Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

SPath: an extensible query-language for Scala

Nicholas Nguyen[a]

a. http://nicnguyen.github.com

Abstract Scala combines the functional and object-oriented paradigms and is good at supporting embedded domain-specific languages. Scala presents therefore an opportunity for designing a new kind of query-language that belongs to the family of XPath-like query-languages, for querying semi-structured data that is stored within internal memory. In the terminology of XPath, a relation between the nodes of a tree, is called an axis and their purpose is to provide a way of navigating between the nodes within a tree. An XPath query is then, approximately, a path of axis-steps, resembling a directory path in a file-system. XPath has 13 axes. However, it is natural to extend the notion of axis to any relation on tree-nodes, by allowing queries with user-defined axes. The effect of the approach that has been taken in this article is an alternative syntax and semantics for user-defined axes, compared with XPath 2.0.

It also turns out that separating the query-language from specific types of trees, enables reuse with any tree data-structure that conforms to the composite design-pattern.

SPath is an XPath-like query-language based on Linear Temporal Logic that has been implemented as a domain-specific language, embedded into Scala.

Keywords Scala, SPath, domain-specific languages, linear temporal logic

1 Introduction.

Semi-structured data has become ubiquitous in object-oriented software development. Indeed, for the modern software developer, it may seem as though XML pervades everywhere from web-services and domain-modeling, to the more mundane use of configuration files. Query-languages for XML are also widely supported in terms of the W3C specifications for XPath 1.0 [W3Cb], XPath 2.0 [W3Cc] and XQuery [W3Cd], as well as their implementations. The implementations of XPath within object-oriented languages vary from strict adherence to the specifications [jsr04, xqj09] to variations that are loosely based on the essential features of XPath such as in Scala [OSV08] and LINQ-to-XML for C# [Mic]. Both Scala and C# share the similar style of allowing user-defined functions to appear in the path of a query, as defined by XPath 2.0.

The user-defined functions of XPath 2.0, in essence, specify relationships between tree-nodes. A query is then, approximately, formed by chaining together a series of calls to these axis-functions.

This article presents a different approach for writing queries with user-defined axes, in a new query-language called SPath. The main advantages that SPath brings are:

- The syntax for user-defined axes in SPath are treated in the same way as the syntax for the core axes of XPath 1.0. In XPath 2.0, the syntax for user-defined axes and the core axes, is different.
- When the axis-relations are decoupled from the type of the document-tree that is to be queried, as they are in SPath, then the query-language can be reused with any tree data-structure that has been implemented according to the composite design-pattern.
- SPath is an extension of the conditional axes of [Mar04b], with user-defined axes. User-defined axes can also be conveniently defined in terms of SPath queries, which introduces *higher-order* axes.

This article is organised as follows. The next section provides a brief overview of XML and XPath. The syntax for a core subset of XPath 2.0 is presented in Section 2.1. Conditional axes [Mar04b] are explained in Section 2.2.

Section 3 presents an overview of SPath and a translation from the core XPath 2.0 of Section 1, into SPath. Section 4 presents two examples of SPath for querying native XML documents in Scala.

The remainder of the article presents SPath in detail. Section 5 introduces SPath as an extension of LTL in which formulas are annotated with arbitrary node-relations. These node-relations represent user-definable axes. Section 6 summarises the implementation of query evaluation in SPath and its theoretical worst-case running time. SPath is examined with the XPathMark performance test [Fra07] and a separate comparison with Scala's native XML API. The SPath queries based on XPathMark are in Appendix A. Section 7 presents the embedding of SPath into Scala. Section 8 concludes the article by comparing SPath with previous work from the literature. Further work for SPath is outlined in Section 9.

Query evaluation in SPath makes use of the classic algorithm of [GPV$^+$95] for translating an LTL formula into an automaton. The adapted algorithm is presented in Appendix B. Proofs are attached in Appendix C.

2 Background.

2.1 The document object model, XML and XPath 2.0.

The XPath specifications are defined in terms of a conceptual model for document-trees. That is, the nodes of the trees are labeled with tags; have attributes and siblings are ordered, but the trees do not conform to a particular API or specification for XML. XPath trees are node-labeled in contrast to being edge-labeled. The node-labeled approach affects the way trees are addressed within path expressions and how values at nodes are accessed, as described in [MSB03]. The document order is the order that nodes are encountered, when the serialised document is read in one pass as a stream from start to end. The document order is the same as the preorder traversal of the tree.

```
NodeTest      ::=  QName | *
AxisStep      ::=  Axis::Filter
Filter        ::=  NodeTest([Predicate])?
PathExpr      ::=  AxisStep(/AxisStep)*
Predicate     ::=  PathExpr
                |  Predicate and Predicate
                |  Predicate or Predicate
                |  not Predicate
Axis          ::=  FunctionCall | ForwardAxis | ReverseAxis
FunctionCall  ::=  QName(.)
ForwardAxis   ::=  self | child | descendant | descendent-or-self
                |  following | following-sibling
ReverseAxis   ::=  parent | ancestor | ancestor-or-self
                |  preceding | preceding-sibling
```

Figure 1 – A core syntax for a subset of XPath 2.0.

The document object model (DOM) defines a general interface for document-trees. Its interface enables navigation and manipulation of trees that are represented by more specialised formats such as XML or XHTML. The DOM specification [W3Ca] is language and platform neutral, due to it being defined using the interface definition language (IDL). Of most relevance to this work is that the DOM defines trees using the composite pattern [GHJV95]. In particular, in the following IDL code fragment from the DOM specifications, the `Node` interface, which represents tree nodes has an attribute, `childNodes`, to access the children of a node. This attribute has the type of a `NodeList`, which is composed, recursively, of an ordered list of `Node`.

```
interface Node {
 ...
 readonly attribute unsigned short nodeType;
 readonly attribute Node parentNode;
 readonly attribute NodeList childNodes;
 ...
}

interface NodeList {
 Node item(in unsigned long index);
 readonly attribute unsigned long length;
};
```

Similiarly, the `parentNode` attribute of the `Node` interface provides a reference to a node's parent. In Section 7.2, SPath is shown to be compatible with any tree API, which at a minimum, provides access to a node's children in document order, such as the attribute `childNodes`.

The `Node` interface represents all kinds of objects within an XML document, such as elements, attributes, text nodes, CDATA, comments and processing instructions. The value of a node's `nodeType` attribute, has a value of an enumerated type and determines which kind of XML object is represented by a node.

The formal syntax for a core of XPath 2.0 is shown in Figure 1 [1]. The syntax is a core subset of unabbreviated XPath that is based on the XPath 2.0 specification

[1] The syntax is defined in BNF extended with the regular-expression operators, ? and *.

[W3Cc]. The core of the XPath language consists of path expressions, `PathExpr`, for location steps and predicates, `Predicate`, which act like filters on tree nodes. Predicates can also contain path expressions, recursively. An axis step consists of an axis, a node test and an optional predicate. A node test can be either an element name or $*$, which matches any element node. The syntax in Figure 1 has just one feature of XPath 2.0, namely function calls can occur in location steps. The function call `QName(.)` applies the function `QName` to its argument `.`, which is bound to the current context node.

The axes are divided into forward and reverse axes, according to the relative position, in document order, of the context node to which an axis is applied and the nodes returned by the axis.

An example query in XPath 2.0, generated by the syntax in Figure 1 is:

```
descendant-or-self::A[child::B or parent::D]/child::C
```

This query returns a set of element nodes labeled `C`. The query can be explained in a top-down manner starting at the root of the document-tree. Evaluation begins at the root and searches for descendant nodes labeled with `A`, which have either a `B` child or a `D` parent. The `C` children of such `A` nodes are then returned as the result-set of the query, in document order. In the abbreviated syntax of XPath, this query can be written as:

```
//A[B or parent::D]/C.
```

For more detail about XPath 2.0 and its data model, the reader is referred to [W3Ce], [W3Cc], [Kay04]. Readers interested in a formal semantics for evaluating XPath queries may wish to refer to [Mar04a].

2.2 Conditional axes.

Conditional axes have been proposed by Maarten Marx in [Mar04b, Mar04a]. A conditional axis takes the form: do a step along the `child` axis until `test` holds at the resulting node. Conditional axes are not present in XPath 1.0 but can be expressed in XPath 2.0 using variables, conditional constructs and recursive user-defined functions. However, direct support for conditional axes are a desirable feature for XPath-like query languages because they preserve the simple path-like syntax of XPath 1.0, and eliminate the boilerplate-code that is needed for implementing a conditional axis with variables and a looping construct. In SPath, a conditional axis takes the form `\\(f, test)` [2], and can be specified with any user-defined axis, `f`, in addition to XPath's core axes.

