TextStep}) \\
& & \mid \text{Aggregate}\ '('\ \mathit{ArithmeticExpr}\ ')' \\
\text{PlainStep} & \leftarrow & \text{Axis}\ \mathit{QName} \\
\text{AttributeStep} & \leftarrow & (\mathit{'attribute'}\ '::'\ \mid '@')\ \mathit{QName} \\
\text{TextStep} & \leftarrow & (\mathit{'child'}\ '::'\ \mid '/')\ \mathit{'text()'} \\
\text{Aggregate} & \leftarrow & \mathit{'max'} \mid \mathit{'min'} \mid \mathit{'count'} \mid \mathit{'sum'} \mid \mathit{'avg'} \\
\text{Alias} & \leftarrow & \text{SequenceAlias} \mid \text{PlainAlias} \\
\text{SequenceAlias} & \leftarrow & (\mathit{'prev'} \mid \mathit{'first'} \mid \mathit{'last'})\ '('\ \mathit{Variable}\ ')' \\
\text{PlainAlias} & \leftarrow & \mathit{Variable}
\end{array}$$

Fig. 2. XSeq Syntax (QName, Variable, BoolExpr, Constant, and ArithmeticExpr are defined in the text).

Axis	Shorthand
self	•
child	/
descendant	//
attribute	@
following_sibling	λ (empty string, i.e. default axis)
first_child	/\
immediate_following_sibling	\

Conditions. In XSeq, a `Condition` can be any predicate which is a boolean combination of *atomic formulas*. An atomic formula is a *binary operator* applied to two *operand*s. A *binary operator* is one of $=$, $\neq$, $<$, $>$, $\leq$, $\geq$. An *operand* is any algebraic combination (using +, -, etc.) and aggregates of string or numerical constants, and the attributes or text contents of variable nodes.

Example 2.1 (**A family tree.**). Our XML document is a family tree where every node has several attributes: Cname (for name), Bdate (for birthdate), Bplace (for the city of birth) and each node can contain an arbitrary number of sub-entities Son and Daughter. Under each node, the siblings are ordered by their Bdate.

In the following, we use this schema as our running example.

Example 2.2. Find the birthday of Mary's sons.

QUERY 1.

```
//daughter[@Cname='Mary'] /son /@Bdate
```

Kleene-* and parentheses. Similar to Regular XPath [ten Cate 2006] and its dialects [ten Cate and Marx 2007b; ten Cate and Segoufin 2008], XSeq supports path expressions such as /a(/b/c)*/d, where a Kleene-* expression A^* is defined as the infinite union $\cdot \cup A \cup (A/A) \cup (A/A/A) \cup \cdots$

Example 2.3. Find those sons born in 'New York', who had a chain of male descendants in which all the intermediary sons were born in 'Los Angeles' and the last one was again born in 'New York'. For all such chains, return the name of the last son.[5]

QUERY 2.

```
// son[@Bplace='NY'] (/son[@Bplace='LA'])* /son[@Bplace='NY'] /@Cname
```

The parentheses in $()^*$ can be omitted when there is no ambiguity. Also, note the difference between the semantics of (/son)∗ and //son: the latter only requires a son in the last step rather than the entire path.

Syntactic Alternatives. In XSeq, the node selection conditions can be alternatively moved to an optional where clause, in favor of readability. When a condition is moved to the where clause, its step should be replaced with a variable (variables in XSeq start with $). Also, similarly to XPath 2.0 and XQuery, the query output in XSeq can be moved to an optional return clause. Query 3 below is an alternative way of writing Query 2 in XSeq. Here, tag($X) returns the tag name of variable $\$X$.

QUERY 3.

```
return $B@Cname
from //son[@Bplace='NY'] (/$A)* /$B[@Bplace='NY']
where tag($A)='son' and $A@Bplace='LA' and tag($B)='son'
```

For clarity, in this paper we mainly use this alternative syntax.

Order Semantics, Aggregates. XSeq is a *sequence query language*. Therefore, unlike XPath where the input and output are a set (or binary relation), in XSeq the XML stream is viewed as a pre-order traversal of the XML tree. Thus, both the input and the output of an XSeq query are a *sequence*. The XML nodes are ordered according to[6] their relative position in the XML document.

As a result, besides the traditional aggregates (e.g., sum, max), XSeq also supports sequential aggregates (SeqAggr in Fig. 2) which are only applied to variables under a Kleene-* For instance, the path expression /son(/$X)*, last($X) @name returns the name of the last X in the (/$X)* sequence. Similarly, first($X) returns the first node of the (/$X)* and prev($X) returns the node before the current node of the sequence.

[5]This is an example of a well-known class of XML queries which has been proven [ten Cate 2006] as not expressible in Core XPath 1.0.

[6]When a WINDOW is defined over the XML stream, the input nodes can be re-ordered. For simplicity of the discussion, we do not discuss re-ordering.

Finally, `$X @Bdate` $>$ `prev($X) @Bdate` ensures that the nodes that match `(/$X)*` are in increasing order of their birth date.

Siblings. Since XSeq is designed for complex sequential queries, its default axis (i.e. when no explicit axis is given) is the 'following_sibling'. The omission of the 'following_sibling' allows for concise expression of complex horizontal patterns.

Example 2.4. Find all the younger brothers of 'Mary'.

QUERY 4.

```
return $S@Cname
from //$D[@Cname='Mary'] $S
where tag($D)='daughter' and tag($S)='son'
```

Here, since no other axes appear between D and S, they are treated as siblings.

Immediately Following. This is the construct that gives XSeq a clear advantage over all the previous extensions of XPath in terms of expressiveness, succinctness and optimizability. We believe that one of the main shortcomings of the previous XML languages for CEP applications is their lack of explicit constructs for expressing the notion of '*immediately following*' (see Section 3). Thus, to overcome this, XSeq provides two explicit axes, \ and /\, for *immediately following* semantics. For example, `Y\X` will return the *immediately next sibling* of node `Y`, while `Y/\X` will return the *very first child* of node `Y`. Similarly to other constructs, these operators return an empty set if no such node can be found, e.g., when we are at the last sibling or a node with no children.

Example 2.5. Find the first two elder siblings of 'Mary'.

QUERY 5.

```
return $X@Cname, $Y@Cname
from  //daughter[@Cname='Mary'] \$X \$Y
```

Example 2.6. Find the second child of 'Mary'.

QUERY 6.

```
return $Y@Cname
from  //daughter[@Cname='Mary'] /\$X \$Y
```

Partition By. Inspired by relational Data Stream Management Systems (DSMS), XSeq supports a partitioning operator that is very essential for many CEP applications. Nodes can be partitioned by their key, so that different groups can be processed in parallel as the XML stream arrives. Although this construct does not add to the expressiveness, it provides a more concise syntax for complex queries and better opportunities for optimization. However, XSeq only allows partitioning by an attribute field and requires that except this attribute, the rest of the path expression in the partitioning clause be a prefix of the path expression in the from clause. This constraint is important for ensuring efficiency and also for avoiding queries with ill semantics.

Example 2.7. For each city, find the oldest male born there.

By knowing the cities that are present in our XML, we could write several queries, one for each city e.g., `min(//son[@Bplace =' LA'] @Bdate)`. However, in streaming applications such information is generally not provided a priori. Moreover, instead of running several queries over the same stream, an explicit partition by clause allows for simultaneous handling of different key values and is much easier to optimize. For instance:

QUERY 7.

```
return $X @Bplace, min($X @Bdate)
from //$X
where tag($X) = 'son'
partition by //son @Bplace
```

Path Complementation. XSeq does not provide explicit constructs for path complementation (e.g., `except` in XPath 2.0). This restriction does not reduce XSeq's expressivity, as it has been shown that path complementation can be expressed using Kleene-* and path intersection [Cate and Lutz 2009]. The reason behind this restriction in XSeq is that, by forcing the programmer to simulate the negation with other constructs, the resulting query is often more amenable to optimization. For instance, the query of Example 2.3 could be expressed in XPath 2.0 using their `except` operator as:

```
//son[@Bplace='NY']//son[@Bplace='NY']@Cname
except
//son[@Bplace='NY']//son[@Bplace != 'LA']//son[@Bplace='NY']@Cname
```

However, as shown in Query 2, this query can be expressed in XSeq without using the negation.

Path Variables. In Query 3, we showed how variables in XSeq could replace the NameTest of a Step. Such variables are called *step variables*. In practice, and in fact in all the real world examples of Section 3, we hardly need any feature beyond these step variables. However, for more expressive power[7], XSeq also supports the so-called *path variables* that can replace path expressions, as shown in the PathExprDefinition rule of Fig. 2.

Example 2.8. Find daughter followed by a sequence of siblings with alternating genders, namely daughter, son, daughter, son, and so on.

QUERY 8.

```
return first($Z) $X @Cname
from // ($Z: $X $Y $Z)
where tag($X) = 'daughter' and tag($Y) = 'son'
```

This query defines the path variable $\$Z$ as $\$X\ \$Y\ \$Z$ which means $\$Z$ is recursively defined as $\$X\ \$Y$ followed by itself. In this particular example, $(\$Z : \$X\ \$Y\ \$Z)$ is equivalent to $(\$Z : \$X\ \$Y)^*$, but in general not all recessive path variables can be replaced with Kleene-*[8]. Also, note that path variables do not have to be recursive, e.g. $\$Z$ in $(\$Z : \$X\ \$Y)^*$ is a valid path variable too.

The same step variable can appear multiple times in the from clause. However, for path variables we differentiate between their definition and their reference. XSeq requires that path variables be defined only once in the from clause. For instance,

```
return first($Z) $X @Cname
from // ($Z: $X $Y $Z) $Z
where tag($X) = 'daughter' and tag($Y) = 'son'
```

is a valid query, but the following query is not allowed:

[7]This particular feature of XSeq is interesting from a theoretical point of view, as it makes the language Monadic Second Order (MSO)-complete, thus, subsuming previous extensions of XPath.
[8]Recursive path variables are a more powerful form of recursion than than Kleene-*. See Section 6.

```
return first($Z) $X @Cname
from  // ($Z: $X $Y $Z) ($Z: $X)
where tag($X) = 'daughter' and tag($Y) = 'son'
```

as it redefines the path variable $\$Z$.

Moreover, there is also a restriction on how path variables can be referenced in the from clause [9]. Before explaining this restriction, we first need to define the concepts of $yield$ and $nend$ for a path variable.

Definition 2.9. For a path variable $\$X$ defined as $(\$X : P)$, where P is a path expression, we define the $nend(\$X)$ as all the path variables in P which do not appear at the end of a production for P. We also recursively define $yield(\$X) = \bigcup_{\$Y \in P} yield(\$Y) \cup \{\$Y\}$ where $\$Y$ iterates over all the path variables appearing in P.

For instance, for $(\$X : \$X\ \$Y\ \$X)(\$Y : /son/\$Z)$ as the from clause, $nend(\$X) = \{\$X, \$Y\}$ and $yield(\$X) = \{\$X, \$Y, \$Z\}$. Now, we are ready to formally define the restriction on referencing path variables: Path variables in XSeq can appear multiple times in the from clause, as long as the following rule is not violated:

RULE 1. *For every path variable defined as* $(\$X : P)$, $\$X \notin yield(\$Y)$ *for* $\forall \$Y \in nend(\$X)$.

Intuitively, this rule disallows circular definitions of path variables. The reason behind this restriction is that allowing arbitrary references to a path variable can make the language non-regular, and hence not amenable to efficient implementation[10].

Other Constructs in XSeq. `union` and `intersect` have the same semantics as in XPath. If the user desires an XML output, he can embed the XSeq query in an XQuery or XSLT expression. Formatting the output is out of the scope of this paper and makes an interesting future direction of research. Instead, in this paper, we only focus on the query expression and its efficient execution for CEP applications.

In the next section, we will use these basic constructs to express more advanced queries from a wide range of CEP applications.

3. ADVANCED QUERIES FROM COMPLEX EVENT PROCESSING

In this section we present more complex examples from several domains and show that XSeq can easily express such queries.

3.1. Stock Analysis

Consider an XML stream of stock quotes as defined in Fig. 3. Let us start with the following example.

Example 3.1 (**Falling pattern**). Find those stocks whose prices are decreasing.

QUERY 9 (FALLING PATTERN IN XSEQ).

```
return last($X)@price
from /stocks /$Z (\$X)*
where tag($Z) = 'transaction' and tag($X) = 'transaction'
  and $X@price < prev($X)@price
partition by /stocks /transaction@company
```

[9]Later in Section 6.1, we define the restriction rule more formally

[10]For example, allowing $(\$X : a\ \$X\ \$Y)(\$Y : b)$ would represent the pattern $a^n b^n$ which is not MSO expressible.

```
<! DOCTYPE stocks [
<! ELEMENT stocks (transaction*)>
<! ATTLIST transaction company CDATA #REQUIRED>
<! ATTLIST transaction price CDATA #REQUIRED>
<! ATTLIST transaction buyer IDREF #REQUIRED>
<! ATTLIST transaction date CDATA #REQUIRED>
]>
```

Fig. 3. The DTD for the stream of Nasdaq transactions.

This is in fact the same query as the one we had expressed in XPath 2.0 in Fig. 1. Comparing the convoluted query of Fig. 1 with Query 9 clearly illustrates the importance of having explicit constructs for sequential and Kleene-* constructs in enabling CEP applications. This clarity and succinctness at the language level provide more opportunities for optimization which eventually translate to more efficiency, as shown in Sections 4 and sec:experiments, respectively. Next, let us consider the 'V'-shape pattern which is a well-known query in stock analysis.

Example 3.2 (**'V'-shape pattern**). Find those stocks whose prices have formed a 'V'-shape. That is, the price has been going down to a local minimum, then rising up to a local maximum which was higher than the starting price.

The 'V'-shape query only exemplifies many important queries from stock analysis [11] that are provably impossible to express in Core XPath 1.0 and Regular XPath, simply both of these languages lack the notion of 'immediately following sibling' in their constructs. XPath 2.0, however, can express these queries through the use of its `for` and quantified variables: using these constructs, XPath 2.0 can 'simulate' the concept of 'immediately following sibling' in XPath 2.0 by double negation, i.e. ensuring that 'for each pair of nodes, there is nothing in between'. But this approach leads to very convoluted XPath expressions which are extremely hard to write/understand and almost impossible to optimize (See Fig. 1 and Section 7).

On the other hand, XSeq can express this queries with its simple constructs that can be easily translated and optimized as VPA:

QUERY 10 ('V'-PATTERN IN XSEQ).

```
return last($Y)@price
from /stocks /$Z (\$X)* (\$Y)*
where tag($Z) = 'transaction'
  and tag($X) = 'transaction' and tag($Y) = 'transaction'
  and $X@price < prev($X)@price
  and $Y@price > prev($Y)@price
partition by /stocks /transaction@company
```

A more interesting pattern would be the falling wedge pattern, which shows the power of sequence aggregates in XSeq language.