3 An overview of SPath.

An overview of the different stages and components within SPath is displayed in Figure 2. The user defines the SPath query, which is translated into an LTL formula and then compiled into an automaton. The evaluation engine runs the query on the document and returns a node-set in document order.

The remainder of this section aims at providing an intuition of the semantics of SPath queries by showing how XPath 2.0 queries are translated into SPath.

[2] SPath abbreviates `\\(f, *)` with `\\(f)` and `\\(child, test)` with `\\(test)`, where `val * = _ => true`.

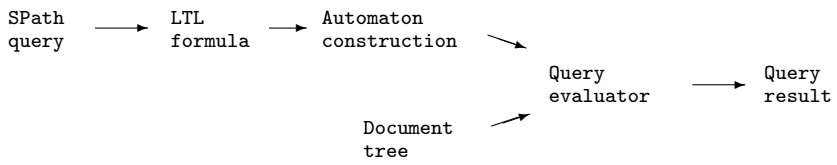

Figure 2 – The different stages and components of evaluating a query in SPath.

3.1 Translating XPath 2.0 into SPath.

In this section, the core subset for XPath 2.0, defined in Section 1 is translated into the `XSPathLite` extension of `SPath`, for querying Scala's native XML documents. `XSPathLite` is instantiated on `scala.xml.Node`. In Figure 3, each symbol in the grammar for XPath 2.0 from Section 1 is assigned a translation function. The target domain of these translation functions is the SPath class `Query`, which represents LTL formulas. Each XPath axis is mapped to a similarly named Scala function, each with the type aliased by `axis`.

The translation assumes that SPath's axis-functions, the class `Query` and the following definitions are in the current scope. These definitions are brought into the current scope, i.e. imported, by a mixin composition of the trait `XSPathLite`.

```
type axis   :  Node => Iterable[Node]
Element     :  Query
%           :  Query
\           :  (axis, Query) => Query
not         :  Query => Query
```

An axis is modeled as a function from tree nodes to ordered sets of tree nodes. `Element` represents a query that matches element nodes with specific labels. The query, `%`, is an SPath predicate that matches any element node. The function \ is an entry function for starting a path expression. The translation also makes use of the following methods, which are defined in the class `Query` :

```
\     :  (axis, Query) => Query
and   :  Query => Query
or    :  Query => Query
?     :  Query => Query
```

The function \ allows a path expression to be extended and is similar to the function \ that is imported from the trait `XSPathlite`. The function `?` creates an SPath predicate from a query and is intended to model an XPath predicate of the form, `[Predicate]`.

The fact that `Query` is the target domain for most of the translation, is a characteristic of the fluent-interface pattern for embedded domain-specific languages [Fow10]. The XPath query from Section 1:

```
descendant-or-self::A[child::B or parent::D]/child::C
```

can now be translated into the SPath query:

```
\(descendantOrSelf, A ? (\(child, B) or \(parent, D))) \(child, C)
```

$$
\begin{array}{lcl}
[\![\bullet]\!]_{NT} & : & \texttt{NodeTest} \longrightarrow \texttt{Query} \\
[\![\bullet]\!]_{F} & : & \texttt{Filter} \longrightarrow \texttt{Query} \\
[\![\bullet]\!]_{P} & : & \texttt{Predicate} \longrightarrow \texttt{Query} \\
[\![\bullet]\!]_{PE} & : & \texttt{PathExpr} \longrightarrow \texttt{Query} \\
[\![\bullet]\!]_{A} & : & \texttt{Axis} \longrightarrow \texttt{axis}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\texttt{QName}]\!]_{NT} & = & \texttt{Element(QName)} \\
[\![\texttt{*}]\!]_{NT} & = & \texttt{\%}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\texttt{NodeTest}]\!]_{F} & = & [\![\texttt{NodeTest}]\!]_{NT} \\
[\![\texttt{NodeTest[Predicate]}]\!]_{F} & = & [\![\texttt{NodeTest}]\!]_{NT}\ \texttt{?(}[\![\texttt{Predicate}]\!]_{P}\texttt{)}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\texttt{PathExpr}]\!]_{P} & = & [\![\texttt{PathExpr}]\!]_{PE} \\
[\![\texttt{Predicate}_1\ \texttt{and}\ \texttt{Predicate}_2]\!]_{P} & = & [\![\texttt{Predicate}_1]\!]_{P}\ \texttt{and}\ [\![\texttt{Predicate}_2]\!]_{P} \\
[\![\texttt{Predicate}_1\ \texttt{or}\ \texttt{Predicate}_2]\!]_{P} & = & [\![\texttt{Predicate}_1]\!]_{P}\ \texttt{or}\ [\![\texttt{Predicate}_2]\!]_{P} \\
[\![\texttt{not Predicate}]\!]_{P} & = & \texttt{not(}[\![\texttt{Predicate}]\!]_{P}\texttt{)}
\end{array}
$$

$$
[\![\texttt{Axis}_0\texttt{::Filter}_0\texttt{/Axis}_1\texttt{::Filter}_1\ \ldots\texttt{/Axis}_k\texttt{::Filter}_k\ ]\!]_{PE} =
$$
$$
\texttt{\textbackslash(}[\![\texttt{Axis}_0]\!]_{A}\texttt{,}[\![\texttt{Filter}_0]\!]_{F}\texttt{)\textbackslash(}[\![\texttt{Axis}_1]\!]_{A}\texttt{,}[\![\texttt{Filter}_1]\!]_{F}\texttt{)...\textbackslash(}[\![\texttt{Axis}_k]\!]_{A}\texttt{,}[\![\texttt{Filter}_k]\!]_{F}\texttt{)}
$$

$$
\begin{array}{lcllcl}
[\![\texttt{QName(.)}]\!]_{A} & = & \texttt{QName} & [\![\texttt{parent}]\!]_{A} & = & \texttt{parent} \\
[\![\texttt{child}]\!]_{A} & = & \texttt{child} & [\![\texttt{descendant}]\!]_{A} & = & \texttt{descendant} \\
[\![\texttt{self}]\!]_{A} & = & \texttt{self} & [\![\texttt{following-sibling}]\!]_{A} & = & \texttt{followingSibling} \\
[\![\texttt{following}]\!]_{A} & = & \texttt{following} & [\![\texttt{preceding-sibling}]\!]_{A} & = & \texttt{precedingSibling} \\
[\![\texttt{preceding}]\!]_{A} & = & \texttt{preceding} & [\![\texttt{ancestor-or-self}]\!]_{A} & = & \texttt{ancestorOrSelf} \\
[\![\texttt{ancestor}]\!]_{A} & = & \texttt{ancestor} & [\![\texttt{descendant-or-self}]\!]_{A} & = & \texttt{descendantOrSelf}
\end{array}
$$

Figure 3 – Translating a core subset of XPath 2.0 into SPath.

where `A` is defined as: `val A = Element("A")` and `B`, `D` and `E` are defined in a similar way. Here, `Element` is a companion object of the `Element` class, which has an `apply` method that creates a new `Element` object with its `label` set to the argument, e.g. `"A"` such that `A.label == "A"`. Just as the XPath query can be abbreviated as:

```
//A[B or parent::D]/C.
```

so too, the SPath query can be abbreviated as:

```
\\(A ?(\(B) or \(parent, D)))\C
```

This abbreviated SPath query uses the overloaded function `\`, which has `child` as a default axis. For example, `\(B)` is syntactic sugar for `\(child, B)`. The function `\\` represents a conditional axis [Mar04b] and is defined in Sections 5 and 7 in terms of the underlying LTL expressions. An SPath query `e` is evaluated at the document-node `n` with the function `$`, as in, `$(n, e)`, which returns a result-set of nodes in document order. The type of the result set is `Iterable[Node]` such the `n`th item of the iteration appears before the `(n+1)`th item in the order of the document. The syntax for evaluating queries in SPath is borrowed from [jQu].

3.2 Translating attributes.

XPath has an attribute axis and an abbreviated syntax for comparing attribute values. Attributes were not included in the translation and are treated here separately. For

example, `attribute::id eq "1"` selects the attribute nodes, of the context node, that are named `id` and have a value equal to `1`. In abbreviated XPath, this are written as `@id = "1"`.

Elements are defined in SPath as predicates. The query `Element("A")` represents a propositional function that is true for a node that has label `"A"`. In SPath, attributes are defined in a similar way to elements such that `Attribute("id")` represents a propositional function that is true for a node `n` when `n` has an attribute named `"id"`. The `Attribute` class also defines the method:

```
def == :  String => Query
```

that takes a string value and returns a predicate-query. For example, suppose

```
val id = Attribute("id")
```

then, by invoking the method `==` on `id`, using Scala's infix notation as follows,

```
id == "1"
```

returns a predicate that is true for a node `n` when `n` has an attribute named `id` and its value is equal to `"1"`.

Elements and attributes can be combined. For example, the `A` elements that have an `id` attribute with value `"1"` can be specified in SPath as `A(id == "1")`. This is possible because the `Element` class extends `Function[Predicate, Query]` and overrides its `apply` method, returning the conjunction of itself and its argument, i.e. in this example, `A and (id == "1")`. Finally, `Attribute` has a method `@@` that maps a `Node` to its attribute value. For example, `id @@ n` returns `"1"` for the element node `val n = <A id = "1"/>`.

3.3 Translating absolute path expressions.

The syntax for XPath 2.0 in Section 2 has only relative path expressions, i.e. path expressions that begin at the context node. The XPath specification includes absolute path expressions, which start at the root of the document that contains the context node. An absolute path expression in XPath begins with either `/` or `//`. SPath also defines absolute path expressions by accessing the root node from any context node within a document. This is done by using a conditional axis that follows the parent axis from the context node until a node that does not possess a parent i.e. the root node. This traversal from the context node to the root node is expressed in SPath by the query

```
\\(parent, root)
```

where `root` is a predicate that matches the node without a parent. This query applies the `parent` axis-function to the context node until reaching the root node. The absolute XPath expressions `//A` and `/A` are thus coded in SPath as

`\\(parent, root)\\A` and `\\(parent, root)\A`

and are abbreviated by `~\\(A)` and `~\(A)`, respectively.

3.4 Translating the context position and size into SPath.

The syntax for XPath in Figure 1 allows a maximum of one predicate within an `AxisStep`. This differs from XPath, which allows any number of predicates in the form of $[\texttt{Predicate}_1]...[\texttt{Predicate}_n]$. Multiple predicates are particularly useful when specifying the context position. For example, the XPath queries:

```
//A[B and position() = 3]
 //A[B][position() = 3]
```

are distinguished with respect to the document:

```
<x>
 <A id="1"><B/></A>
 <A id="2"><B/></A>
 <A id="3"/>
 <A id="4"><B/></A>
</x>
```

The first query returns an empty result but the second query returns a singleton node-set that contains the node `<a id="4"><b/></a>`. The translations of the two XPath queries above, in SPath are:

```
\\(A)$nth(3)?(\(B))
\\(A?(\(B)))$nth(3)
```

The context function `$nth` returns an instance of the class `AxisStep(f:axis)`, which is a subclass of `Query`. The axis `f` evaluates the query that receives the method invocation at the context node and applies a transformation to the resulting node-set. For example, in the first query, `$nth(3)` returns an axis that evaluates `\\(a)` and creates a view from the third item of the resulting node-set. Instances of `AxisStep(f:axis)` are interpreted within the DSL for SPath, as the query `\(f)`. In general, the user can define a context function for any transformation on ordered node sets with the `$context` method in the class `Query`:

```
$context(Iterable[Node] => Iterable[Node]) :  AxisStep
```

SPath also defines the context functions `$ltrim(n:Int)` and `$rtrim(n:Int)`, which remove the specified number of nodes from the left and right of the context node-set.

Predicates over the context size can also be generated by the overloaded context function:

```
$context(Int => Boolean) :  Predicate
```

which takes a function as an argument that maps the size of the node-set to a `Boolean` value. For example the context function `$size` is defined as follows:

```
$size(i:Int) = $context((s:Int) => s == i)
```

The following XPath query selects the `A`-node descendants that have exactly 3 `B` children:

```
//A[count(B) = 3]
```

and is written in SPath as follows:

```
\\(A)?(\(B)$size(3)).
```

```
val department =
   <department>
      <students>
           <student id="1" name="John"/>
           <student id="2" name="Jill"/>
      </students>
      <courses>
         <course id="1" tid="1" name="Object-Oriented Programming">
            <students>
                 <student id="1"/>
                 <student id="2"/>
            </students>
         </course>
         <course id="2" tid="2" name="Logic">
            <students>
                 <student id="1"/>
            </students>
         </course>
      </courses>
      <tutors>
           <tutor id="1" name="Jane"/>
           <tutor id="2" name="Jake"/>
      </tutors>
   </department>
```

(a) Modeling a many-to-many relationship in XML for the domain *department-of-education*.

```
1   import xspath.XSPathLite
2   object ManyToManyRelationExample extends XSPathLite {
3    val student = Element("student")
4    val students = Element("students")
5    val courses = Element("courses")
6    val course = Element("course")
7    val tutor = Element("tutor")
8    val department = Element("department")
9    val tutors = Element("tutors")
10   val id = Attribute("id")
11   val tid = Attribute("tid")
12   val name = Attribute("name")
13   def courseAxis :  axis = n => n.label match {
14    case student.label => $(n, ~\\(courses)\course ?(\\(student(id == id @@ n))))
15    case tutor.label => $(n, ~\\(courses)\course(tid == id @@ n))
16    case _ => empty
17   }
18   def tutorAxis :  axis = n => n.label match {
19    case student.label => $(n, \(courseAxis)\tutorAxis)
20    case course.label => $(n, ~\\(tutors)\tutor(id == tid @@ n))
21    case _ => empty
22   }
23   def studentAxis :  axis = n => n.label match {
24    case tutor.label => $(n, \(courseAxis)\studentAxis)
25    case course.label =>
26     $(n, ~\(students)\student(id on ~\\(courses)\course(id == id @@ n)\\student))
27    case _ => empty
28   }
29  }
```

(b) A specialised query language for the domain *department-of-education*.

Figure 4

```
1   import xspath.XSPathLite
2   object EmployeesExample extends XSPathLite {
3    val employee = Element("employee")
4    val id = Attribute("id")
5    val mgrId = Attribute("mgrId")
6    val manager :  axis = n => $(n, \(parent)\employee(id == mgrId @@ n))
7    val reports :  axis = n => $(n, \(parent)\employee(mgrId == id @@ n))
8    def main(args:  Array[String]) {
9     val company =
10      <company>
11       <department>
12        <employees>
13           <employee id="1" />
14           <employee id="2" mgrId="1" />
15           <employee id="3" mgrId="1" />
16           <employee id="4" mgrId="2" />
17           <employee id="5" mgrId="2" />
18           <employee id="6" mgrId="3" />
19           <employee id="7" mgrId="3" />
20        </employees>
21       </department>
22      </company>
23    val result1 = $(company, \\(employee(id == "7"))\\(manager, not(mgrId)))
24    println(result1 map id)
25    val result2 = $(company, \\(employee(id == "1"))\reports\\reports)
26    println(result2 map id)
27   }
28  }
```

Figure 5 – A specialised query language for the *employee* domain.

4 Examples.

This section presents two examples of SPath being used to query XML documents in Scala. The examples are written from the perspective of a user of SPath. Both of the examples make use of XML documents that contain attributes that reference the attributes of other elements. This technique is used to reduce the size of XML documents. For example, instead of duplicating an element everywhere it is needed within the data, a reference to its unique identity can be put in place of the element.

Example 4.1 *Querying a many-to-many relation in XML.*

The XML document in Figure 4(a) shows data that models the relationship between students, courses and tutors within a department of education.

A student can enroll on many courses, each course is taught by one tutor and each tutor can teach many courses. In this example, the problem will be to retrieve the implicit relationships from the document, of student-tutor and tutor-student. Figure 4(b) shows the full code listing that extends `XSPathLite` and creates the attribute, element and axis definitions that are domain-specific to the *department-of-education* schema. The class `XSPathLite` is needed for querying Scala's native XML documents and was introduced in Section 3.

The first step is to define the boilerplate for elements and attributes of the document on lines 3 to 12. These are required for writing queries in the axis definitions on lines 13 to 28. Next, the idea is to define the axes for the single-step relationships for

student-course, tutor-course and their inverses. The transitive relations, student-tutor and tutor-student, can then be defined simply, in terms of the single-step relations. First, the course-axis defines the courses of both a student and a tutor on lines 13 to 17. The tutor-axis can be defined in a similar way on lines 18 to 22, except that now, the transitive relationship for the tutors of a student, is simply the composition of the course-axis with the tutor-axis. The student-axis is defined for tutors and courses on lines 23 to 28.