Example 3.3 (**Falling wedge pattern**). Find those stocks whose price fluctuates as a series of 'V'-shape patterns, where in each 'V' the range of the fluctuation becomes smaller. Fig. 4(b) shows a falling wedge pattern.

QUERY 11 (FALLING WEDGE PATTERN IN XSEQ).

```
return $R @price, last($Y) @price
```

[11] `http://www.chartpattern.com/`

```
from /stocks /$R ((\$S)* \$X (\$T)* \$Y)*
where tag($R) = 'transaction' and tag($S) = 'transaction'
  and tag($X) = 'transaction' and tag($T) = 'transaction'
  and tag($Y) = 'transaction'
  and $R @price > first($S) @price
  and prev($S) @price > $S @price
  and last($S) @price > $X @price
  and $X@price < first($T) @price
  and prev($T) @price < $T @price
  and last($T) @price < $Y @price
  and prev($X) @price < $X @price
  and prev($Y) @price > $Y @price
partition by /stocks /transaction @company
```

3.2. Social Networks

Twitter provides an API[12] to automatically receive the stream of new tweets in several formats, including XML. Assume the tweets are ordered according to their date timestamp:

```
<! DOCTYPE twitter [
<! ELEMENT twitter ((tweet)*)>
<! ELEMENT tweet (message)>
<! ELEMENT message (#PCDATA)>
<! ATTLIST tweet tweetid CDATA #REQUIRED>
<! ATTLIST tweet userid CDATA #REQUIRED>
<! ATTLIST tweet date CDATA #REQUIRED> ]>
```

Example 3.4 (*Detecting active users*). In a stream of tweets, report users who have been active over a month. A user is active if he posts at least a tweet every two days.

This query, if not impossible, would be very difficult to express in XPath 2.0 or Regular XPath. The main reason is that, again due to their lack of 'immediate following', they cannot easily express the concept of "adjacen" tweets.

QUERY 12 (DETECTING ACTIVE USERS IN XSEQ).

```
return first($T) @userid
from /twitter /$Z (\$T)*
where tag($Z) = 'tweet' and tag($T) = 'tweet'
  and $T@date-prev($T)@date < 2
  and last($T)@date-first($T)@date > 30
partition by /twitter /tweet @userid
```

3.3. Inventory Management

RFID has become a popular technology to track inventory as it arrives and leaves retail stores. Below is a sample schema of events, where events are ordered by their timestamp:

```
<! DOCTYPE events [
<! ELEMENT events (event*)>
<! ELEMENT event (message)>
<! ELEMENT message (#PCDATA)>
<! ATTLIST event ts CDATA #REQUIRED>
<! ATTLIST event itemid CDATA #REQUIRED>
<! ATTLIST event eventtype CDATA #REQUIRED> ]>
```

Example 3.5 (*Detecting Item Theft*). Detect when an item is removed from the shelf and then removed from the store without being paid for at a register.

[12] http://dev.twitter.com/

QUERY 13 (DETECTING ITEM THEFT IN XSEQ).

```
return $T@itemid
from /events /$T \$W* \$X
where tag($T) = ‘event’ and tag($W) = ‘event’ and tag($X) = ‘transaction’
  and $T@eventtype = ‘removed from shelf’
  and $X@eventtype = ‘removed from store’
  and $W@eventtype != ‘paid at register’
partition by /events/event@itemid
```

3.4. Directory Search

Consider the following first-order binary relation which is familiar from temporal logic [ten Cate and Marx 2007b]:

$$\begin{aligned}\phi(x,y) = {} & \text{descendant}(x,y) \wedge q(y) \wedge \\ & \forall z(\text{descendant}(x,z) \wedge \text{descendant}(z,y) \rightarrow p(z))\end{aligned}$$

For instance, for a directory structure that is represented as XML, by defining q and p predicates as $q(y)$: ‘y is a file’ and $p(z)$: ‘z is a non-hidden folder’, the ϕ relation becomes equivalent to the following query:

Example 3.6. Retrieve all reachable files from the current folder by repeatedly selecting non-hidden subfolders.

According to the results from [ten Cate and Marx 2007b], such queries are not expressible in XPath 1.0. This query, however, is expressible in XPath 2.0 but not very efficiently. E.g.,

`//file except //folder[@hidden=‘true’]//file`

Such queries can be expressed much more elegantly in XSeq (and also in Regular XPath):

QUERY 14 (ϕ QUERY IN XSEQ).

```
(/folder[@hidden = ‘false’])* /file
```

3.5. Genetics

Haemophilia is one of the most common recessive X-chromosome disorders. In genetic testing and counseling, if the fetus has inherited the gene from an affected grandparent the risk to the fetus is 50% [Alexander et al. 2000]. Therefore, the inheritance risk for a person can be estimated by tracing the history of haemophilia among its even-distance ancestors, i.e. its grandparents, its grand-parents' grand-parents, and so on.

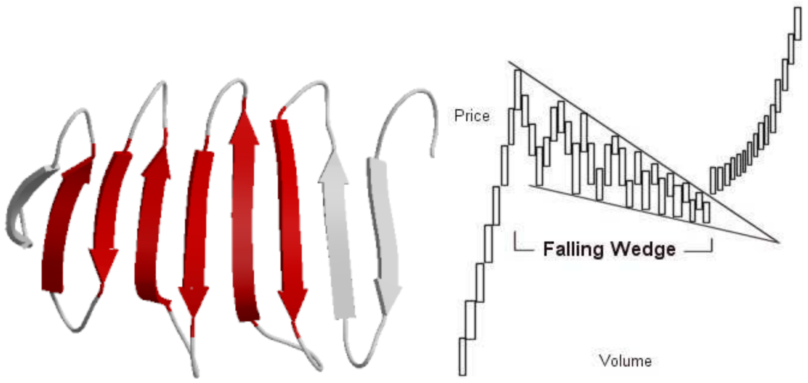
Example 3.7. Given an ancestry XML which contains the history of haemophilia in the family, identify all family members who are at even-distance from an affected member, and hence, at risk.

This query cannot be easily expressed without Kleene-* [Cate and Lutz 2009], but is expressible in XSeq:

QUERY 15 (DESCENDANTS OF EVEN-DISTANCE FROM A NODE).

```
return $Z @Cname
from //$X[@haemophilia = ‘true’] (/$Y /$Z)*
```

Queries 14 and 15 are not expressible in XPath 1.0, are expressible in XPath 2.0 but not efficiently, and are easily expressible in Regular XPath and XSeq.



(a) (b)

Fig. 4. (a) The β-meander motif (b) The falling wedge pattern

3.6. Protein, RNA and DNA Databases

The world-wide community of life scientists has access to a large number of public bioinformatics databases and tools. As more and more of the resources offer programmatic web-service interface, XML becomes a widely-used standard data exchange format for basic bioinformatics data. Many public bioinformatics databases provide data in XML format.

Proteins, RNA and DNA are sequences of linear structures, but they are usually with complex secondary or even higher-order structures which play important roles in their functionality. Searching complex patterns in these rich-structured sequences are of great importance in the study of genomics, pharmacy and so on. XSeq provides a powerful declarative query language for access bioinformatics databases, which enables complex pattern searching.

For instance, the *structural motifs* are important supersecondary structures in proteins, which have close relationships with the biological functions of the protein sequences. These motifs are of a large variety of structural patterns, usually very complex, e.g., the β-meander motif is composed of two or more consecutive antiparallel β-strands linked together, as depicted in Fig. 4(a)[13], while each β-strand is typically 3 to 10 amino acid. Consider now protein data with a simplified schema as below. Example 16 uses XSeq to detect such motifs.

```
<!DOCTYPE uniprot [
<!ELEMENT uniprot (protein)*>
<!ELEMENT protein (fullName, feature+)>
<!ELEMENT fullName (#PCDATA)>
<!ATTLIST feature type CDATA #REQUIRED> ]>
```

QUERY 16 (DETECTING β-MEANDER MOTIFS).

```
return $N/text()
from //protein[$N] /$F \$G (\$H)*
where tag($N) = ‘fullName’
  and tag($F) = ‘feature’ and $F@type = ‘beta-strand’
  and tag($G) = ‘feature’ and $G@type = ‘beta-strand’
  and tag($H) = ‘feature’ and $H@type = ‘beta-strand’
```

[13]http://en.wikipedia.org/wiki/Beta_sheet

3.7. Temporal Queries

Expressing temporal queries represents a long-standing research interest. A number of language extentions and ad-hoc solutions have been proposed. Traditional temporal databases use a state-oriented representation, where tuples of a database are time-stamped with their maximal period of validity. This state-based representation requires temporal coalescing and/or temporal joins even for basic query operations (e.g. projection), and are thus prone to inefficient execution. Some recent research work has proposed using XML-based event-oriented representation for transaction-time temporal database, where value updates in database history are recorded as events [Amagasa et al. 2000; Wang et al. 2008; Zaniolo 2009]. For example, below is the DTD of a temporal employee XML, where each employee has a sequence of *salary* and *dept* elements time-stamped by the *tstart*, *tend* attributes, representing the update events ordered by their start time in the database's evolution history.

```
<!DOCTYPE employees [
<!ELEMENT employees (employee*)>
<!ELEMENT employee (name (salary | dept)+)>
<!ATTLIST employee id CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT salary (#PCDATA)>
<!ELEMENT dept (#PCDATA)>
<!ATTLIST salary tstart CDATA #REQUIRED>
<!ATTLIST salary tend CDATA #IMPLIED>
<!ATTLIST dept tstart CDATA #REQUIRED>
<!ATTLIST dept tend CDATA #IMPLIED> ]>
```

XSeq is a powerful event-oriented temporal language, which can easily express basic temporal operations (e.g., temporal joins and temporal coalescing), as well as very complex temporal sequence patterns. This can be illustrated by the following examples.

First, let us consider the well-known RISING query which is a famous temporal aggregate introduced by TSQL2 [Snodgrass 2009].

Example 3.8 (*RISING*). What is the maximum time range during which an employee's salary is rising?

QUERY 17.

```
return max(last($X) @tend - first($X) @tstart)
from // $Z* (\$X)*
where tag($X) = ‘employee’
  and $X/salary/text() > prev($X)/salary/text()
  and $X@tstart <= prev($X)@tend
partition by //employee @id
```

Example 3.9. Find employees who have risen quickly without changing department. More precisely, we want to find employees who

(1) once hired (with some salary and into some department),
(2) have gone through one or more salary adjustments, followed by
(3) a transfer to another department,
(4) for a final salary that is 40% above the initial one.

This complex pattern can be expressed succinctly by Query 18.

QUERY 18.

```
return $X
from //employee[@$X] /$A \$B \$C (\$D)* \$E
where tag($A) = ‘salary’ and tag($B) = ‘dept’
```

```
and tag($C) = 'salary' and tag($D) = 'salary'
and tag($E) = 'dept' and tag($X) = 'id'
and $E/text() <> $B/text()
and last($D)/text() > 1.4 * $A/text()
```

3.8. Software Trace Analysis

Modern programming languages and software frameworks offer ample support for debugging and monitoring applications. For example, in the .NET framework, the *System.Diagnostics* namespace contains flexible classes which can be easily incorporated into applications to output runtime debug/trace information as XML files. The following XML snippet shows a software trace of a function `fibonacci` that recursively called itself but in the end threw out an exception.

```
<main>
  ...
  <fibonacci @input = '500'>
    <fibonacci @input = '499'>
     ...
      <exception @msg = 'overfolow' />
    </fibonacci>
  </fibanacci>
  ...
</main>
```

Searching and analyzing the patterns in software traces could help debugging. For example, we can easily identify the input to the last iteration of the function `fibonacci` and the depth of the recursive calls by

QUERY 19.

```
return last($F) @input, count($F)
from //$X (/$F)* /$E
where tag($X) != 'fibonacci'
  and tag($F) = 'fibonacci'
  and tag($E) = 'exception'
```

4. XSEQ OPTIMIZATION

The design and choice of operators in XSeq is heavily influenced by whether they can be efficiently evaluated or not. Our criterion for efficiency of an XSeq operator is whether it can be mapped to a Visibly Pushdown Automaton (VPA). The rationale behind choosing VPA as the underlying query execution model is two-fold. First, XSeq is mainly designed for complex patterns and patterns can be intuitively described as transitions in an automaton: fortunately, VPAs are expressive enough to capture all the complex patterns that can be expressed in XSeq. Secondly, VPAs retain many attractive computational properties of finite state automata on words [Alur and Madhusudan 2004]. In fact, by translation into VPAs, we can exploit several existing algorithms for streaming evaluation [Madhusudan and Viswanathan 2009] and optimization of VPAs [Mozafari et al. 2010a]. For unfamiliar readers, we have provided a brief background on VPAs in Appendix A.

In Section 4.1, we provide a high-level description of our algorithm for translating the most commonly used operators of XSeq (other operators are covered in Section 6) into equivalent VPAs which can faithfully capture the same pattern in the input[14]. Then, in Sections 4.2 and 4.3, we present several static (compile-time) and run-time

[14]Informally, we say that an XSeq query and a VPA are equivalent when every portion of the input XML that produces an output result in the former will be also accepted by the latter and vice versa.

optimizations of VPAs in our XSeq implementation. In Section 7, we study the effectiveness of these optimizations in practice.

4.1. Efficient Query Plans via VPA

In this section, we describe an inductive algorithm to translate the most commonly used features of XSeq into efficient and equivalent VPAs. This algorithm can handle all forward axes, Kleene-*, and step variables (similar fragments to those studied in [Gauwin et al. 2011]).

Later, in Section 6, we also provide a general algorithm for translating any arbitrary XSeq query Q (including path variables and backward axes) into a VPA (which in general, can be larger and hence, less efficient) accepting all the input trees on which Q returns a non-empty set of results. However, in practice, most of commonly used queries (including all of those of Section 3) can be efficiently handled by the algorithm presented below.

Note that although the theoretical notion of VPAs only allows for transitions based on fixed symbols of an alphabet, for efficiency reasons, in our real implementation, we allow the states of the VPA to store values and also to transition when a predicate evaluates to true[15].

As described above, compiling XSeq queries into efficient query plans starts by constructing an equivalent VPA for the given query. We construct this VPA by an iterative bottom-up process where we start from a single-state (trivial) VPA and at each forward Step of the XSeq query, we compose the original VPA with a new VPA that is equivalent with the current Step. Next, we show how different forward axes can be mapped into equivalent VPAs. Lastly, we show some of the other constructs of the XSeq query that can be similarly handled.