The students of a tutor is defined in a similar way, as the composition of the course-axis with the student-axis, on line 24. The students of a course is defined as the students that appear as listed on the course on line 26. A selected student must join its `id` attribute with a student element beneath the required course. The join is specified by using the `on` method of the `id` attribute. The `on` method returns a predicate that is true for a student that has an entry under the course with a matching `id` attribute.

The following queries can now use these axis-functions, to retrieve the transitive relationships, of John's tutors and Jane's students, as follows:

```
$(department, \\(student(name == "John"))\tutorAxis)
$(department, \\(tutor(name == "Jane"))\studentAxis)
```

Example 4.2 *Querying an employee-manager relationship.*

This is example is based on [Kaya]. The organisation of employees within a company is represented in XML as a tree using identity referencing instead of XML's natural tree structure. The XML document that is being queried in this example is listed on lines 10 to 22, in the complete working example in Figure 5. The `employee` element and its attributes are defined on lines 3 to 5. The `manager` axis on line 6 maps an `employee` to its manager and the `reports` axis maps a manager to its direct reports, on line 7. The query on line 23 shows SPath's until-like application of the `manager` axis, which traverses the document until reaching a node without a `mgrId` attribute, specified by `not(mgrId)`. The second query on line 25 applies the closure of the `reports` axis to the employee with `id 1`, yielding all the employees beneath employee 1.

5 A foundation for SPath.

This sections reviews LTL and presents the formal language for SPath, which is based on LTL. For a detailed account of LTL, the reader is referred to [JGP99].

An LTL formula is generated by the following grammar:

$$\mathtt{e} ::= \mathtt{p} \mid \mathtt{X}\ \mathtt{e} \mid \mathtt{e}\ \mathtt{U}\ \mathtt{e} \mid \mathtt{e} \wedge \mathtt{e} \mid \mathtt{e} \vee \mathtt{e} \mid \neg\ \mathtt{e}$$

An atomic proposition, `p`, is a function that maps each state in a model to a boolean truth value:

$$\mathtt{p} : \ \mathtt{States} \rightarrow \{\mathtt{true}, \mathtt{false}\}$$

The connectives `X` and `U` refer to the future states and the connectives $\wedge$, $\vee$ and $\neg$ are classical logical-connectives for conjunction, disjunction and negation. The meaning of these formulas is given by the satisfaction rules for LTL that determine when a sequence of states, π, satisfies a formula, `e`. The satisfaction relation, $\pi \models \mathtt{e}$, is defined by induction on the structure of formulas. Sequences of states in $\mathtt{States}^*$ are ranged

over by π and modeled as functions from natural numbers (including zero) to states. The tail of π starting at the index i, is written π_i and defined as $\pi_i(j) = \pi(i+j)$. The satisfaction relation for LTL, $\models$, is defined as follows:

$$\begin{array}{lll}
\pi \models \mathrm{p} & \text{iff} & \mathtt{p}(\pi(0)) = \mathtt{true} \\
\pi \models \neg\ \mathtt{e} & \text{iff} & \pi \not\models \mathtt{e} \\
\pi \models \mathtt{e}_1 \wedge \mathtt{e}_2 & \text{iff} & \pi \models \mathtt{e}_1 \text{ and } \pi \models \mathtt{e}_2 \\
\pi \models \mathtt{e}_1 \vee \mathtt{e}_2 & \text{iff} & \pi \models \mathtt{e}_1 \text{ or } \pi \models \mathtt{e}_2 \\
\pi \models \mathtt{X}\ \mathtt{e} & \text{iff} & \pi_1 \models \mathtt{e} \\
\pi \models \mathtt{e}_1\ \mathtt{U}\ \mathtt{e}_2 & \text{iff} & \exists i : \pi_i \models \mathtt{e}_2 \text{ and } \forall j, 0 \leq j < i : \pi_j \models \mathtt{e}_1
\end{array}$$

A proposition, $\mathtt{p}$, is satisfied by the sequence of states, π, whenever $\mathtt{p}$ is true for the first state in π. The rules for logical negation, conjunction and disjunction, define the intuitive meaning of these connectives. The rule for `X` requires that `e` holds for the subsequence, π_1, that starts from the next state. The rule for `U` requires that $\mathtt{e}_1$ holds in zero or more states, from the current state, until a state in which $\mathtt{e}_2$ holds immediately afterwards. The connectives `X` and `U` are suitably known as ne`X`t and `U`ntil.

The sequences, π, that form the model of an LTL system are typically generated by a single binary relation, $\longrightarrow$, over the set of states, `States`. This relation defines the model from the initial state, $\pi(0)$. However, in SPath, the states are nodes of the document-tree and the model may use any number of relations over document-tree nodes in order to generate sequences, π, which form the LTL model. In particular, it is natural to annotate the two formulas that can refer to the future, `X` and `U`, with a relation that determines how to generate the next state in the sequence of states. This is how SPath is defined next.

The syntax of SPath formulas, `e`, is defined as follows:

$$\begin{array}{c}
\mathtt{e} \in \mathtt{Expr} ::= \mathtt{p} \mid \mathtt{X}_\chi\ \mathtt{e} \mid \mathtt{e}\ \mathtt{U}_\chi\ \mathtt{e} \mid \mathtt{e} \vee \mathtt{e} \mid \mathtt{e} \wedge \mathtt{e} \\
\begin{array}{rcl}
\mathtt{p} & : & \mathtt{TreeNodes} \rightarrow \{\mathtt{true}, \mathtt{false}\} \\
\chi & : & \mathtt{TreeNodes} \rightarrow \mathcal{P}(\mathtt{TreeNodes})
\end{array}
\end{array}$$

The syntax assumes two sets of countably infinite identifiers for propositional functions (ranged over by `p`) and axis-identifiers (ranged over by χ). An SPath formula, `e`, is a negation-free[3] LTL formula with alterations to both of the standard LTL connectives that refer to the future, `X` and `U`. These formulas are annotated by axis-identifiers, ranged over by χ, each defining an axis-function. In SPath, an axis is modeled as a function from tree-nodes to sets of tree-nodes. The rules that determine when a sequence of tree-nodes satisfies an SPath formula are the standard LTL rules with additional constraints on the rules for the logical connectives `X` and `U`, as follows:

$$\begin{array}{lll}
\pi \models \mathtt{X}_\chi\ \mathtt{e} & \text{iff} & \pi(1) \in \chi(\pi(0)) \text{ and } \pi_1 \models \mathtt{e} \\
\pi \models \mathtt{e}_1\ \mathtt{U}_\chi\ \mathtt{e}_2 & \text{iff} & \exists i : \pi_i \models \mathtt{e}_2 \text{ and } \forall j, 0 \leq j < i : \pi_j \models \mathtt{e}_1 \text{ and } \pi(j+1) \in \chi(\pi(j))
\end{array}$$

The rule for $\mathtt{X}_\chi\ \mathtt{e}$ requires that the second tree-node of π, $\pi(1)$, follows its predecessor, $\pi(0)$, along the axis χ. The rule for $\mathtt{e}_1\ \mathtt{U}_\chi\ \mathtt{e}_2$ has a similar requirement for all positions that satisfy $\mathtt{e}_1$.

A consequence of the language defined this far is that conflicts can occur between several axis-functions on a single sequence of tree-nodes, π, during the application of the satisfaction rules. For example, in the following formula, the rules require both of

[3]Negative queries are recovered in the Scala DSL for SPath in Section 7.

$$
\begin{array}{rcl}
\texttt{SPath(p)} & = & \texttt{true} \\
\texttt{SPath(e}_1 \vee \texttt{e}_2\texttt{)} & = & \texttt{SPath(e}_1\texttt{)} \;\&\; \texttt{SPath(e}_2\texttt{)} \\
\texttt{SPath(e}_1 \wedge \texttt{e}_2\texttt{)} & = & (\texttt{SPath(e}_1\texttt{)} \;\&\; \texttt{Axes(e}_2\texttt{)} = \emptyset) \text{ or } (\texttt{SPath(e}_2\texttt{)} \;\&\; \texttt{Axes(e}_1\texttt{)} = \emptyset) \\
\texttt{SPath(X}_\chi\ \texttt{e)} & = & \texttt{SPath(e)} \\
\texttt{SPath(e}_1\ \texttt{U}_\chi\ \texttt{e}_2\texttt{)} & = & \texttt{SPath(e}_2\texttt{)} \;\&\; \texttt{Axes(e}_1\texttt{)} = \emptyset
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{Axes(p)} & = & \emptyset \\
\texttt{Axes(e}_1\ \vee\ \texttt{e}_2\texttt{)} & = & \texttt{Axes(e}_1\texttt{)} \cup \texttt{Axes(e}_2\texttt{)} \\
\texttt{Axes(e}_1\ \wedge\ \texttt{e}_2\texttt{)} & = & \texttt{Axes(e}_1\texttt{)} \cup \texttt{Axes(e}_2\texttt{)} \\
\texttt{Axes(X}_\chi\ \texttt{e)} & = & \{\chi\} \cup \texttt{Axes(e)} \\
\texttt{Axes(e}_1\ \texttt{U}_\chi\ \texttt{e}_2\texttt{)} & = & \{\chi\} \cup \texttt{Axes(e}_1\texttt{)} \cup \texttt{Axes(e}_2\texttt{)}
\end{array}
$$

Figure 6 – SPath formulas without branching conflicts.

the axes χ and χ' to agree on the next tree-node in π from the current state:

$$\texttt{X}_\chi\ \texttt{e}_1\ \wedge\ \texttt{X}_{\chi'}\ \texttt{e}_2$$

A similar conflict can occur with the formula, $\texttt{e}_1\ \texttt{U}_\chi\ \texttt{e}_2$, between the axis χ and the axes occurring in $\texttt{e}_1$. These conflicts are avoided by considering only the formulas for which conflicts cannot occur. A formula, `e`, is conflict-free whenever $\texttt{SPath(e)} = \texttt{true}$. The function `SPath` maps formulas, `e`, to $\{\texttt{true}, \texttt{false}\}$ and is defined in Figure 6.

`SPath` places constraints on the two problem cases for $\wedge$ and $\texttt{U}_\chi$ by restricting the occurrences of axes, using an auxiliary function called `Axes`. This function returns the set of axis-identifiers occurring within an SPath formula. It is now possible to define the query-evaluation algorithm, in the following section, for conflict-free formulas with the guarantee that only one axis is applicable in each state.

Figure 7 shows an extension of `Expr`, called `SugaredExpr`, which adds location steps to SPath. A translation from `SugaredExpr` to SPath expressions, `Expr`, is defined by the function $[\![\bullet]\!]$. The crux of the translation rests on the insertion function, `insert`. Here, $\texttt{insert(e}_1\texttt{, e}_2\texttt{)}$ inserts $\texttt{e}_1$ into $\texttt{e}_2$'s rightmost, innermost position beneath the temporal operators `X` and `U`. The top-down translation of sugared expressions is necessary for constructing de-sugared SPath expressions in the fluent-interface pattern in Section 7. Applying the translation to the sugared SPath query

$$\texttt{(\textbackslash(}\chi_1\texttt{, A)\textbackslash\textbackslash(}\chi_2\texttt{, B))\textbackslash(}\chi_3\texttt{, C)}$$

yields the de-sugared SPath expression:

$$\texttt{X}_{\chi_1}\ \texttt{(A}\ \wedge\ \texttt{(*}\ \texttt{U}_{\chi_2}\ \texttt{(B}\ \wedge\ \texttt{X}_{\chi_3}\ \texttt{C)))}$$

6 Query evaluation in SPath.

SPath makes direct use of the automaton construction algorithm of [GPV+95] to evaluate queries. The full construction algorithm that is used in SPath is presented in Appendix B. The number of states in the automaton construction, $\mathcal{A}$, for a query `e` is written $|\mathcal{A}|$ and is $O(2^{|\texttt{e}|})$ where $|\texttt{e}|$ is the size of `e` [GPV+95]. The exponential growth of automaton states is caused by splitting states with the formulas `e U e'` or `e` $\vee$ `e'`. For example, the following queries generate automata with a number of states that is exponential in k:

```
s ∈ SugaredExpr  ::=  Expr
                  |   \(χ, s)
                  |   \\(χ, s)
                  |   s \(χ, s)
                  |   s \\(χ, s)

              ⟦•⟧   :   SugaredExpr ⟶ Expr
        ⟦\(χ, s)⟧   =   X_χ ⟦s⟧
       ⟦\\(χ, s)⟧   =   * U_χ ⟦s⟧    where * = n => true
   ⟦s1 \(χ, s2)⟧   =   insert (X_χ ⟦s2⟧, ⟦s1⟧)
  ⟦s1 \\(χ, s2)⟧   =   insert (* U_χ ⟦s2⟧, ⟦s1⟧)
              ⟦e⟧   =   e

          insert   :   Expr × Expr ⟶ Expr
   insert (e, p)   =   p ∧ e
insert (e, X_χ e1) =   X_χ insert (e, e1)
insert (e, e1 U_χ e2) = e1 U_χ insert (e, e2)
insert (e, e1 ∧ e2) =  e1 ∧ insert (e, e2)
insert (e, e1 ∨ e2) =  insert (e, e1) ∨ insert (e, e2)
```

Figure 7 – Syntactic sugar for location path steps

$$\texttt{(e}_1\ \texttt{U}\ \texttt{e}_1\texttt{')} \wedge \ldots \wedge \texttt{(e}_k\ \texttt{U}\ \texttt{e}_k\texttt{')}$$

$$\texttt{(e}_1\ \vee\ \texttt{e}_1\texttt{')} \wedge \ldots \wedge \texttt{(e}_k\ \vee\ \texttt{e}_k\texttt{')}$$

However, the first form of query is excluded from SPath. The second form of query generates a state for each truth assignment that satisfies the query. The second form of query is allowed by SPath. An advantage of using the automaton construction algorithm of [GPV+95], in SPath, is that it provides a natural solution for implementing the evaluation of conditional axes [Mar04b]. For example, the conditional axis, `e` $\texttt{U}_\chi$ `e'`, applies χ repeatedly until `e'`. Furthermore, query evaluation by an explicit-state enumeration enables caching of visited states, thereby terminating on queries such as `\\(self)` [4].

Query evaluation in SPath constructs the product-automaton $\mathcal{A} \times \mathcal{D}$, on-demand. The complexity of model checking for LTL is proportional to the size of the product automaton and in the worst case, takes time $O(|\mathcal{D}| \times 2^{|\texttt{e}|})$ [GPV+95, LP85] where $|\mathcal{D}|$ is the size of the document $\mathcal{D}$. Useful queries typically generate small automata so the exponential blow-up of the number of states of the automaton construction for a query is in practice not as severe as the worst-case. A state within the product-automaton is a pair, `(q, n)`, consisting of an automaton-state `q` of $\mathcal{A}$ and a node `n` of the document. In SPath, π is a finite path of tree-nodes, so the accepting condition of the automata $\mathcal{A}$ is not the Büchi acceptance condition for infinite runs [Hol03]. For SPath, the accepting states of the automaton are all of the states, `q`, which have `q.next` $= \emptyset$, i.e. no formulas hold in the successors of `q`. These states are guaranteed to exist by Proposition 6.1.

[4] Another example is the evaluation of the query `\\(`χ`)` on the document `<A><B/></A>` where `val` χ `= n => if n.label == "A" $(n, \(B)) else $(n, \(parent, A))`.

```
1   def evaluate(map:  Map[Node, Iterable[T]],
                 result:  ListBuffer[T], cache:  Cache):  Iterable[T] = {
2    if (map.keys.size == 0)
3     return distinct(result)
4    val newMap = HashMap[Node, Iterable[T]]()
5    for ((q,ns) <- map.iterator) {
6     if (q.isFinalNode)
7       result ++= ns
8     else {
9      val ns2 = distinct(ns flatMap q.uniqueAxis)
10     for (q2 <- q.outgoing) {
11      val ns3 = ns2 filter(o => !cache.seen(q2, o) && q2.isSatisfiedBy(o))
12      if (ns3.size > 0) {
13       newMap += q2 -> ns3
14       cache remember(q2, ns3)
15      }
16     }
17    }
18   }
19   evaluate(newMap, result, cache)
20  }
```

Figure 8 – A standard breadth-first search for the evaluation of SPath queries.

Proposition 6.1 *For any SPath formula* e*, there exists a state* q *in the automaton construction for* e *such that* $q.next = \emptyset$*.*

By Proposition 6.2, each state, q, in the automaton construction determines a unique axis from the future-time subformulas in `q.old` - i.e. the subformulas that hold at the state q. The unique axis is used to generate the next document-tree nodes for the generation of $\mathcal{A} \times \mathcal{D}$.