In the following, whenever connecting the accepting state(s) of a VPA to the starting state(s) of the previous VPA, since VPAs are closed under concatenation, the resulting automaton is still a valid VPA.

Handling /: The /`X` axis is equivalent to a VPA with two states `E` and `O` where `E` is the starting state at which we invoke the stack on open and closed tags accordingly (see Appendix A for the rules regarding stack manipulation in a VPA), and transition to the same state on all input symbols as long as the consumed input in `E` is well-nested. Upon seeing the appropriate open tag (e.g., $\langle$`X`$\rangle$) we non-deterministically transition to our accepting state `O`.

Handling @: In the presence of the attribute specifier, @, we add a new state `A` as the new accepting state which will be transitioned to from our previous accepting state upon seeing any attribute. We remain in state `A` as long as the input is another attribute, i.e. to account for multiple attributes of the same open tag.

Fig. 5(a) demonstrates the VPA for /`son@Bdate`. Fig. 6 shows the intuitive correspondence of this VPA with the navigation of the XML document, where:

— `E` matches zero or more (well-nested) subtrees in the pre-order traversal of the XML tree,
— `O` matches the open tag for son, i.e. $\langle$`son`$\rangle$,
— `A` matches the attribute list of $\langle$`son`$\rangle$, namely `O`.

To see the correspondence between this VPA and the XSeq query, note that to find all the direct sons of a daughter, we navigate through the pre-order traversal of the subtree under each daughter node, then *non-deterministically* skip an arbitrary number

[15]However, in the formal analysis of XSeq's expressiveness in Section 6, we will use the theoretical notion of a VPA, i.e. without any storage besides the actual states and without any predicates.

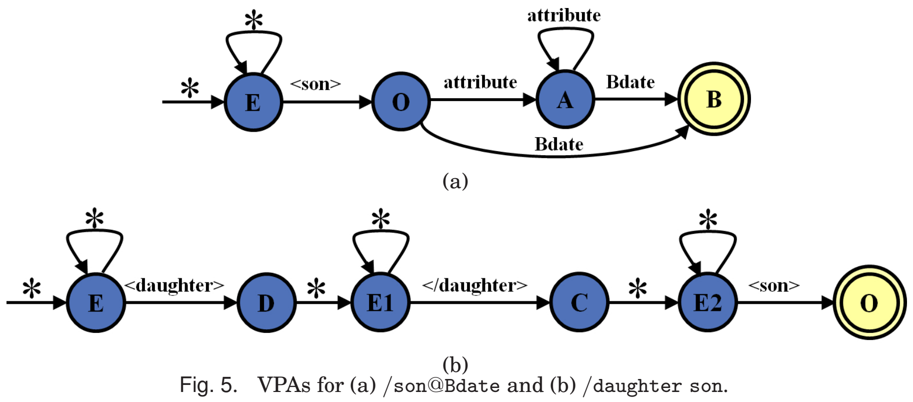


Fig. 5. VPAs for (a) /son@Bdate and (b) /daughter son.

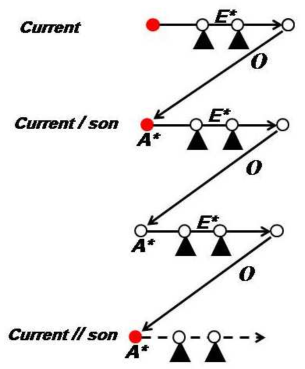


Fig. 6. Visual correspondence of VPA states and XSeq axes.

of her children (i.e., E*) until visiting one of her children who is a son (i.e., O), and then finally visit all the tokens that correspond to his son's attributes, i.e. A*. The non-determinism assures that we eventually visit all the sons under each daughter.

Handling ()*: Kleene-* expressions in XSeq, such as $(/\texttt{son})^*$, are handled by first constructing a VPA for the part inside the parentheses, say V_1, then adding an ϵ-transition from the accepting state of V_1 back to its starting state. Since VPAs are closed under Kleene-*, the resulting automaton will still be a VPA.

Handling //: The // axis can also be easily defined as a Kleene-* of the / operator. For instance, the //daughter construct is equivalent to $(/\texttt{X}) * /\texttt{daughter}$, where X is a wild card, i.e. matches any open tag. Fig. 6 shows the correspondence between the VPA states for // and the familiar traversal of the XML document.

Handling siblings: Let V_1 be the VPA that recognizes the query up to node D. The VPA for recognizing the sibling of D, say node S, is constructed by adding four new states (E1, C, E2 and O) to V_1, where:

— We transition from the accepting state(s) of V_1 to E1. E1 invokes the stack on open and closed tags accordingly, and transitions to itself on all input symbols as long as the consumed input in E1 is well-nested.
— Upon seeing a close tag of D, we non-deterministically transition from E1 to C.

— We transition from C to E2 upon any input. Similar to E1, E2 invokes the stack on open and closed tags accordingly, and transitions to itself on all input symbols as long as the consumed input in E2 is well-nested.
— Upon seeing an open tag for the sibling, i.e. $\langle \mathtt{S} \rangle$, we non-deterministically transition from E2 to state O which is marked as the accepting state of the new VPA.

Fig. 5(b) shows the VPA for query "/daughter son". The intuition behind this construction is that E1 skips all possible subtrees of the last daughter non-deterministically, while E2 non-deterministically skips all other siblings of the current daughter until it reaches its sibling of type son.

Handling \ : The construct \X is handled according to the last axis that has appeared before it. Let V_1 be the VPA for the XSeq query up to \X. When the previous axis is vertical (e.g. / or //), then we only need to add one new state to the V_1, say O, where from all the accepting states of V_1 we transition to state O upon seeing any open tag of X. The new accepting state will be O.

When the axis before \X is horizontal (e.g. siblings), we add three new states to V_1, say E, C and O, where:

— We transition from the accepting state(s) of V_1 to E. At E, we invoke the stack upon open and closed tags accordingly, and transition to E on all input symbols as long as the consumed input in E is well-nested.
— We non-deterministically transition from E to C upon seeing a close tag of the last (horizontal) axis.
— We transition from C to O upon an open tag for X and fail otherwise. O will be the new accepting state of the VPA.

Handling predicates. In general, arbitrary predicates cannot be handled using the inductive construction described in this section, e.g., when a predicate refers to nodes other than the one being processed. Thus, our construction in this section assumes that the predicates only refer to attributes of the current node. (In Section 6 we consider arbitrary predicates.)

In our real implementation of XSeq, we simply use a few variables (a.k.a. registers) at each state, in order to remember the latest values of the operands in the predicate(s) that need to be evaluated at that state. However, in our complexity analysis in Section 6, we use the abstract form of a VPA, namely where a state is duplicated as many as there are unique values for its operands.

Handling partition by. Since the pattern in the 'partition by' clause is the prefix of the pattern in the 'from' clause, the partition by clause can be simply treated as a new predicate on the attribute which is partitioned by. For example, when translating Query 17 into a VPA, assume that the 'partition by' attribute (i.e., ID) has k different values, i.e. $v_1, \cdots, v_k$. Then, we replicate the current VPA k times, each corresponding to a different value of the ID attribute. Once a value of ID is read, say v_i, we transition to the starting state of the VPA that corresponds to v_i and thereon, we simply check that at every state of that sub-automata the current value of the ID attribute is equal to v_i, i.e. otherwise we reject that run of the automata.

Handling other constructs Union, intersection, and, node tests can all be implemented with their corresponding operations on the intermediary VPAs, as VPAs are closed under union, intersection and complementation. The translations are thus straightforward (omitted here for space constraints).

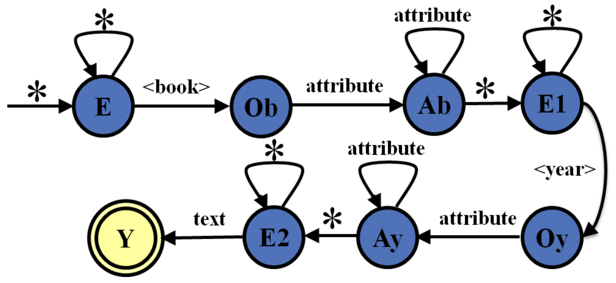


Fig. 7. $//\mathtt{book}/\mathtt{year}/\mathtt{text}()$

4.2. Static VPA Optimization

Cutting the inferrable prefix. When the schema (e.g. DTD) is available, we can always remove the longest prefix of the pattern as long as (i) the prefix has not been referenced in the `return` or the `where` clause, and (ii) the omitted prefix can be always inferred for the remaining suffix. For example, consider the following XSeq query, defined over the SigmodRecord dataset[16]:
$//\mathtt{issue}/\mathtt{articles}/\mathtt{authors}/\mathtt{author}[\mathtt{text}()=$'`Alan Turing`']
This XSeq query generates a VPA with many states, i.e. 3 states for every step. However, based on the DTD, we infer that `author` nodes always have the same prefix, i.e. $\mathtt{issue}/\mathtt{articles}/\mathtt{authors}/$. Thus, we remove the part of the VPA that corresponds to this common prefix. Due to the sequential nature of VPAs, such simplifications can greatly improve the efficiency by reducing a global pattern search to a more local one.

Reducing non-determinism from the transition table. Our algorithm for translating XSeq queries produces VPAs that are typically non-deterministic. Reducing the degree of non-determinism always improves the execution efficiency by avoiding many unnecessary backtracks. In general, *full determinization* of a VPA is an expensive process, which can increase the number of states from $O(n)$ to $O(2^{n^2})$ [Alur and Madhusudan 2004].

However, there are special cases that the degree of non-determinism can be reduced without incurring an exponential cost in memory. Since self-loops in the transition table are one of the main sources of non-determinism, whenever self-loops can only occur a fixed number of times, the XSeq's compile-time optimizer removes such edges from the generated VPA by duplicating their corresponding states accordingly. For instance, consider the XSeq query $//book/year/text()$ and its corresponding VPA in Fig. 7. If we know that `book` nodes only contain two subelements, say `title` followed by `year`, the optimizer will replace `E1` with 3 new states (without any self-loops) to explicitly skip the title's open, text and closed tags. The latter expression ($E1$^3) is executed more efficiently as it will be deterministic.

Reducing non-determinism from the states. In order to skip all the intermediate subelements, the automatically generated VPAs contain several states with incoming and outgoing ϵ-transitions. In the presence of the XML schema, many of such states become unnecessary and can be safely removed before evaluating the VPA on the input. We have several rules for such safe omissions. Here, we only provide one example.

Let us once again consider the query and the VPA of Fig. 7 as our example. If according to the schema, we know that the `year` nodes cannot contain any subelements,

[16]http://www.cs.washington.edu/research/xmldatasets/

the optimizer will remove E2 entirely. Also, if a node, say `year`, does not have any attributes, the optimizer will remove its corresponding state, here Ay.

4.3. Run-time VPA Optimization

In the previous sections, we demonstrated how XSeq queries can be translated into equivalent VPAs and presented several techniques for reducing the degree of non-determinism in our VPAs. One of the main advantages of using VPAs as the underlying execution model is that we can take advantage of the rich literature on efficient evaluation of VPAs. In particular we use the one-pass evaluation of the VPAs as described in [Madhusudan and Viswanathan 2009] and use the pattern matching optimization of VPAs as described in [Mozafari et al. 2010a].

In a straightforward evaluation of a VPA over a data stream, one would consider the prefix starting from every element of the stream as a new input to the VPA. In other words, upon acceptance or rejection of every input, the immediate next starting position would be considered. However, for word automata, it is well-known that this naive backtracking strategy can be easily avoided by applying pattern matching techniques such as the KMP [Knuth et al. 1977] algorithm. Recently, a similar pattern matching technique was developed for VPAs, known as VPSearch [Mozafari et al. 2010a]. Similar to word automata, VPSearch avoids many unnecessary backtracks and therefore, reduces the number of VPA evaluations. We have implemented VPSearch and its run-time caching techniques in our Java implementation of XSeq. Further details on streaming evaluation of VPAs and the VPSearch algorithm can be found in [Madhusudan and Viswanathan 2009] and [Mozafari et al. 2010a], respectively. Because of the excellent VPA execution performance achieved by K*SQL [Mozafari et al. 2010a], we have used the same run-time engine for XSeq queries once they are compiled into a VPA (see Section 7).

In the next two sections, we define the formal semantics of XSeq and present our results on its expressiveness and complexity.

5. FORMAL SEMANTICS OF XSEQ

While in the previous section we informally illustrated the semantics of different XSeq operators through intuitive examples, in this section we provide the formal semantics of XSeq which once restricted to its navigational features, will pave the way for a rigorous analysis of the language in Section 6. We first define an XML tree.

Definition 5.1 (*XML Tree*). An XML tree Tr is an unranked ordered tree $Tr = (V, L, \downarrow, \rightarrow)$ where V is a set of nodes, $L : V \rightarrow \Sigma$ is a labeling of the nodes to symbols of a finite alphabet Σ, and $R_{\downarrow}$ and $R_{\rightarrow}$ are respectively the parent-child and immediately following sibling relationships among the nodes. For leaf nodes v, we define $R_{\downarrow}(v) = \perp$. Also, for the rightmost child v we define $R_{\rightarrow}(v) = \perp$. We refer to the root node of Tr as $root(Tr)$.

Using $R_{\downarrow}$ and $R_{\rightarrow}$, we can similarly define R_{ax} where ax is any of the Axes[17] in Fig. 2. Next, we define a query, where for simplicity, we ignore the output clause and only consider the 'decision' version of the query, namely query can only return a 'true' if it finds a match, and otherwise returns nothing.

Definition 5.2 (*Query*). We represent an XSeq query of form "return true from doc() P where C" as $Q = (P, C)$ where P is a PathExpr and C is a Condition. When C is absent, we use "true" instead, i.e. $Q = (P, true)$.

[17]For clarity, in this section, we use capitalized words when referring to any of the production rules of Fig. 2.

Definition 5.3 (*Normalized Query*). A query $Q = (P, C)$ is normalized if all Predicates in P are patterns.

Note that we can always normalize any query $Q = (P, C)$ by applying the following steps:

(1) For each Condition Predicate cp in P, rewrite cp into disjunctive normal form $cp_1 \vee cp_2 \vee \cdots \vee cp_k$. Then rewrite Q into $Q' = (P_1 \cup P_2 \cup \cdots, P_k, C)$ such that each P_i only contains cp_i. Repeat this process until all Condition Predicates in P_i are in conjunctive form. Let the resulting query be $Q^0 = (P^0, C^0)$.
(2) For each conjunctive Condition Predicate $pred$ in Step s of P^0, extract all of its path expressions, say $p_1, p_2, ..., p_j$.
(3) By renaming the last Step in p_i with a new Step variable v_i, obtain p_i'. Add a Predicate $[p_i']$ to Step s.
(4) Remove $pred$ from s. Express $pred$ using $\{v_i\}$, say $pred'$. Add $pred' \wedge$ $\{$constraints on $\{v_i\}\}$ to C^0.
(5) If p_i contains Kleene-* and the last Step can be empty (e.g. due to a Kleene-*), rewrite p_i into the union of a set of path expression, whose last Step cannot be empty.