Proposition 6.2 *For any formula* e*, if* $SPath(e)$ *then* $|\Theta(q)| \leq 1$ *and* $|q.next| \leq 1$ *for every state* q *in the automaton construction for* e *where*

$$\begin{aligned} \Theta(q) \quad = \quad & \{X_\chi\, e \mid X_\chi\, e \in q.old\} \cup \\ & \{e_1\, U_\chi\, e_2 \mid e_1\, U_\chi\, e_2 \in q.old \,\&\, e_2 \notin q.old\} \end{aligned}$$

Definition 6.3 *An automaton construction* $\mathcal{A}$ *accepts* π *on the run* $q_0 \ldots q_k$ *when* $q_k.next = \emptyset$*; and for* $k \geq i \geq 0$*,* $p \in q_i.old$*:* $p(\pi(i)) = true$*; and for* $k > i \geq 0$*:* $\pi(i+1) \in \chi(\pi(i))$ *where* χ *is the unique axis in* q_i*.*

Corollary 6.4 follows from the correctness proof of [GPV+95].

Corollary 6.4 *The automaton* $\mathcal{A}$*, constructed for the SPath query* e*, accepts exactly the same finite sequences of tree-nodes that satisfy* e*.*

The product-automaton is constructed in SPath using a standard breadth-first search algorithm, which is shown in Figure 8. A state of the automaton, $\mathcal{A}$, is represented by the class `Node`, which has the following methods:

```
isFinalNode :  Boolean
uniqueAxis :  axis
outgoing :  Iterable[Node]
isSatisfiedBy(o :  T) : Boolean
```

The method `q.isFinalNode` is true when the node `q` is an accepting state of the automaton according to Proposition 6.1. The method `uniqueAxis` returns the unique axis of the node according to Proposition 6.2. These methods run in constant time. The method `q.outgoing` returns a collection of automaton nodes such that if `q2` $\in$ `q.outgoing` then `q` $\longrightarrow$ `q2` is an edge in the automaton $\mathcal{A}$. The expression `q.isSatisfiedBy(o)` is true when the document node `o` satisfies all of the propositions in `q.old`, according to Definition 6.3.

The method `distinct(it:  Traversable[T]): Iterable[T]` returns a new collection containing unique document nodes from the argument, `it`. SPath determines unique document nodes by wrapping a node `o:T` in an instance of `IdentityWrapper(o)`, which overrides `equals` based on reference equality in Scala, `o eq o'`; and overrides `hashCode` using `java.lang.System.identityHashCode`.

The evaluation algorithm also makes use of a cache for storing visited states of the product automaton. The class `Cache` has two methods,

```
seen(q:  Node, o :  T) : Boolean
remember(q:Node, it :  Iterable[T])
```

The method `seen(q, o)` is true when the cache contains the state of the product automaton, `(q, IdentityWrapper(o))` and runs in constant time. The method `remember(q, ns)` stores the states `(q, IdentityWrapper(o))` for all `o` $\in$ `ns`, into the cache and runs in O(`ns.size`) time.

The tail-recursive function `evaluate`, is parameterised by the abstract type, `T`, which is the type of the tree-nodes. This function has three arguments. The first argument is a map from states of the automaton to sets of tree-nodes, representing the frontier of the search of $\mathcal{A} \times \mathcal{D}$ i.e if `(q, ns)` $\in$ `map` and `n` $\in$ `ns` then `(q, n)` $\in$ $\mathcal{A} \times \mathcal{D}$. The second argument is a result-set of tree-nodes. The third argument is a cache of visited states of $\mathcal{A} \times \mathcal{D}$. The function terminates on line `3` when the map is empty and the recursive call is on line `19`. The idea behind the algorithm is to advance the frontier of the search by one step within one call of `evaluate`. Each state in the map is advanced by one step on lines `8` to `17`. If the state of the automaton, `q`, is an accepting state then its tree-nodes are added to the result-set on line `7`. Otherwise the unique axis-function is applied to the state's document-nodes, `ns`, on line `9`. The set of document-nodes that is obtained from applying the axis-function, are then matched against each automaton-state, `q2`, on an outgoing edge from the current state `q`, on line `11`. The new states of the product-automaton are created on line `13`. The cache prevents each state being visited more than once along the depth of the search, on lines `11` and `14`. The selection of distinct states on line `9` prevents each state from being visited more than once, along the breadth of the search.

The worst-case running-time of query evaluation in SPath needs to take into account *higher-order* axis and propositional functions i.e. functions that evaluate queries. These kinds of propositions and axes were not represented in the formal syntax of SPath but they occur frequently in the examples.

Informally, the degree of an SPath query, axis or propositional function is the number of calls to `evaluate` that can occur on the function-call stack during the evaluation of the query, axis or propositional function. To make this definition precise,

axes and propositional functions will now be modeled to represent the nesting of query evaluation in SPath.

A statement `S` represents a code block in the host language Scala. A statement consists of a code block `b`, which does not evaluate a query; an evaluation of a query `e` at the tree-node `o`, `$(o, e)`; or the sequential composition of statements, `S;S'`. Propositional and axis-functions are now represented by a pair, consisting of an object variable `o` and a statement `S`:

$$\begin{array}{rcl} \texttt{p},\ \chi \in \texttt{UserFunctions} & ::= & \texttt{o => S} \\ \texttt{S} \in \texttt{Statements} & ::= & \texttt{b} \mid \texttt{\$(o, e)} \mid \texttt{S;S} \end{array}$$

The queries occurring in a block of Scala code is defined by the function `Queries` as follows:

$$\begin{array}{rcl} \texttt{Queries (b)} & = & \emptyset \\ \texttt{Queries (\$(o, e))} & = & \{\texttt{e}\} \\ \texttt{Queries (S;S')} & = & \texttt{Queries(S)} \cup \texttt{Queries(S')} \end{array}$$

These definitions assume that the number of calls to `$` at runtime in a code block `S` is statically determined by the size of `S`.

Definition 6.5 *The degree of a query, axis or propositional function is written* $\Phi(e)$, $\Phi(\chi)$ *and* $\Phi(p)$ *respectively and defined as:*

$$\begin{array}{rcll} \Phi(o \texttt{ => } S) & = & 0 & \mathit{Queries(S)} = \emptyset \\ \Phi(o \texttt{ => } S) & = & \mathit{Max}\ \{\Phi(e) \mid e \in \mathit{Queries(S)}\} & \mathit{Queries(S)} \neq \emptyset \\ \Phi(e) & = & 1 + \mathit{Max}\ \{\Phi(o \texttt{ => } S) \mid o \texttt{ => } S \in \mathit{UF(e)}\} & \end{array}$$

where `UF(e)` *are the user functions occurring directly in* `e`*:*

$$\begin{array}{rcl} \mathit{UF}\ (p) & = & \{p\} \\ \mathit{UF}\ (X_\chi\ e) & = & \{\chi\} \cup \mathit{UF(e)} \end{array} \qquad \begin{array}{rcl} \mathit{UF}\ (e \vee e') & = & \mathit{UF}\ (e) \cup \mathit{UF}\ (e') \\ \mathit{UF}\ (e \wedge e') & = & \mathit{UF}\ (e) \cup \mathit{UF}\ (e') \\ \mathit{UF}\ (e\ U_\chi\ e') & = & \{\chi\} \cup \mathit{UF(e)} \cup \mathit{UF(e')} \end{array}$$

Definition 6.6 *The size of a query expression* `e` *and a statement* `S` *is defined by* $|e|$ *and* $|S|$ *respectively, as follows:*

$$\begin{array}{rcl} |X_\chi\ e| & = & |\chi| + |e| \\ |e\ U_\chi\ e'| & = & |\chi| + |e| + |e'| \\ |e \vee e'| & = & |e| + |e'| \\ |e \wedge e'| & = & |e| + |e'| \\ |o \texttt{ => } S| & = & 1 + |S| \end{array} \qquad \begin{array}{rcl} |b| & = & 0 \\ |\$(o,\ e)| & = & |e| \\ |S; S'| & = & |S| + |S'| \end{array}$$

Proposition 6.7 *The worst-case running-time of query evaluation in SPath is*

$$O((2^{|e|} \times |\mathcal{D}|)^3 \times |e|^2)^{\Phi(e)}$$

Proposition 6.8 *The worst-case space complexity of query evaluation in SPath is*

$$O(2^{|e|} \times |\mathcal{D}|^2 \times \phi(e))$$

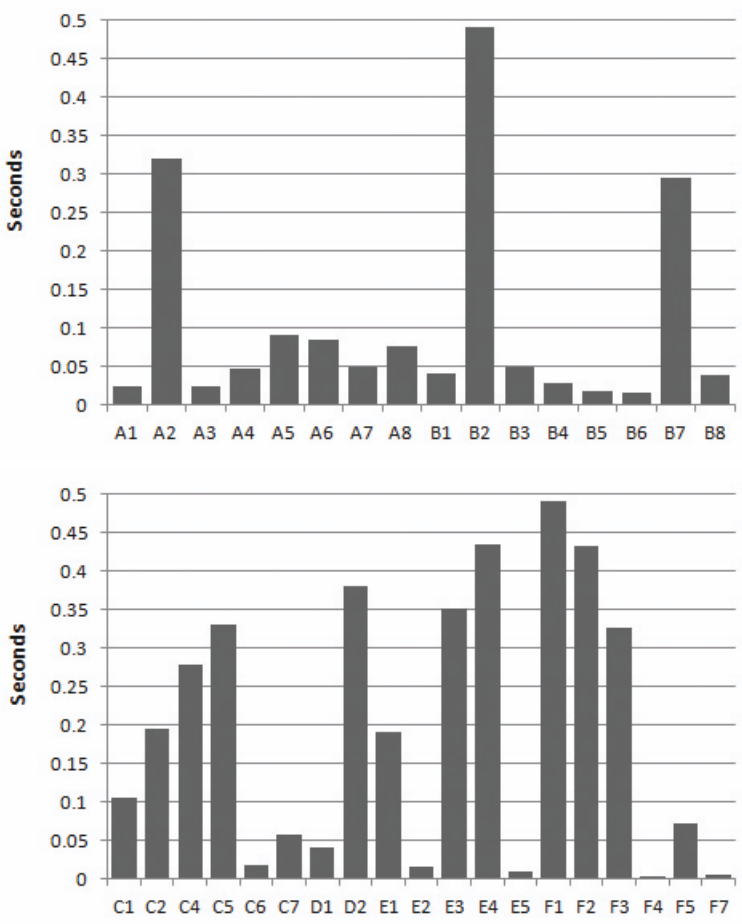

Figure 9 – Running times of evaluating SPath queries based on the XPathMark performance test. The queries ran on an AMD Phenom 9650 Quad-Core Processor 2.30 GHz. 8.00 GB Memory(RAM)

6.1 Experiments.

SPath has been tested on 35 practical queries, which are based on a subset of the XPathMark performance test [Fra07]. The XPathMark performance test has 6 sets of queries, A to F. Sets A to E consist of XPath queries and set F contains XQuery programs that compute closures with user-defined functions. Set F has been implemented in SPath with conditional axes. The domain of the test is an auction website [SWK+02]. The main entities within the domain model are open auctions, bids, persons, regions, categories and items. The document was created by the XMark data generation tool [xma] with a scale factor of 0.02. The document size is 2.37 MB, containing 95,392 nodes and 7,384 attributes. The evaluation times of the queries are shown in Figure 9. The evaluation times show that SPath is usable on practical queries and that the exponential growth of SPath's search space has been contained. The queries of the XPathMark performance test consist mostly of linear paths, which do not generate exponentially sized automata in SPath.

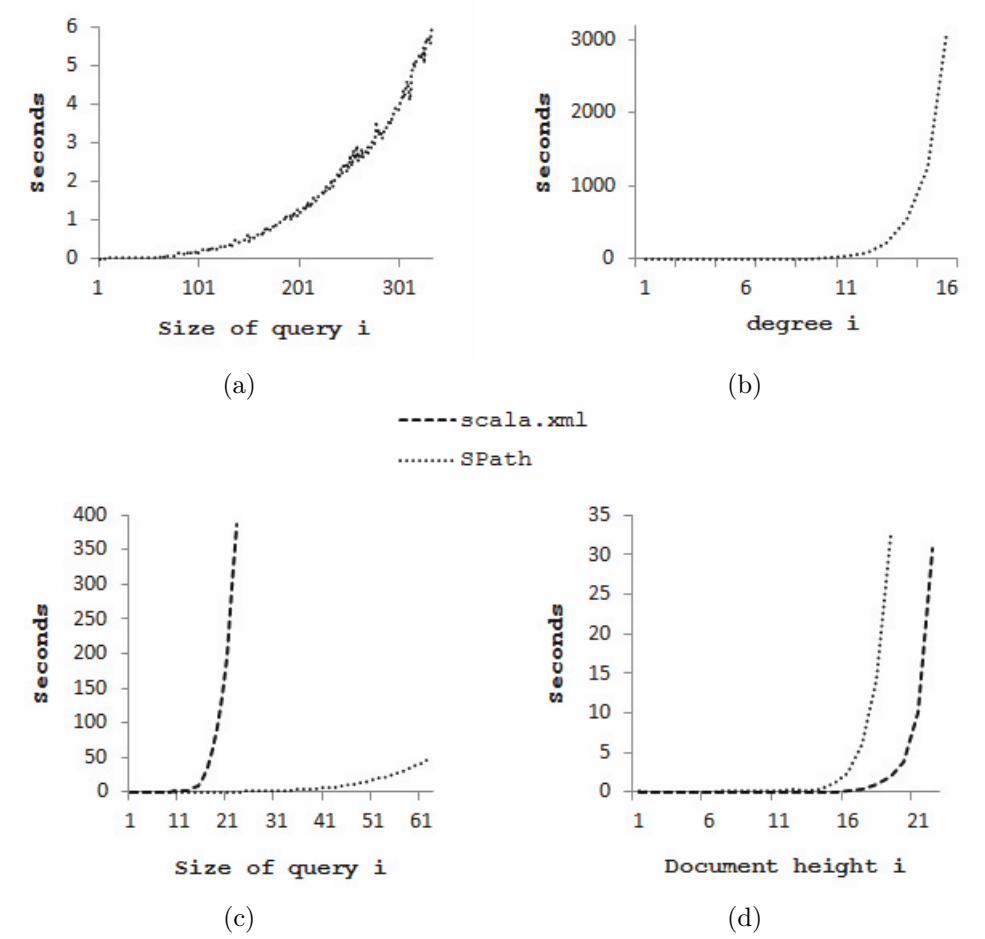

Figure 10 – Running times of evaluating SPath queries (a), (b) and a comparison between SPath and `scala.xml` (c) and (d).

A separate experiment was done to show how query-evaluation scales in SPath, with a comparison to Scala's XML API. The first kind of queries to be evaluated are based on [GKP02], which were shown to cause exponential growth in earlier XPath engines. The SPath version of the queries are formed of a basic query that makes two steps, first to the children and then back up to the parents, as follows

```
\(*)\(parent, *)
```

The basic query is repeated `i` times:

$$\underbrace{\texttt{\textbackslash(*)\textbackslash(parent, *)}\ldots\texttt{\textbackslash(*)\textbackslash(parent, *)}}_{\texttt{i}\times}$$

and is evaluated on the simple document:

```
<A><B/><B/></A>
```

The running time of this query is shown in Figure 10(a), and scales better in the size of the query than naive evaluation, which results in exponential growth [GKP02]. This

query is similar to the queries B11 to B15 of the XPathMark performance test, which also repeat 2 location steps `i` number of times. These examples may seem contrived but they do make a practical point.

The apparent cause of the exponential growth of the naive algorithm is that it allows duplicate intermediate results during evaluation. Consider the following natural language query on the XMark domain model: the names of categories that have an item being sold in an auction that someone is watching. This query may be written in XPath as :

```
/site/people/person/watches/watch/id(open_auction)/itemref/id(item)/incategory/
                                                              ↪ id(category)/name
```

and in SPath as:

```
site\people\person\watches\watch\watch\$id(open_auction)\itemref\$id(item)\incategory\
                                                              ↪ $id(category)\name
```

Suppose that the naive algorithm evaluates this query on a document that contains 10 people who are all watching the same 10 auctions and each auction sells an item that belongs to each of the same 10 categories. The resulting node set of categories would contain 10^3 nodes that are duplicates of a single category. The final result set therefore contains a single name. In this query, each intermediate result after a location step really contains a single node, which can be obtained by selecting distinct intermediate nodes, as in line `9` of SPath's evaluation algorithm in Figure 8. Without `distinct` on line `9`, SPath would perform like the naive algorithm.