Thus, in the rest of this discussion, we assume all the queries are in their normalized form.

Example 5.4. The normalized form of the query $doc()/a[c/d > e/f]/b$ is as follows:

```
doc()/a[c/$X][e/$Y]/b
where tag($X) = 'd' and tag($Y) = 'f'
 and $X > $Y
```

Definition 5.5. For a Pattern p, we define $\chi(p)$ = {all the NameTest's that appear in p}.

When the same NameTest appears multiple times, we keep all occurrences in χ, e.g. by adding an index.

Example 5.6. For $p = doc()/a/b/\$X/a$, we have $\chi(p) = \{a_1, b, a_2, \$X\}$.

Definition 5.7 (*Base of a Predicate*). For any Step of the form "$ax :: nt\ [p]$" where ax is an Axis, nt is a NameTest and p is a Pattern, we define the base of $[p]$ as $\beta(p) = nt$.

Example 5.8. For the PathExpr $A/B[C[D]]$ we have $\beta(C[D]) = B$ and $\beta(D) = C$.

Definition 5.9 (*Flattening a Pattern*). For a Pattern P, $\tilde{P}$ is the result of removing all the Predicates from P.

Example 5.10. Consider $p = A/B[C[D]]$. Then, $\tilde{p} = A/B$. Thus, $\chi(p)$ may be different from $\chi(\tilde{p})$.

Definition 5.11 (*Meaning of a Pattern*). Given an XML tree Tr, for any Pattern P we define its meaning, denoted as $[[p]]^{Tr}$, recursively, as follows [18] where $[[p]]^{Tr} \subseteq (N \times (\chi(P) \cup \{\perp\}))^*$:

— If $P = doc()\ p$, define
$[[doc()\ p]] = \{\langle(root(Tr) : \perp), (n_1 : L_1), \cdots (n_k : L_k)\rangle \in [[p]]\}$
— If $P = p$ where p is a PathExpr, define
$[[p]] = \{\langle(n_1 : L_1), \cdots, (n_k : L_k)\rangle \mid n_i \in N, L_i \in \chi(\tilde{p}) \cup \{\perp\}\}$

[18]For brevity, we assume the tree is fixed and thus, denote $[[p]]^{Tr}$ as $[[p]]$.

— If P is a single Step of form $nt :: ax$ where nt and ax are the NameTest and Axis, respectively, define
$[[nt :: ax]] = \{\langle(n : \bot), (m : nt)\rangle \mid n, m \in N, L(m) = nt$ and $(n, m) \in R_{ax}\}$
— If P is a single Step of form $nt :: ax[p]$ where nt, ax, and p are the NameTest, Axis, and Pattern[19] respectively, define
$[[nt :: ax[p]]] = \{\langle(n0 : \bot), (n : l)\rangle\rangle \mid \langle(n_0 : \bot), (n : l)\rangle \in [[nt :: ax]], \exists\alpha$ s.t. $\langle(n : \bot), \alpha\rangle \in [[p]]\}$
— If P is a path variable v with the definition $(v : p)$ define $[[v]] = [[p]]$
— If $P = p_1p_2$, define
$[[p_1p_2]] = \{\langle\alpha_1, (n : l), \alpha_2\rangle \mid \exists n, l$ s.t. $\langle\alpha_1, (n : l)\rangle \in [[p_1]]$ and $\langle(n : \bot), \alpha_2\rangle \in [[p_2]]\}$
— If $P = (p)*$, define
$[[(p)*]] = \{\langle(n_0 : l_0), \alpha_1, (n_1 : l_1), \alpha_2, (n_2 : l_2), \cdots, (n_{k-1} : l_{k-1}), \alpha_k, (n_k : l_k)\rangle \mid \langle(n_{i-1} : \bot), \alpha_i, (n_i : l_i)\rangle \in [[p]]$ for $i = 1, \cdots, k\} \cup \{\epsilon\}$

Example 5.12. Consider an XML tree $Tr = (V, L, \downarrow, \rightarrow)$, where $V = \{a_1, b_1, b_2, c_1\}$, $L = \{(a_1, a), (b_1, b), (b_2, b), (c_1, c)\}$, $R_\downarrow = \{(a_1, b_1), (a_1, b_2), (b_1, c_2)\}$, and $R_\rightarrow = \{(b_1, b_2)\}$. Here, even though:
$[[doc()/a/b]] = \{\langle(root(Tr) : \bot), (a_1 : a), (b_1 : b)\rangle, \langle(root(Tr) : \bot), (a_1 : a), (b_2 : b)\rangle\}$,
$[[doc()/a/b[/c]]]$ contains only one sequence, i.e.,
$[[doc()/a/b[/c]]] = \{\langle(root(Tr) : \bot), (a_1 : a), (b_1 : b)\rangle\}$.
$\langle(root(Tr) : \bot), (a1 : a), (b2 : b)\rangle$ is not in $[[doc()/a/b[/c]]]$ because there is no sequence starting with $(b_2 : \bot)$ in $[[/c]] = \{\langle(b_1 : \bot)(c_1 : c)\rangle\}$.

Definition 5.13 (***Environment***). An environment is any mapping $e_{P,\alpha,n} : \chi(P) \rightarrow N \cup \{\bot\}$ where P is a Pattern, $\alpha \in [[p]]$, and $n \in N \cup \{\bot\}$.

Definition 5.14 (*Valid Environment*). An environment $e_{P,\alpha,n}$ is valid iff one of the following conditions holds:

(1) P is a PathExpr, $P = \tilde{P}$, and for all $l \in \chi(P)$ we have $e_{P,\alpha,n}(l) = n'$ if there exists n' such that $\alpha = \langle(n : \bot), \cdots, (n' : l), \cdots\rangle$
(2) P is a PathExpr with top-level Predicates[20] $p_1, \cdots, p_k$, and there exist $\alpha_i \in [[p_i]]$ for $1 \leq i \leq k$ such that $e_{P,\alpha,n} = e_{\tilde{P},\alpha,n} \bigcup \cup_{i=1}^{k} e_{p_i,\alpha_i,e_{\tilde{P},n}(\beta(p_i))}$ where $e_{\tilde{P},\alpha,n}$ and $e_{p_i,\alpha_i,e_{\tilde{P},n}(\beta(p_i))}$ are also valid environments.
(3) $P = doc()\ p$ where p is a PathExpr, and there exist $\alpha' \in [[p]]$ such that $e_{p,\alpha',root(Tr)}$ is a valid environment.

Example 5.15. Given the XML tree defined in Example 5.12, consider Pattern $P = doc()/a/\$X[/c]$. The environment $e_{P,\alpha,root(Tr)}$ is valid if $\alpha = \{\langle(root(Tr) : \bot), (a_1 : a), (b_1 : \$X)\rangle\}$, and $e_{P,\alpha,root(Tr)}(\$X) = b_1$.

Definition 5.16 (*Condition Evaluation Under A Valid Environment*). We define when a Condition C evaluates to true under a valid environment e (which we denote as $e \models C$) by defining how to replace different types of Operand with constant values. Once all the Operands in C are replaced with their constant values, the entire Condition can be also evaluated by following the conventional rules of arithmetic and boolean expression. There are different types of Operands:

— Constant is trivial.
— $seq(X)@attr$, where $\alpha = \langle(n_1, l_1), ..., (n_i, l_i), ...(n_m, l_m)\rangle$ and $e_{P,\alpha,n}(X) = n_i$, is replaced with attribute 'attr' of node $n_j, 1 \leq j \leq m$ where $l_j = X$ and :

[19]Note that since the query is normalized, here we do not need to consider Conditions as Predicate.
[20]For instance, for $p = A/B[C[D]][E]/T[H]$ the top-level Predicates are $p_1 = C[D]$ and $p_2 = E$, and $p_3 = H$.

$$\begin{array}{rcl}
\text{PathExpr} & ::= & \text{PathExpr}\ '*' \ \mid \text{PathExpr}\ '\mathit{intersect}'\ \text{PathExpr} \mid \text{VStep} \\
\text{VStep} & ::= & \text{Step}\ \text{Variable}* \\
\text{Step} & ::= & \text{Axis}\ '::'\ \text{NameTest}\ [\text{Variable}] \\
 & \mid & \text{Axis}\ '::'\ \text{NameTest} \\
\text{Axis} & ::= & \odot \mid \downarrow \mid \uparrow \mid \rightarrow \mid \leftarrow
\end{array}$$

Fig. 8. CXSeq Syntax

— if seq=prev and for $j + 1 \leq k \leq i - 1$, we have $l_k \neq X$;
— if seq=first, and for $1 \leq k \leq j - 1$, we have $l_k \neq X$;
— if seq=last, and for $j + 1 \leq k \leq m$, we have $l_k \neq X$;
Otherwise, we replace it with the null value.

— $X@text()$ is replaced with the text value of node n_j where n_j is defined as above.
— $agg(X@attr)$, where X is in $\tilde{P}$, a valid environment $e_{P,\alpha,n}$ is picked and $\alpha = \langle(n_1, l_1), ..., (n_i, l_i), ...(n_m, l_m)\rangle$, is replaced with $agg(\{n_i | l_i = X, 1 \leq i \leq m\})$.

Definition 5.17 (***Query Evaluation***). We say that a query $Q = (P, C)$ recognizes the XML tree Tr, iff $[[doc()P]] \neq \emptyset$ and for all valid environments $e_{P,root(Tr)}$, $e_{P,root(Tr)} \models C$.

6. EXPRESSIVENESS AND COMPLEXITY

In Section 4, we provided the high-level idea of how most of XSeq queries can be optimized and translated into equivalent VPAs. In this section, we provide our results on the expressiveness of XSeq, and its complexity for query evaluation —two fundamental questions for any query language.

Throughout this section, Σ is the alphabet (i.e., set of unique tokens in the XML document), and MSO is monadic second order logic over trees.

The full language of XSeq is too rich for a rigorous logical analysis, and thus we focus on its navigational features by excluding arithmetics, string manipulations and aggregates. Thus, in Section 6.1, we first obtain a more concise language, called CXSeq[21].

We show that, given a CXSeq query Q, the set of input trees for which Q contains a match (we call this the domain of Q) is an MSO definable language. Conversely, for every MSO definable language L, there exists a CXSeq with domain L. The proof of this statement can be found in Appendix B.

In Section 6.3, we use this equivalence result to derive the complexity of query evaluation of CXSeq queries.

6.1. CXSeq

In Fig. 8, we have provided the syntax of the query language CXSeq.

A query is a tuple $K = (V, v_0, \rho)$ where V is a finite set of variables, $v_0 \in V$ is the starting variable, and $\rho : V \times P$ is a set of productions where P is the set of elements defined by the grammar in Fig. 8 starting with $PathExpr$. In the grammar a Variable is an element of V.

The semantics of a query is given with pairs of nodes and is parameterized over variables. The idea is that every XPath query that can be "generated" by the above grammar should be in some sense executed. Consider the query:

$$(X, \uparrow :: aX), (X, \uparrow :: a)$$

Its semantics should be equivalent to $(\uparrow :: a)+$.

[21]Similar approaches in analyzing XPath 1.0 and 2.0, has led to sub-languages Core XPath 1.0[ten Cate and Marx 2007b] and Core XPath 2.0[ten Cate and Marx 2007a].

Given a variable $v \in V$ we define $S(v) \subseteq N \times N$ as the set of pairs of nodes that satisfy the query v. Informally $S(v)(n, n')$ iff starting in node n we can reach node n' following the query v. Informally every variable v defines a set of pair, but the final result of K is the set of pairs assigned to v_0. We can now proceed inductively and define the semantics as the least fix point of the following relations. $S_{v,\pi} \subseteq N \times N$ (for each v and π) defines the pair of nodes belonging to v when starting with the production π.

(1) if $\pi = \uparrow :: s[v_1]v_2$, $S_{v_1}(x, y_1)$, $S_{v_2}(x, y_2)$, $\mathrm{lab}(x) = s$, and $R_\downarrow(x, z)$, then $S_{v,\pi}(z, y_2)$;
(2) if $\pi = \downarrow :: s[v_1]v_2$, $S_{v_1}(x, y_1)$, $S_{v_2}(x, y_2)$, $\mathrm{lab}(x) = s$, and $R_\downarrow(z, x)$ and does not exists z', $R_\rightarrow(z', x)$, then $S_{v,\pi}(z, y_2)$;
(3) if $\pi = \leftarrow :: s[v_1]v_2$, $S_{v_1}(x, y_1)$, $S_{v_2}(x, y_2)$, $\mathrm{lab}(x) = s$, and $R_\rightarrow(x, z)$, then $S_{v,\pi}(z, y_2)$;
(4) if $\pi = \rightarrow :: s[v_1]v_2$, $S_{v_1}(x, y_1)$, $S_{v_2}(x, y_2)$, $\mathrm{lab}(x) = s$, and $R_\rightarrow(z, x)$, then $S_{v,\pi}(z, y_2)$;
(5) if $\pi = \odot :: s[v_1]v_2$, $S_{v_1}(x, y_1)$, $S_{v_2}(x, y_2)$, and $\mathrm{lab}(x) = s$, then $S_{v,\pi}(x, y_2)$;
(6) if $\pi = \pi_1 \cap \pi_2$, $S_{v,\pi} = S_{v,\pi_1} \cap S_{v,\pi_2}$;
(7) the other cases are analogous.

Finally $S_v = \bigcup_{(v,\pi)\in\rho} S_{v,\pi}$.

This language allows us to define productions of the form

$$X := \downarrow :: aYZ$$

Without further restrictions this extension would be too expressive. For example the productions

$$X := \downarrow :: aXY, Y := \downarrow :: b$$

would represent the query $(a/)^n(b/)^n$ which is not MSO expressible. In order to reduce the expressiveness we limit the use of recursion. Given a production $p = (v, \pi)$ let $\mathrm{dv}(\pi)$ be the following set of variables:

— $\mathrm{dv}(\pi \cap \pi') = \mathrm{dv}(\pi) \cup \mathrm{dv}(\pi')$;
— $\mathrm{dv}(\pi*) = \mathrm{va}(\pi)$;
— $\mathrm{dv}(d :: a[v]v_1 \ldots v_{n+1}) = \{v_1, \ldots, v_n\}$;
— $\mathrm{dv}(d :: a[v]) = \{\}$;
— $\mathrm{dv}(d :: av_1 \ldots v_{n+1}) = \{v_1, \ldots, v_n\}$;
— $\mathrm{dv}(d :: a) = \{\}$.

Similarly we define for a production $p = (v, \pi)$ the set $\mathrm{va}(\pi)$ be the following set of variables:

— $\mathrm{va}(\pi \cap \pi') = \mathrm{dv}(\pi) \cup \mathrm{dv}(\pi')$;
— $\mathrm{va}(\pi*) = \mathrm{va}(\pi)$;
— $\mathrm{va}(d :: a[v]v_1 \ldots v_n) = \{v_1, \ldots, v_n\}$;
— $\mathrm{va}(d :: a[v]) = \{v\}$;
— $\mathrm{va}(d :: av_1 \ldots v_n) = \{v, v_1, \ldots, v_n\}$;
— $\mathrm{va}(d :: a) = \{\}$.

Now given a variable $v \in V$ we define the sets of variables reachable from v as yi_v as the set satisfying the following equation

$$\mathrm{yi}_v = \{v\} \cup \bigcup_{(v,\pi)\in\rho} \bigcup_{v'\in\mathrm{va}(\pi)} \mathrm{yi}_{v'}$$

We can now formalize the restriction on our grammar.

Definition 6.1. A query $K = (V, v_0, \rho)$ is *safe* iff for each $(v, \pi) \in \rho$, for each $v' \in \mathrm{dv}(\pi)$, $v \notin \mathrm{yi}(v')$.

For the rest of the presentation, we will only consider safe CXSeq queries.

Notice that the $\downarrow$ axis has the meaning of first child of a node instead of child. This language also allows to define the operators $axis*$ using the $*$ operator. We can also extend the language to allow nested stars, and the same translation as before will work. For example the production $(v, ((\uparrow\colon a)v' * v'')*)$ can be transformed into the following set of productions

$$(v, (\odot : \_)t_1); (t_1, (\uparrow\colon a)t_2t_1); (t_1, (\odot :: \_)); (t_2, (\odot : \_)t_3v''); (t_3, (\odot : \_)v't_3); (t_3, \odot : \_)$$

where t_1, t_2, t_3 are fresh names.

To better understand the semantics let's consider the following regular XPath query:

$$\downarrow:: a \downarrow:: b(\downarrow:: c)*$$

This will be encoded in CXSeq with the following productions:

$$(X, \downarrow:: aYZ), (Y, \downarrow:: b), (Z, \odot : \_WZ), (Z, \odot : \_), (W, \downarrow\colon c)$$

where X is the first production.

6.2. Regularity of CXSeq and Complexity

This section contains the two main results on CXSeq. Given a query K we define its domain as $D_K = \{w | K(w) \neq \emptyset\}$. CXSeq is equivalent to MSO in terms of domain expressiveness and therefore the domain of every CXSeq query can be translated into a VPA.

THEOREM 6.2. *For every CXSeq query $K = (V, v_0, \rho)$, the the domain D_K of K is an MSO definable language. Conversely for every MSO definable language L, there exists a CXSeq query K such that $L = D_K$.*

THEOREM 6.3. *For every CXSeq query $K = (V, v_0, \rho)$, there exists an equivalent VPA A over Σ such that $L(A) = D_K$. A will have $O(r^5 \cdot length(K)^5 \cdot 2^{r \cdot length(K)})$ where $r = |\rho|$.*

The proofs of the above theorems can be found in Appendix B.

6.3. Query Evaluation Complexity

LEMMA 6.4 (QUERY EVALUATION). *Data and query complexities for CXSeq's query evaluation are* PTIME *and* EXPTIME*, respectively.*

PROOF. By mapping CXSeq queries into VPAs, the query evaluation of the former corresponds to the language membership decision of the latter. Using the membership algorithm provided in [Madhusudan and Viswanathan 2009], we only need space O($s^4 \cdot log\ s \cdot d + s^4 \cdot n \cdot log\ n$) where n is the length of the input, d is the depth of the XML document (thus, $d < n$), and s is the number of the states in the VPA. PTIME data complexity comes from n and the EXPTIME query complexity comes from s which is exponential in the query size (see Theorem 6.3). □

We conclude this section with a result on containment of query domains.

LEMMA 6.5 (QUERY DOMAIN CONTAINMENT). *Given two CXSeq queries K_1 and K_2, it is decidable to check whether the domain of K_1 is contained in the domain of K_2. Moreover, the problem is* 2-EXPTIME*-complete.*

PROOF. Once two CXSeq queries are translated into VPAs, their query domain containment problem corresponds to the language inclusion problem for their domain VPAs, say M_1 and M_2. To check $L(M_1) \subseteq L(M_2)$, we check if $L(M_1) \cap \overline{L(M_2)} = \emptyset$. Given M_1 with s_1 states and M_2 with s_2 states, we can determinize [Tang 2009] and complement the latter to get a VPA for $\overline{L(M_2)}$ of size O($2^{{s_2}^2}$). $L(M_1) \cap \overline{L(M_2)}$ is then of

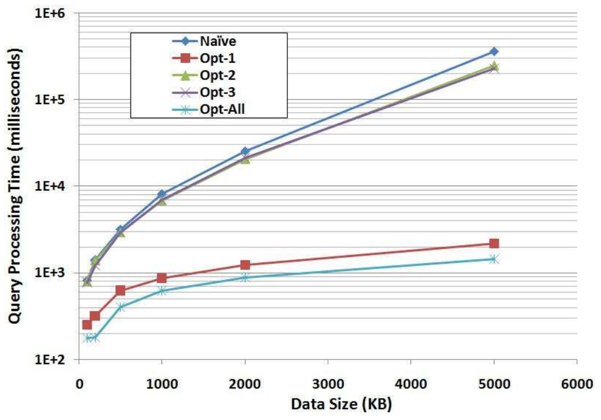


Fig. 9. Contribution of different optimization techniques.

size O($s_1 \cdot 2^{{s_2}^2}$), and emptiness check is polynomial (cubic) in the size of this automaton. Since, s_1 and s_2 are themselves exponential in the size of their CXSeq queries, membership in 2-EXPTIME holds. For completeness of the 2-EXPTIME, note that CXSeq syntactically subsumes Regular XPath$(*, \cap)$ for which the query containment has been shown to be 2-EXPTIME-complete [Cate and Lutz 2009]. □

7. EXPERIMENTS

In this section we study the amenability of XSeq language to efficient execution. Our implementation of the XSeq language consists of a parser, VPA generator, a compile-time optimizer, and the VPA evaluation and optimization run-time, all coded in Java. We first evaluate the effectiveness of our different compile-time optimization heuristics in isolation. We then compare our XSeq system with the state-of-the-art XML engines for (i) complex sequence queries, (ii) Regular XPath queries, and (iii) simple XPath queries. While these systems are designed for general XML applications, we show that XSeq is far more suited for CEP applications. In fact, XSeq achieves up to two orders of magnitude out-performance on (i) and (ii), and competitive performance on (iii). Finally, we study the overall performance, throughput and memory usage of our system under different classes of patterns and queries.

All the experiments were conducted on a 1.6GHz Intel Quad-Core Xeon E5310 Processor running Ubuntu 6.06, with 4GB of RAM. We have used several real-world datasets including NASDAQ stocks that contains more than $7.6M$ records[22] since 1970, and also the Treebank dataset[23] that contains English sentences from Wall Street Journal and has with a deep recursive structure (max-depth of 36 and avg-depth of 8). We have also used XMark [Schmidt and et. al. 2002] which is well-known benchmark for XML systems and provides both data and queries. Due to lack of space, for each experiment we only report the results on one dataset. The results and main observations, however, were similar across different datasets.

7.1. Effectiveness of Different Optimizations

In this section, we evaluate the effectiveness of the different compile-time optimizations from Section 4.2, by measuring their individual contribution to the overall perfor-

[22] `http://infochimps.org/dataset/stocks_yahoo_NASDAQ`
[23] `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`

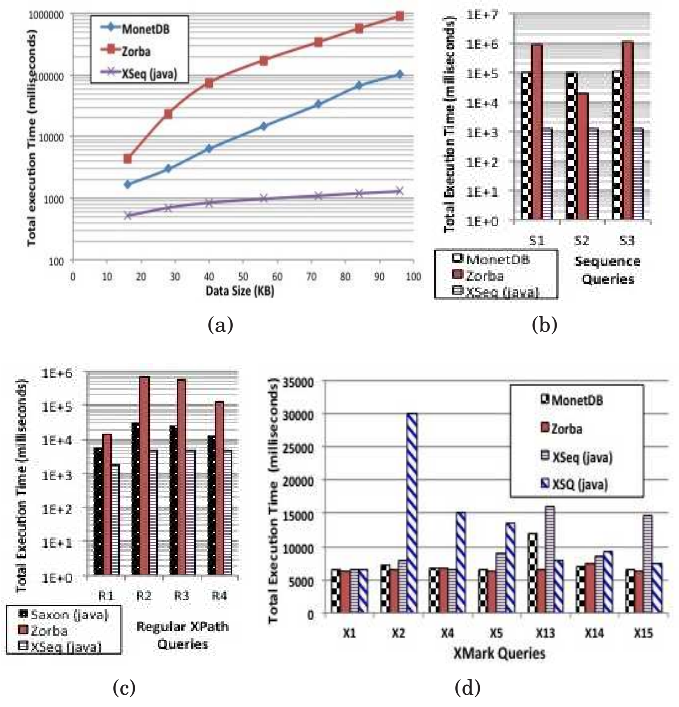


Fig. 10. XSeq vs. XPath/XQuery engines: (a) 'V'-pattern query over Nasdaq stocks, (b) Sequence queries over Nasdaq stocks, (c) Regular XPath queries over XMark data, and (d) conventional XPath queries from XMark.

mance[24]. For this purpose, we executed the X2 query from XMark [Schmidt and et. al. 2002] over a wide range of input sizes (generated by XMark, from 50KB to 5MB). The results of this experiment are reported in Fig. 9, where we use the following acronyms to refer to different optimization heuristics (see Section 4.2):

Opt-1	Cutting the inferrable prefix
Opt-2	Reducing non-determinism from the pattern clause
Opt-3	Reducing non-determinism from the where clause

In this graph, we have also included the naive and combined (Opt-All) versions, namely when, respectively, none and all of the compile-time optimizations are applied. The first observation is that combining all the optimization techniques delivers a dramatic improvement in performance (1-2 orders of magnitude, over the naive one).

Cutting the inferable prefix, Opt-1, leads to fewer states in the final VPA. Like other types of automata, fewer states can significantly reduce the overall degree of non-determinism. The second reason behind the key role of Opt-1 in the overall performance is that it reduces non-determinism from the *beginning* of the pattern: this is particularly important because non-determinism in the starting states of a VPA is usually disastrous as it prevents the VPA from the early detection of unpromising traces of the input. In contrary, reducing non-determinism in the pattern and the `where` clause (Opt-2, Opt-3) has a much more local effect. In other words, the latter techniques only

[24]The effectiveness of the VPA evaluation and optimization techniques have been previously validated in their respective papers [Madhusudan and Viswanathan 2009; Mozafari et al. 2010a].

remove the non-determinism from a single state or edge in the automata, while the rest of the automata may still suffer from non-determinism. However local, Opt-2 and Opt-3 can still improve the overall performance when combined with Opt-1. This is because of the extra information that they learn from the DTD file.

7.2. Sequence Queries vs. XPath Engines

We compare our system against two[25] of the fastest academic and industrial engines: MonetDB/XQuery[Boncz and et. al. 2006] and Zorba [Bamford and et. al. 2009]. First, we used several sequence queries on Nasdaq transactions (embedded in XML tags), including the 'V'-shape pattern (defined in Example 3.2 and Query 10). By searching for half of a 'V' pattern, we defined another query to find 'decreasing stocks'. Also, by defining two occurrences of a 'V' pattern, we defined what is known as the 'W'-shape pattern [26]. We refer to these queries as S1, S2 and S3. We also defined several Regular XPath queries over the treebank dataset, named R1, R2, R3 and R4 where,
R1: /FILE/EMPTY(/VP)*/NP,
R2: /FILE(/EMPTY)*/S,
R3: /FILE(/EMPTY)*(/S)*/VP,
R4: /FILE(/EMPTY)*/S(/VP)*/NP

Sequence queries. For expressing these queries (namely S1, S2 and S3) in XQuery, we had to mimic the notion of 'immediately following sibling', i.e. by checking that for each pair of siblings in the sequence, there are no other nodes in between. The XQuery versions of S2 has been given in Fig. 1. Due to the similarity of S1 and S3 to S2 here we omit their XQuery version (roughly speaking, S1 and S3 consist of, respectively, two and four repetitions of S2).

Not only were sequence queries difficult to express in XPath/ XQuery but were also extremely inefficient to run. For instance, for the queries at hand, neither of Zorba or MonetDB could handle any input data larger than 7KB. The processing times of these sequence queries, over an input size of 7KB, are reported in Fig. 10(b). Note that the Y-axis is in log-scale: *the same sequence queries written in XSeq run between 1-3 orders of magnitude faster than their XPath/XQuery counterparts do on two of the fastest XML engines*. Fig. 10(a) shows that gap between XSeq and the other two engines grows with the input size. This is due to the linear-time query processing of XSeq which, in turn, is due to the linear-time algorithm for evaluation of VPAs along with the backtracking optimizations when the VPA rejects an input [Mozafari et al. 2010a]. Zorba and MonetDB's processing time for these sequence queries are at least quadratic, due to the nested nature of the queries.

In summary, the optimized XSeq queries run significantly (1-3 orders of magnitude) faster than their equivalent counterparts that are expressed in XQuery. This result indicates that traditional XML languages such as XPath and XQuery (although theoretically expressive enough), due to their lack of explicit constructs for sequencing, are not amenable to effective optimization of complex queries that involve repetition, sequencing, Kleene-*, etc.

Regular XPath queries. As mentioned in Section 1, despite the many benefits and applications of Regular XPath, currently there are no implementations for this language (to our best knowledge). One of the advantages of XSeq is that it can be also seen as the first implementation of Regular XPath, as the latter is a subset of the former. In order to study the performance of XSeq for Regular XPath queries (e.g., R1, $\cdots$, R4) we compared our system with the only other alternative, namely implementing

[25]Since the sequence queries of this experiment are not expressible in XPath, we could not use the XSQ [Peng and Chawathe 2003] engine as it does not supports XQuery.
[26]'W'-pattern (a.k.a. double-bottom) is a well-known query in stock analysis.

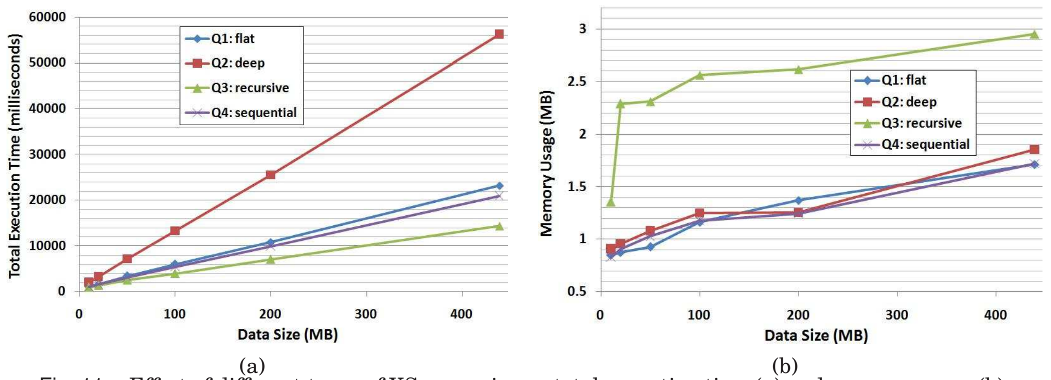


Fig. 11. Effect of different types of XSeq queries on total execution time (a) and memory usage (b).

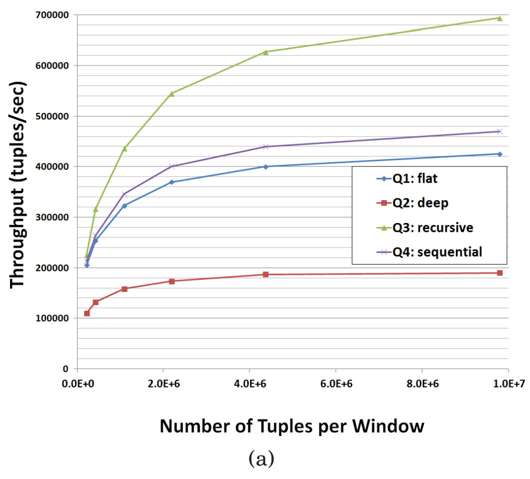
the Kleene-* operator as a higher-order user-defined functions (UDF) in XQuery. Since MonetDB does not support such UDFs, we used another engine, namely Saxon [Kay 2008]. The results for 464KB of treebank dataset are presented in Fig. 10(c) as Zorba, again, could not handle larger input size. Thus, for Regular XPath queries, similarly to sequence queries, XSeq proves to be 1-2 orders of magnitude faster than Zorba, and between 2-6 times faster than Saxon. Also, note that the relative advantage of Saxon over Zorba is only due to the fact that Saxon loads the entire input file in memory and then performs an in-memory processing of the query [Kay 2008]. However, this approach is not feasible for streaming or large XML documents[27].

7.3. Conventional Queries vs. XPath Engines

As shown in the previous section, complex sequence queries written in XSeq can be executed dramatically faster (from 0.5 to 3 orders of magnitude) than even the fastest of XPath/ XQuery engines. In this section, we continue our comparison of XSeq and native XPath engines by considering simpler XPath queries, i.e. queries without sequencing and Kleene-*. For this purpose, we used the XMark queries which in Fig. 10(d) are referred to as X1, X2, and so on[28]. Once again, we executed these queries on MonetDB, Zorba (as state-of-the-art XPath/XQuery engines) and XSQ (as state-of-the-art streaming XPath engine) as well as on our XSeq engine. In this experiment, the XMark data size was 57MB. Note that both Zorba and MonetDB are implemented in C/C++ while XSeq is coded in Java, which generally accounts for an overhead factor of 2X in a fair comparison with C/C++ implementations. The results are summarized in Fig. 10(d). The XSeq queries were consistently competitive compared to all the three state-of-the-art XPath/XQuery engines. XSeq is faster than XSQ for most of the tested queries. For some queries, e.g. X2 and X4, XSeq is even 2-4 times faster. Even compared with MonetDB and Zorba, XSeq is giving surprisingly competitive performance, and for some queries, e.g. X4, were even faster. Given that XSeq is coded in Java, this is an outstanding result for XSeq. For instance, once the java factor is taken into account, the only XMark query that runs slower on the XSeq engine is X15, while the rest of the queries will be considered about 2X faster than both MonetDB and Zorba.

In summary, once the maturity of the research on XPath/ XQuery optimization is taken into account, our natural extension of XPath that relies on a simple VPA-based

[27]Due to lack of space, we omit the results for the case when the input size cannot fit in the memory. Briefly, unlike XSeq, Saxon results in using the disk swap, and thus, suffers from a poor performance.
[28]Due to space limit and similarity of the result , here we only report 7 out of the 20 XMark queries.



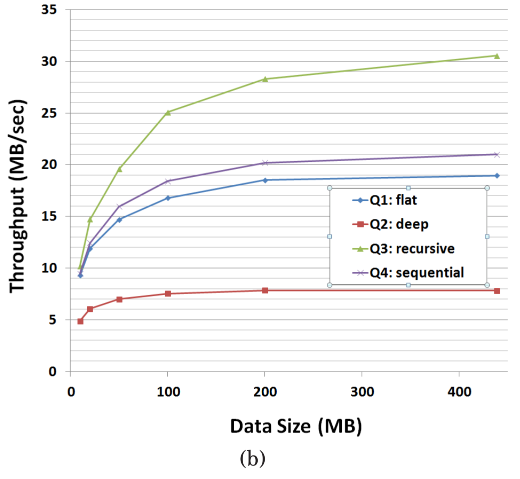


Fig. 12. The effect of different types of queries on (a) Total query execution time, (b) Throughput in terms of tuple processing, and (c) Throughput in terms of datasize.

optimization seems very promising: XSeq achieves better or comparable performance on simple queries, and is dramatically faster for more involved queries.

7.4. Throughput for Different Types of Queries

To study the performance of different types of queries in XSeq, we selected four representative queries with different characteristics which, based on our experiments, covered a wide range of different classes of XML queries. To facilitate the discussion, below we label the XML patterns as 'flat', 'deep', 'recursive' and 'monotone':

Q1: flat	/site/people/person[@id = 'person0']/name/text()
Q2: deep	/site/closed_auctions/closed_auction/annotation/ description/parlist/listitem/parlist/listitem/text/ emph/keyword/text()
Q3: recursive	(parlist/listitem)*
Q4: monotonic	//closed_auctions/ (\X[tag(X)='closed_auction' and X@price < prev(X)@price])*

We executed all these queries on XMark's dataset. Also, the first two queries (Q1 and Q2) are directly from XMark benchmark (referred to as Q1 and Q15 in [Schmidt and et. al. 2002]). We refer to them as 'flat' and 'deep' queries, respectively, due to their few and many axes. In XMark's dataset, the `parlist` and `listitem` nodes can contain one another, which when combined with the Kleene-*, is the reason why we have named Q3 'recursive'. The Q4 query, called 'monotonic', searches for all sequences of consecutive closed auctions where the price is strictly decreasing. These queries reveal interesting facts about the nature of XSeq language and provide insight on the types of XSeq queries that are more amenable to efficient execution under the VPA optimizations.

The query processing time is reported in Fig. 11(a). The first important observation is that XSeq has allowed for linear scalability in terms of processing time, regardless of the query type. This has enabled our XSeq engine to steadily maintain an impressive throughput of 200,000-700,000 tuples/sec, or equivalently, 8-31 MB/sec even when facing an input size of 450MB. This is shown in Fig. 12(a) and 12(b) in which the X-axes are drawn in log-scale. Interestingly, the throughput gradually improves when the window size grows from 200K to 1.1M tuples. This is mainly due to the amortized cost of VPA construction and compilation, and other run-time optimizations such as backtrack matrices [Mozafari et al. 2010a] that need to be calculated only once.

Among these queries, the best performance is delivered for Q3 and Q4. This is because they consist of only two XPath steps, and therefore, once translated into VPA, result in fewer states. Q1 comes next, as it contains more steps and thus, a longer pattern clause. Q2 achieves the worst performance. This is again expected, because Q2's deep structure contains many tag names which lead to more states in the final VPA. In summary, this experiment shows that with the help of the compile-time and run-time optimizations, XSeq queries enjoy a linear-time processing. Moreover, the fewer axes (i.e. steps) involved in the query, the better the performance.

8. PREVIOUS WORK

XML Engines. Given the large amount of previous work on supporting XPath/XQuery on stored and streaming data, we only provide a short and incomplete overview, focusing on the streaming ones. Several XPath streaming engines have been proposed over the years, including TwigM [Chen et al. 2006], XSQ [Peng and Chawathe 2003], and SPEX [Olteanu et al. 2003]; also the processing of regular expressions, which are similar to the XPath queries of XSQ, is discussed in [Olteanu et al. 2003] and [Barton and et. al. 2003]. XAOS [Barton and et. al. 2003] is an XPath processor for XML streams that also supports reverse axes (parent and ancestor), while support for predicates and wildcards is discussed in [Josifovski et al. 2005]. Finally, support for XQuery queries on very small XML messages (<100KB) is discussed in [Florescu and et. al. 2003].

Language extensions. Extending the expressive power of XPath has been the focus of much research [ten Cate 2006; ten Cate and Marx 2007b; 2007a; ten Cate and Segoufin 2008; Marx 2005]. For instance, *Core XPath 2.0* [ten Cate and Marx 2007a], extended Core XPath 1.0 with path intersection, complementation, and quantified variables. *Conditional XPath* [Marx 2005], extended XPath with 'until' operators, while the inclusion of a least fixed point operator was proposed in [ten Cate 2006]. More modest extensions, that better preserved the intuitive clarity and simplicity of Core XPath 1.0, included *Regular XPath* [ten Cate 2006], *Regular XPath*$^{\approx}$ [ten Cate and Marx 2007b] and *Regular XPath(W)* [ten Cate and Segoufin 2008]. These allowed expressions such as $/\texttt{a}(/\texttt{b}/\texttt{c})^{*}/\texttt{d}$, where a Kleene-* expression A^{*}, was defined as the infinite union $\cdot \cup A \cup (A/A) \cup (A/A/A) \cup \cdots$ Even for these more modest extensions, however, efficient implementation remained an issue: in 2006, the following open problem was declared as a challenge for the field [ten Cate 2006]: *Efficient algorithms for computing the transitive closure of XPath path expressions.*

VPA. Visibly Pushdown Automata (VPA) have been recently proposed for checking Monadic Second Order (MSO) formulas over dual-structured data such as XML [Alur and Madhusudan 2004; 2006], and have led to new streaming algorithms for XML processing [Madhusudan and Viswanathan 2009; Pitcher 2005]. The recently proposed query language K*SQL [Mozafari et al. 2010b; 2010a] used VPAs to achieve good performance and expressivity levels needed to query both relational and XML streams. However, while very natural for relational data, K*SQL is quite procedural and verbose for XML, whereby the equivalents of simple XPath queries are long and complex K*SQL statements. At the VPA implementation level, however, the same VPA optimization techniques support both XSeq and K*SQL.

Gauwin and Niehren provided translations for a streamable fragment of forward XPath into nested word automata [Gauwin and Niehren 2011]. XSeq on the other hand, is an MSO-complete language (and hence, subsumes XPath) and therefore, our translation to VPAs handles a much larger class of queries.

The current manuscript is an extended version of a conference paper [Mozafari et al. 2012], with new material that were not published in the SIGMOD version, including new applications (Sections 3.6, 3.8), formal semantics (Section 5), and proofs and complexity results (Sections 6.1, 6.2 and Appendix B).

9. CONCLUSION AND FUTURE WORK

We have described the design and implementation of XSeq, a query language for XML streams that adds powerful extensions to XPath while remaining very amenable to optimization and efficient implementation. We studied the power and efficiency of XSeq both in theory and in practice, and proved that XSeq subsumes Regular XPath and its dialects, and hence, provides the first implementation of these languages as well. Then, we showed that well-known complex queries from diverse applications, can be easily expressed in XSeq, whereas they are difficult or impossible to express in XPath and its dialects. The design and implementation of XSeq leveraged recent advances in VPAs and their online evaluation and optimization techniques.

Inasmuch as XPath provides the kernel of several query languages, such as XQuery, we expect that these languages will also benefit from the extensions and implementation techniques described in this paper. In analogy to YFilter [Diao et al. 2003], where thousands of XPath expressions were merged into one NFA, the fact that VPAs are closed under union creates important opportunities for concurrent execution of numerous number of XSeq queries. Another line of future research is to use XSeq in applications with other examples of visibly pushdown words, such as software analysis, JSON files, and RNA sequences.

ACKNOWLEDGMENTS

This work was supported in part by NSF (Grant No. IIS 1118107). We would like to thank the reviewers, Balder ten Cate, Alexander Shkapsky, Nikolay Laptev and Shi Gao for their comments.

REFERENCES

ALEXANDER, M., FAWCETT, J., AND RUNCIMAN, P. 2000. *Nursing practice: hospital and home : the adult*. Churchill Livingstone; 2nd edition.

ALUR, R. AND MADHUSUDAN, P. 2004. Visibly pushdown languages. In *STOC*.

ALUR, R. AND MADHUSUDAN, P. 2006. Adding nesting structure to words. In *Developments in Language Theory*.

AMAGASA, T., YOSHIKAWA, M., AND UEMURA, S. 2000. A data model for temporal xml documents. In *DEXA*. 334–344.

BAMFORD, R. AND ET. AL. 2009. Xquery reloaded. *VLDB*.

BARTON, C. AND ET. AL. 2003. Streaming xpath processing with forward and backward axes. In *ICDE*.

BONCZ, P. AND ET. AL. 2006. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *SIGMOD*.

BRENNA, L., GEHRKE, J., HONG, M., AND JOHANSEN, D. 2009. Distributed event stream processing with non-deterministic finite automata. In *DEBS*.

CATE, B. T. AND LUTZ, C. 2009. The complexity of query containment in expressive fragments of xpath 2.0. *J. ACM 56,* 6.

CHEN, Y., DAVIDSON, S. B., AND ZHENG, Y. 2006. An efficient xpath query processor for xml streams. In *ICDE*.

DIAO, Y., ALTINEL, M., FRANKLIN, M. J., ZHANG, H., AND FISCHER, P. 2003. Path sharing and predicate evaluation for high-performance xml filtering. *TODS 28,* 4.

FLORESCU, D. AND ET. AL. 2003. The bea/xqrl streaming xquery processor. In *VLDB*.

FURCHE, T., GOTTLOB, G., GRASSO, G., SCHALLHART, C., AND SELLERS, A. J. 2011. Oxpath: A language for scalable, memory-efficient data extraction from web applications. *PVLDB 4,* 11.

GAUWIN, O. AND NIEHREN, J. 2011. Streamable fragments of forward xpath. In *CIAA*. 3–15.

GAUWIN, O., NIEHREN, J., AND TISON, S. 2011. Queries on xml streams with bounded delay and concurrency. *Inf. Comput. 209,* 3, 409–442.

JOSIFOVSKI, V., FONTOURA, M., AND BARTA, A. 2005. Querying xml streams. *VLDB Journal 14,* 2.

KAY, M. 2008. Ten reasons why saxon xquery is fast. *IEEE Data Eng. Bull. 31,* 4.

KNUTH, D. E., JR., J. H. M., AND PRATT, V. R. 1977. Fast pattern matching in strings. *SIAM J. Comput. 6,* 2.

KOCH, C. 2009. Xml stream processing. In *Encyclopedia of Database Systems*.

LAPTEV, N. AND ZANIOLO, C. 2012. Optimization of massive pattern queries by dynamic configuration morphing. In *ICDE*. 917–928.

LUCKHAM, D. C. 2001. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley.

MADHUSUDAN, P. AND VISWANATHAN, M. 2009. Query automata for nested words. In *MFCS*.

MARX, M. 2005. Conditional xpath. *TODS 30,* 4.

MOZAFARI, B., ZENG, K., AND ZANIOLO, C. 2010a. From regular expressions to nested words: Unifying languages and query execution for relational and xml sequences. *PVLDB 3,* 1.

MOZAFARI, B., ZENG, K., AND ZANIOLO, C. 2010b. K*sql: A unifying engine for sequence patterns and xml. In *SIGMOD*.

MOZAFARI, B., ZENG, K., AND ZANIOLO, C. 2012. High-performance complex event processing over xml streams. In *SIGMOD Conference*. 253–264.

OLTEANU, D., KIESLING, T., AND BRY, F. 2003. An evaluation of regular path expressions with qualifiers against xml streams. In *ICDE*.

PENG, F. AND CHAWATHE, S. S. 2003. Xpath queries on streaming data. In *SIGMOD Conference*.

PITCHER, C. 2005. Visibly pushdown expression effects for xml stream processing. In *PLAN-X*.

SCHMIDT, A. AND ET. AL. 2002. Xmark: a benchmark for xml data management. In *VLDB*.

SNODGRASS, R. T. 2009. Tsql2. In *Encyclopedia of Database Systems*.

STRÖMBÄCK, L. AND SCHMIDT, S. 2009. An extension of xquery for graph analysis of biological pathways. In *DBKDA*.

TANG, N. V. 2009. A tighter bound for the determinization of visibly pushdown automata. In *INFINITY*.

TEN CATE, B. 2006. The expressivity of xpath with transitive closure. In *PODS*.

TEN CATE, B. AND MARX, M. 2007a. Axiomatizing the logical core of xpath 2.0. In *ICDT*.

TEN CATE, B. AND MARX, M. 2007b. Navigational xpath: calculus and algebra. *SIGMOD Record 36,* 2.

TEN CATE, B. AND SEGOUFIN, L. 2008. Xpath, transitive closure logic, and nested tree walking automata. In *PODS*.

VAGENA, Z., MORO, M. M., AND TSOTRAS, V. J. 2007. Roxsum: Leveraging data aggregation and batch processing for xml routing. In *ICDE*.

WANG, F., ZANIOLO, C., AND ZHOU, X. 2008. Archis: an xml-based approach to transaction-time temporal database systems. *VLDB J. 17,* 6, 1445–1463.

WU, E., DIAO, Y., AND RIZVI, S. 2006. High-performance complex event processing over streams. In *SIGMOD*.

ZANIOLO, C. 2009. Event-oriented data models and temporal queries in transaction-time databases. In *TIME*. 47–53.

ZENG, K., YANG, M., MOZAFARI, B., AND ZANIOLO, C. 2013. Complex pattern matching in complex structures: the xseq approach. In *ICDE Demo*.

Received February 2007; revised March 2009; accepted June 2009

APPENDIX

In this appendix, we provide a brief overview of the visibly pushdown automata (VPA) followed by proofs for our complexity results in Section 6.

A. BACKGROUND ON VPA

Informally, visibly pushdown words and their closely related models, namely nested words [Alur and Madhusudan 2006], model a sequence of letters (i.e., a "normal" word) together with hierarchical edges connecting certain positions along the word. The edges are properly nested (i.e., edges do not cross), but some edges can be pending. Visibly pushdown words generalize normal words (all positions are internal-data) and ordered trees. Also, natural operations (such as concatenation, prefix, suffix) on words are easily generalized to nested words. Visibly pushdown words have found applications in many areas, ranging from program analysis to XML, and even representations of genomic data [Alur and Madhusudan 2006].

Visibly Pushdown Automata (VPA) are a natural generalization of finite state automata to visibly pushdown words. Visibly pushdown languages (VPLs) consist of languages accepted by VPAs. While VPLs enjoy higher expressiveness and succinctness compared to word and tree automata, their decision complexity and closure properties are analogous to the corresponding word and tree special cases. For example, VPLs are closed under union, intersection, complementation, concatenation, and Kleene-* [Alur and Madhusudan 2004]; deterministic VPAs are as expressive as their non-deterministic counterparts; and membership, emptiness, language inclusion and equivalence are all decidable [Alur and Madhusudan 2004; 2006]. Next, we briefly recall the formal definition of a VPA. Readers are referred to the seminal paper [Alur and Madhusudan 2004] for more details.

Let Σ be the finite input alphabet, and let $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$ be a partition of Σ. The intuition behind the partition is: Σ_c is the finite set of *call* (push) symbols, Σ_r is the finite set of *return* (pop) symbols, and Σ_i is the finite set of *internal* symbols. Visibly pushdown automata are formally defined as follows:

Definition A.1. A visibly pushdown automaton (VPA) M over S is a tuple $(Q, Q_0, \Gamma, \delta, F)$ where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, Γ is a finite stack alphabet with a special symbol $\perp$ (representing the bottom-of-stack), and $\delta = \delta_c \cup \delta_r \cup \delta_i$ is the transition relation, where $\delta_c \subseteq Q \times \Sigma_c \times Q \times (\Gamma \backslash \{\perp\})$, $\delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$, and $\delta_i \subseteq Q \times \Sigma_i \times Q$.

If $(q, c, q', \gamma) \in \delta_c$, where $c \in \Sigma_c$ and $\gamma \neq \perp$, there is a *push-transition* from q on input c where on reading c, γ is pushed onto the stack and the control changes from state q to q'; we denote such a transition by $q \xrightarrow{c/+\gamma} q'$. Similarly, if $(q, r, \gamma, q') \in \delta_r$, there is a *pop-transition* from q on input r where γ is read from the top of the stack and popped (if the top of the stack is $\perp$, then it is read but not popped), and the control state changes from q to q'; we denote such a transition $q \xrightarrow{r/-\gamma} q'$. If $(q, i, q') \in \delta_i$, there is an *internal-transition* from q on input i where on reading i, the state changes from q to q'; we denote such a transition by $q \xrightarrow{i} q'$. Note that there are no stack operations on internal transitions. We write St for the set of *stacks* $\{w\perp | w \in (\Gamma\backslash\{\perp\})*\}$. A *configuration* is a pair (q, σ) of $q \in Q$ and $\sigma \in St$. The transition function of a VPA can be used to define how the configuration of the machine changes in a single step: we say $(q, \sigma) \xrightarrow{a} (q', \sigma')$ if one of the following conditions holds:

(1) If $a \in \Sigma_c$ then there exists $\gamma \in \Gamma$ such that $q \xrightarrow{a/+\gamma} q'$ and $\sigma' = \gamma \cdot \sigma$
(2) If $a \in \Sigma_r$, then there exists $\gamma \in \Gamma$ such that $q \xrightarrow{a/-\gamma} q'$ and either $\sigma = \gamma \cdot \sigma'$, or $\gamma = \perp$ and $\sigma = \sigma' = \perp$
(3) If $a \in \Sigma_i$, then $\gamma \in \gamma'$ and $\sigma = \sigma'$.

A (q_0, w_0)-*run* on a word $u = a_1 \cdots a_n$ is a sequence of configurations $(q_0, w_0) \xrightarrow{a_1} (q_1, w_1) \cdots \xrightarrow{a_n} (q_n, w_n)$, and is denoted by $(q_0, w_0) \xrightarrow{u} (q_n, w_n)$. A word u is accepted by M if there is a run $(q_0, w_0) \xrightarrow{u} (q_n, w_n)$ with $q_0 \in Q_0$, $w_0 = \perp$, and $q_n \in Q_F$. The language $L(M)$ is the set of words accepted by M. The language $L \subseteq \Sigma^*$ is a visibly pushdown language (VPL) if there exists a VPA M with $L = L(M)$.

$$
\begin{aligned}
\text{PathExpr} &::= \text{PathExpr } '\mathit{intersect}' \text{ PathExpr } \mid \text{VStep} \\
\text{VStep} &::= \text{Step Variable}* \\
\text{Step} &::= \text{Axis } '::' \text{ NameTest [Variable]} \\
&\mid \text{Axis } '::' \text{ NameTest} \\
\text{Axis} &::= \odot \mid \downarrow \mid \uparrow \mid \rightarrow \mid \leftarrow
\end{aligned}
$$

Fig. 13. CXSeqA

B. CORE XSEQ PROOF OF REGULARITY

In this section, we prove Theorems 6.2 and 6.3. We first introduce two simpler query languages, called CXSeqA and CXSeqB. Next, we show how every CXSeq query can be translated into an equivalent CXSeqA query, and every CXSeqA query can be translated into an equivalent CXSeqB query. Then, we show that the domain of a CXSeqB query can be captured by a VPA (MSO equivalent model), and for every Top Down Tree Automata (MSO equivalent model) T, there exists a CXSeq query that has domain equivalent to the language of T. These last two results together prove that every MSO definable language can be expressed as the domain of some CXSeq query and viceversa (Theorem 6.2). Theorem 6.3 is a consequence of the complexity of the query transformations.

B.1. Core XSeq with Variable Concatenation

In Fig. 13 we introduce CXSeqA as a syntactic restriction of CXSeq and show that CXSeq can be compiled to CXSeqA. In the following constructions we will use $\cap$ instead of $'intersect'$.

THEOREM B.1. *Every CXSeq query K can be transformed into an equivalent CXSeqA query K'.*

PROOF. Given a production (v, π) we define the following function $\text{st}(\pi)$ that extracts a starred path.

— $\text{st}(\pi \cap \pi') =$ if $\text{st}(\pi) \neq null$ then $\text{st}(\pi)$ else $\text{st}(\pi')$;
— $\text{st}(\pi*) = \pi$;
— else $\text{st}(\pi) = null$.

We are given a CXSeq query $K = (V, v_0, \rho)$ such that there exists at least a production $(v, \pi) \in \rho$ such that $\text{st}(\pi) \neq null$.
We rewrite K as follows.

(1) Pick a production (v, π) such that $\text{st}(\pi) = \pi'$;
(2) Replace π' in π with $\odot :: \_fv$ where fv is a fresh variable;
(3) Add the productions $(fv, \odot :: \_)$ and $(fv, \pi fv)$;
(4) Repeat from 1 until there are no more starred productions.

The algorithm terminates since at every step one star is eliminated from the productions. The resulting query is still *safe*. The new query will have $O(|\rho|)$ productions. □

Next we introduce a notion of query size and show that every query of size n is equivalent to some query of size 0. Given a query $K = (V, v_0, \rho)$, the size of K, $size(K)$, is defined as follows. $size(K) = max_{(v,\pi)\in\rho} psize(\pi)$ where

— $psize(\pi \cap \pi') = max(psize(\pi), psize(\pi'))$;
— $psize(d :: a[v]v_1 \dots v_n + 2) = n + 1$;
— $psize(d :: a[v]) = 0$;
— $psize(d :: av_1 \dots v_{n+2}) = n + 1$;

— $psize(d :: a) = 0$.

LEMMA B.2. *Every CXSeqA query K of size n can be transformed into an equivalent CXSeqA query K' of size smaller or equal than 1.*

PROOF. We are given a CXSeqA query $K = (V, v_0, \rho)$ such that there exists at least a production $(v, \pi) \in \rho$ such that $psize(\pi) > 1$.

We pick a production $p = (v, \pi)$ such that $psize(\pi) > 1$ we do the following. We first remove p from ρ. Now we compute the set of productions $\mathrm{pr}_v(\pi)$.

— $\mathrm{pr}_v(\pi \cap \pi') = \mathrm{pr}(\pi) \cup \mathrm{pr}(\pi')$;
— $\mathrm{pr}_v(d :: a[v]v_1\ \dots\ v_{n+2}) = \{(v_2 \dots v_{n+2}, \odot :: \_\ v_2 \dots v_{n+2})\}$;
— $\mathrm{pr}_v(d :: av_1\ \dots\ v_{n+2}) = \{(v_2 \dots v_{n+2}, \odot :: \_\ v_2 \dots v_{n+2})\}$;
— else $\mathrm{pr}_v(\pi) = \{\}$.

and the production $\mathrm{prod}(\pi)$:

— $\mathrm{prod}(\pi \cap \pi') = \mathrm{pr}(\pi) \cap \mathrm{pr}(\pi')$;
— $\mathrm{prod}(d :: a[v]v_1\ \dots\ v_{n+2}) = d :: a[v]v_1(v_2 \dots v_{n+2})$;
— $\mathrm{prod}(d :: av_1\ \dots\ v_{n+2}) = d :: a[v]v_1(v_2 \dots v_{n+2})$;
— else $\mathrm{prod}(\pi) = \pi$.

Therefore we compute $\rho = \rho \cup \mathrm{pr}(\pi) \cup \{(v, \mathrm{prod}(\pi))\}$.

We also compute the following new set of variables $\mathrm{nv}(\pi)$.

— $\mathrm{nv}(\pi \cap \pi') = \mathrm{pr}(\pi) \cup \mathrm{pr}(\pi')$;
— $\mathrm{nv}(d :: a[v]v_1 \dots v_{n+2}) = \{(v_2 \dots v_n + 2)\}$;
— $\mathrm{nv}(d :: av_1 \dots v_{n+2}) = \{(v_2 \dots v_n + 2)\}$;
— else $\mathrm{nv}(\pi) = \{\}$.

Next, we update the set V as follows: $V := V \cup P_\pi \cup \{(v, \mathrm{prod}(\pi))\}$. The algorithm repeats this step until there are no more productions of size greater than 1. The algorithm terminates since at every step, if a rule of size n is picked the number of productions of size n is reduced by one and only rules of size smaller than n are produced.

We now need to show that every intermediate result is a *safe* query. This is straightforward from the construction.

We define the length of a query $length(K)$ as the sum of the length of all the productions in ρ. $length(L) = \sum_{(v,\pi)\in\rho} plength(\pi)$ where

— $plength(\pi \cap \pi') = plength(\pi) + plength(\pi')$;
— $plength(d :: a[v]v_1 \dots v_{n+1}) = n\}$;
— $plength(d :: av_1 \dots v_{n+1}) = n$;
— else $plength(\pi) = 0$.

The new query will have $O(|\rho| \cdot length(K))$ productions. □

LEMMA B.3. *Every CXSeqA query K can be transformed into an equivalent query $K' = (V', v_0', \rho')$ where for each production $(v, \pi)\rho'$, $|\mathrm{dv}(\pi)| = 0$.*

PROOF. Using lemma B.2 we reduce the query to be of size 1.

Now, we are given a CXSeqA query $K = (V, v_0, \rho)$ such that there exists at least a production $(v, \pi) \in \rho$ such that $psize(\pi) = 1$.

For every production $p = (v, \pi)$ such that $psize(\pi) = 1$ we do the following.

We define the following set of variable pairs that need to be normalized as follows. Given a production (v, π), let $vp(\pi)$ be defined as follows:

— $vp(\pi_1 \cap \pi_2) = vp(\pi_1) \cup vp(\pi_2)$
— $vp(d :: a[v]v_1v_2) = \{(v_1, v_2)\}$;

$$\begin{array}{rcl} \text{PathExpr} & ::= & \text{PathExpr } '\mathit{intersect}' \text{ PathExpr } \mid \text{ VStep} \\ \text{VStep} & ::= & \text{Step Variable} \mid \text{Step} \\ \text{Step} & ::= & \text{Axis } ' ::' \text{ NameTest [Variable]} \\ & & \mid \text{ Axis } ' ::' \text{ NameTest} \\ \text{Axis} & ::= & \odot \mid \downarrow \mid \uparrow \mid \rightarrow \mid \leftarrow \end{array}$$

Fig. 14. CXSeqB

— $vp(d :: av_1v_2) = \{(v_1, v_2)\}$;
— else $vp(\pi) = \{\}$.

We show how to eliminate a single variable pair from all the productions. The algorithm then iterates. Let (v_1, v_2) be a pair in $vp(\pi)$ for some $(v, \pi) \in \rho$.

Let $X = \mathrm{nv}(v_1)$ and let $\rho(X) = \bigcup_{x \in X} \rho(x)$.

Create a copy of P' of $P = \rho(X)$, where each variable occurrence $x \in X$ that is not a predicate is replaced by a new variable x'. The set of production P' is used to generate the concatenation effect in the production p. Compute $P'' = \{(v, h_{v_2}(\pi)) | (v, \pi) \in P'\}$ using the following function h.

— $h_{v_2}(\pi_1 \cap \pi_2) = h_{v_2}(\pi_1) \cap h_{v_2}(\pi_2)$
— $h_{v_2}(d :: a[v]) = d :: a[v]v_2$;
— $h_{v_2}(d :: a) = d :: av_2$;
— else $h_{v_2}(\pi) = \pi$.

Add P'' to the set of productions ρ.

Now, for each production $(v, \pi) \in \rho$ containing v_1, v_2 replace it with $(v, g_{v_1 v_2}(\pi))$ where

— $g_{v_1 v_2}(\pi_1 \cap \pi_2) = g(\pi_1) \cap g(\pi_2)$
— $g_{v_1 v_2}(d :: a[v]v_1v_2) = d :: a[v]v_1'$;
— $g_{v_1 v_2}(d :: av_1v_2) = d :: av_1'$;
— else $g(\pi) = \pi$.

Repeat until there are no more production of size 1.

The termination is guaranteed by the fact that whenever a pair (a, b) is eliminated, the same pair cannot appear in the derivation of a, therefore we can always identifying a total order in variables $v_1, \ldots, v_n$, such that whenever we eliminate a pair for a production v_i the number of productions of size 1 out of v_i decreases and only the number of productions for v_j, $j > i$ can increase. The correctness of the algorithm is guaranteed by the fact that every step preserves safety. □

B.2. Core XSeq Basic

Fig. 14 presents the syntax of CXSeqB where queries are only allowed to have size 0. We then show that CXSeqA, and therefore CXSeq, have the same expressiveness as CXSeqB. Finally we show that all the languages we introduced can be compiled into equivalent VPA and are all MSO complete.

Using Lemma B.2 and B.3 we prove the following theorem.

THEOREM B.4. *Every CXSeqA query K can be transformed into an equivalent CXSeqB query K'.*

B.2.1. Translating CXSeqB into VPA. In the following we show that a CXSeqB query can be encoded as an equivalent visibly pushdown automata.

THEOREM B.5. *For every query $K = (V, v_0, \rho)$ in CXSeqB there exists an equivalent VPA A over Σ such that a word w is accepted by A iff $K(w) \neq \emptyset$. A has $O(r^5 \cdot 2^r)$ states where $r = |\rho|$.*

PROOF. Given a query $K = (V, v_0, \rho)$ we construct an equivalent nondeterministic VPA $A = (Q, Q_0, \Gamma, \delta, F)$. The main idea of the construction is that each variable v in V corresponds to a query and at every step the automaton keeps in state the information on which queries are consistent with the neighboring nodes. Whenever the state contains the query v_0, a bit is set to 1 to remember that the starting query has been answered.

The state of A will encode which productions in the grammar the current node and its neighbors have been traversed with. The construction will be nondeterministic since some query results will depend on stretches of the input that have not been read yet. Assuming the query has been executed over the input using a set of productions, the run of the automaton will guess which productions have been used to process each element of the input.

We define a notion of consistency that will allow us to show the correctness of the construction. For $v \in V$, let $\rho(v) = \{(v, \pi) \mid (v, \pi) \in \rho\}$, be the set of productions in ρ with first element v. We introduce the function $f_\pi : (V, (\Sigma \cup \{\_\})^5$, that given a path π, computes the set of queries that must answer the current nodes and its neighbors nodes. We define f inductively as follows:

— $f(\odot :: s[v_1]v_2) = ((\{v_1, v_2\}, s), (\{\}, \_), (\{\}, \_), (\{\}, \_), (\{\}, \_))$;
— $f(\odot :: s[v_1]) = ((\{v_1\}, s), (\{\}, \_), (\{\}, \_), (\{\}, \_), (\{\}, \_))$;
— $f(\odot :: s) = ((\{\}, s), (\{\}, \_), (\{\}, \_), (\{\}, \_), (\{\}, \_))$;
— $f(\downarrow :: s[v_1]v_2) = ((\{\}, \_), (\{v_1, v_2\}, s), (\{\}, \_), (\{\}, \_), (\{\}, \_))$;
— $f(\uparrow :: s[v_1]v_2) = ((\{\}, \_), (\{\}, \_), (\{v_1, v_2\}, s), (\{\}, \_), (\{\}, \_))$;
— $f(\uparrow :: s[v_1]v_2) = ((\{\}, \_), (\{\}, \_), (\{\}, \_), (\{v_1, v_2\}, s), (\{\}, \_))$;
— $f(\uparrow :: s[v_1]v_2) = ((\{\}, \_), (\{\}, \_), (\{\}, \_), (\{\}, \_), (\{v_1, v_2\}, s))$;
— if $f(\pi_1) = ((C_1, c_1), (FC_1, fc_1), (P_1, p_1), (NS_1, ns_1), (PS_1, ps_1))$, and $f(\pi_2) = ((C_2, c_2), (FC_2, fc_2), (P_2, p_2), (NS_2, ns_2), (PS_2, ps_2))$, then $f(\pi_1 \cap \pi_2) = ((C_1 \cup C_2, c_1 \diamondsuit c_2), (FC_1 \cup FC_2, fc_1 \diamondsuit fc_2), (P_1 \cup P_2, p_1 \diamondsuit p_2), (NS_1 \cup NS_2, ns_1 \diamondsuit ns_2), (PS_1 \cup PS_2, ps_1 \diamondsuit ps_2))$, where if $a \in \Sigma \cup \{\_\}$, then $\diamondsuit(a, \_) = \diamondsuit(\_, a) = \diamondsuit(a, a) = a$, and it is undefined in any other case (when undefined the path is also inconsistent, so it should not belong to the productions);
— the omitted cases are similar.

Each state of A is an element of the relation $((2^\rho \times \Sigma) \cup \{\bot\})^5 \times \{0, 1\}$, such that before reading the node n, A is in state $((PC, a), (PFC, b), (PP, c), (PNS, d), (PPS, e), i)$ iff

— for each $(v, \pi) \in PC$ there exists a derivation of v starting in n with the production (v, π) and the label of n is a,
— for each $(v, \pi) \in PFC$ there exists a derivation of v starting from the first child n' of n with the production (v, π) and the label of n' is b,
— for each $(v, \pi) \in PP$ there exists a derivation of v starting from the parent n' of n with the production (v, π) and the label of n' is c,
— for each $(v, \pi) \in PNS$ there exists a derivation of v starting from the next sibling n' of n with the production (v, π) and the label of n' is d,
— for each $(v, \pi) \in PPS$ there exists a derivation of v starting from the previous sibling n' of n with the production (v, π) and the label of n' is e,
— $i = 1$ iff a state such that $(v_0, \pi) \in C$, for some π, has been already traversed.

Whenever a component of the state is $\perp$ it means that that neighbor is not defined. For example if the second component is $\perp$ it means that the node does not have a parent.

We need to restrict the states to not violate the semantics of the production we defined before (f). Given a set of productions Π, we define $\nu(\Pi) = \{v \mid \exists (v, \pi) \in \Pi\}$. Let T be the following set:

$$\{((C, a), t_1, t_2, t_3, t_4) \mid (v, \pi) \in C \wedge f_v(\pi) = ((C, a), t_1, t_2, t_3, t_4)\}$$

A state $((PC', a'), (PFC', b'), (PP', c'), (PNS', d'), (PPS', e'), i)$ is consistent with respect to ρ if the following holds:
for each $((C, a), (FC, b), (P, c), (NS, d), (PS, e)) \in T$, $\diamondsuit(a, a'), \diamondsuit(b, b'), \diamondsuit(c, c'), \diamondsuit(d, d'), \diamondsuit(e, e')$ are defined and $C \subseteq \nu(PC')$, $FC \subseteq \nu(PFC')$, $P \subseteq \nu(PP')$, $NS \subseteq \nu(PNS')$, and $PS \subseteq \nu(PPS')$. If one of the component is $\perp$ the corresponding pair in the tuple should be $(\{\}, \_)$.

The set of states $Q \subseteq ((2^\rho \times \Sigma \cup \{\perp\})^5 \times \{0, 1\}$ contains all the tuples consistent with respect to ρ. The set of stack states Γ is the same as Q. The set of initial states is $Q_0 = (2^\rho \cup \{\perp\}) \times (2^\rho \cup \{\perp\}) \times \{\perp\} \times \{\perp\} \times \{\perp\} \times \{0, 1\} \cap Q$. The set of final states is $F = \{\perp\} \times \{\perp\} \times \{\perp\} \times \{\perp\} \times (2^\rho \cup \{\perp\}) \times \{0, 1\} \cap Q$. where only the previous sibling has replied to some queries (maybe the empty set if the root is not in any). We assume that A only accepts well-matched words.

We can now give the transition function δ of A. We need to maintain the state invariant defined before.

— for every S_1, S_2 such that $(FC, S_1, (PC, a), S_2, \perp, i) \in Q$,
$(FC, S_1, (PC, a), S_2, \perp, i), \quad ((PC, a), FC, P, NS, PS, i)) \in ((PC, a), FC, P, NS, PS, i)(\langle a)$;
— for every S_1, S_2 such that $(FC, S_1, (PC, a), S_2, \perp, 1) \in Q$ and $v_0 \in \nu(PC)$,
$(FC, S_1, (PC, a), S_2, \perp, i), \quad ((PC, a), FC, P, NS, PS, 1)) \in ((PC, a), FC, P, NS, PS, 0)(\langle a)$;
— for every S_1, S_2 such that $(NS, S_1, P, S_2, C, max(i, j)) \in Q$,
$(NS, S_1, P, S_2, C, max(i, j)) \in (\perp, \perp, P', PS', \perp, i)(a\rangle, (C, FC, P, NS, PS, j)$.

This concludes the proof.

We can actually show that A does not need to have Σ as a state component, but can instead just keep a set $\Sigma' \cup \{\_\}$ where Σ' contains only symbols that actually appear in the query, and $\_$ is a place holder for all the other symbols.

The automaton will have $O(r^5 \cdot 2^r)$ states where $r = |\rho|$. $\square$

B.2.2. Translating Top-down Tree Automata into CXSeqB. To show MSO completeness we translate nondeterministic Top-Down Tree Automata (TA) into CXSeqB. A TA A over binary trees, is a tuple (Q, q_0, δ) where Q is a set of states and $q_0 \in Q$ is the initial states. The productions in δ are of the form: $q(b(x, y)) \to q_1(x), q_2(y)$ or $q(z)$ where z and b are respectively a leaf and an internal node. The set of tree accepted by A starting in state q (L_q) is defined inductively as follows: 1) $a(x, y) \in L_q$ if $q(a(x, y)) \to q_1(x), q_2(y) \in \delta$ and $x \in L_{q_1}$ and $y \in L_{q_2}$, 2) $b \in L_q$ if $q(b) \in \delta$.

We consider an encoding of unranked trees into binary trees where for each node $n = a(x, y)$, x, n has label a, x is the first child of n and y is the next sibling of n.

Given a TA $A = (Q, q_0, \delta)$ we construct an equivalent query (V, v_0, ρ), where $V = Q$, $v_0 = q_0$ and ρ is defined as follows. For every rule $q(z) \in \delta$ the production $(q, z :: \odot)$ will belong to ρ. For every rule $q(b(x, y)) \to q_1(x), q_2(y) \in \delta$ the production $(q, (\odot :: b) \cap (\downarrow :: \_[q_1]) \cap (\rightarrow :: \_[q_2]))$ will belong to ρ.

B.3. Regularity of CXSeq and Complexity

The next two theorems follow from the transformation of Section B.2.2 and Theorem B.5.

THEOREM 6.2. *For every CXSeq query $K = (V, v_0, \rho)$, the domain D_K of K is an MSO definable language. Conversely for every MSO definable language L, there exists a CXSeq query K such that $L = D_K$.*

THEOREM 6.3. *For every CXSeq query $K = (V, v_0, \rho)$, there exists an equivalent VPA A over Σ such that $L(A) = D_K$. A will have $O(r^5 \cdot length(K)^5 \cdot 2^{r \cdot length(K)})$ where $r = |\rho|$.*