The second kind of SPath queries that are evaluated in Figure 10(b) shows that query-evaluation scales exponentially in the degree of the query. The query being evaluated is $\backslash(x_i)$, where the axis x_i is defined as follows:

$$\begin{aligned} x_0 &= n \Rightarrow \$(n, \backslash\backslash(*)) \\ x_{i+1} &= n \Rightarrow \$(n, \backslash(x_i)\backslash(x_i)) \end{aligned}$$

The queries were evaluated on a document containing 9 element nodes.

Neither of the first two queries have a direct equivalent in Scala's XML API and so it is not possible to create a comparison.

The third query, evaluated in Figure 10(c), can be written in both SPath and `scala.xml` and is the following in `scala.xml`:

$$\texttt{doc}\ \underbrace{\backslash\backslash\text{`\_'}\ldots\backslash\backslash\text{`\_'}}_{i\times}$$

and in SPath:

$$\$(\texttt{doc},\ \underbrace{\backslash\backslash(*)\ldots\backslash\backslash(*)}_{i\times})$$

Both queries apply the `descendant-or-self axis` `i` times, beginning at the node `doc` - the root of an XML document containing 9 element nodes. The size of the result-set of the `scala.xml` query increases with `i` but the result-set of the SPath query is constantly of size 9, as `i` increases.

The fourth and final query evaluated in Figure 10(d) is `doc \\"A"` in `scala.xml` and `\\(A)` in SPath for the element `A`. These queries were both evaluated on an XML document representing a full binary tree of height `i`, in which each `A` element that is a parent has 2 children elements labeled `A`. The running times scale with the document size 2^i, although SPath rises sooner than `scala.xml`.

7 An embedded DSL for SPath in Scala.

This section explains the embedded DSL in terms of its implementation and characteristics that are shared with typical internal DSL implementations, such as those described in [Fow10]. In particular, SPath has been implemented with the fluent interface and expression-builder patterns. These patterns have been coined by Martin Fowler to describe common techniques for implementing embedded DSLs. SPath's embedded DSL benefits from Scala's syntax and language features, such as closures, infix notation, symbolic identifiers, traits, mixin composition, objects-as-functions, implicit conversions and native XML support.

Figure 11 shows a partial code-listing of the trait `QueryExpression`, which implements the SPath expressions `Expr` and `SugaredExpr`. Lines `8` to `34` implement `Expr` as case classes of the class `Query`. The inner class `Query` implements the insertion function, from Figure 7, on lines `12` to `19`. The two expression from `SugaredExpr`, `s\\(χ, s')` and `s\(χ, s')`, are implemented as methods of the class `Query`, on lines `20` and `21`. The remaining two expressions, `\\(χ, s)` and `\(χ, s)`, are implemented at the top level, on lines `36` and `37`.

The SPath operator `\->`, defined on lines `27` and `42`, is like an iterator for SPath. The expression `\->(f, e)` matches the next document nodes along the axis `f` that satisfy `e`.

The trait `QueryExpression` implements a variant of the fluent-interface pattern of [Fow10]. A fluent interface typically breaks the command-query design principle of [Mey88]. In this principle, only command methods may change the abstract state of objects, whereas query methods have no side-effects on abstract state. An abstract state refers to the observable state of an object, through its interface. However, in a typical fluent interface, each method returns the receiving object (a query) and changes its abstract state (a command). In SPath, each method of the fluent interface, builds and returns a new `Query`, which contains the current receiving object of the method call. The newly created `Query` object is then used for building even larger expressions. In this way, SPath *chains* method calls for building query expressions.

The expression-builder pattern, as described in [Fow10] sometimes separates the methods of the fluent interface, from the model, by using a builder class that wraps the model. In SPath however, the builder methods are defined directly in the `Query` class. This does not mix command-query and fluent interfaces because the execution code for the model i.e. the query evaluator, is defined in the trait `LtlAlgorithm`, which is described below.

In [GKP02, Mar04a, CDGLV09], the core axes of XPath are defined in terms of a minimal set of relations on tree nodes, e.g. the axes can be defined in terms of `right-sibling` and `child`. Here `right-sibling` returns a single node that is the immediate sibling to the right of the context node. This axis does not exist in XPath but enables a simple definition of XPath's core axes. Figure 12 shows the core axes defined as regular expressions over XPath expressions, assuming the axes `self`, `child` and `right-sibling`. SPath's implementation of XPath's core axes is based on the definitions in Figure 12.

Figure 13 shows a partial code-listing of the trait `SPath`. The trait `SPath` is an abstract class that cannot be directly instantiated. The core axes are defined as SPath queries on lines `10` to `17`. These definitions depend only on the abstract methods, `parent` and `children` on lines `2` to `3`. The implementations of `parent` and `children` depend on the particular API for the document-trees. The type of the nodes of the document-trees are parameterised by the parametric type `T` in all of SPath's abstract

```
trait QueryExpression[T] {
 type axis = T => Iterable[T]
 type predicate = T => Boolean
 def *= Predicate(_ => true)
 def defaultAxis: axis
 def not : Query => Query
 def exists : Query => Query
 class Query {
  def and(e: Query) = And(this, e)
  def or(e: Query) = Or(this, e)
  def U(f: axis, e: Query) = Until(f, this, e)
  def insert(e : Query) : Query =
    this match {
     case And(e1, e2) => And(e1, e2 insert e)
     case Or(e1, e2) => Or(e1 insert e, e2 insert e)
     case Until(f2, e1, e2) => Until(f2, e1, e2 insert (e))
     case X(f1, e1) => X(f1, e1 insert (e))
     case _ => this and e
    }
  def \\(f: axis, e: Query) : Query = this insert(*U (f, e))
  def \(f: axis, e: Query) : Query = this insert(X (f, e))
  def \\(f : axis) : Query = \\(f, *)
  def \(f : axis) : Query = \(f, *)
  def \\(e : Query) : Query = \\(defaultAxis, e)
  def \(e : Query) : Query = \(defaultAxis, e)
  def ?(e : Query) : Query = this insert exists(e)
  def \->(f : axis, e : Query) = this insert X(f, not(e) U (f, e))
 }
 case class Predicate(p: T => Boolean) extends Query {
  def evaluate(n : T) = p(n)
 }
 case class And(val l: Query, val r: Query) extends Query
 case class Or(val l: Query, val r: Query) extends Query
 case class Until(f: axis, val l: Query, val r: Query) extends Query
 case class X(f: axis, val next: Query) extends Query
 def \\(f: axis, e: Query): Query = *U (f, e)
 def \(f: axis, e: Query): Query = X(f, e)
 def \\(f: axis): Query = \\(f, *)
 def \(f: axis): Query = \(f, *)
 def \(e: Query): Query = \(defaultAxis, e)
 def \\(e: Query): Query = \\(defaultAxis, e)
 def \->(f : axis, e : Query) = X(f, not(e) U (f, e))
}
```

Figure 11 – The representation of SPath queries in Scala.

```
self
child
right-sibling
parent               =  child^-1
left-sibling         =  right-sibling^-1
descendant           =  child::*(/child::*)*
descendant-or-self   =  self::*(/child::*)*
ancestor             =  parent::*(/parent::*)*
ancestor-or-self     =  self::*(/parent::*)*
following-sibling    =  right-sibling::*(/right-sibling::*)*
preceding-sibling    =  left-sibling::*(/left-sibling::*)*
following            =  (/parent::*)*(/right-sibling::*)+ (/child::*)*
preceding            =  (/parent::*)*(/left-sibling::*)+ (/child::*)*
```

Figure 12 – The core axes of XPath, defined in terms of regular expressions over XPath queries, assuming the primitive axes `self`, `child` and `right-sibling`.

classes e.g. `QueryExpression`, `SPath` and `LTLAlgorithm`.

A user of the trait `SPath` therefore needs to do three things: instantiate the parametric type `T` for the chosen document-tree and implement the abstract methods `parent` and `children`. This usage pattern is covered in more detail in Section 7.2.

The evaluation function `$`, defined on lines 22 to 32, first checks that the query is free from branching conflicts, according to Figure 6. It then constructs the automaton for `e` by calling the function `buildAutomaton`, defined in `LtlAlgorithm`, but not listed in this article. The evaluation function $ returns a function that maps a tree-node to the result-set of the query. The result-set is sorted by document order on line 29 when the evaluation is not nested inside another evaluation i.e. when the counter, `depth`, reaches zero.

The syntax for SPath in Section 5 does not include negative queries. Therefore, negation is introduced into the embedded DSL on line 33 of the trait `SPath` by the function, `not`. This function takes a query and returns a `Predicate` that is true when the result of evaluating the query is empty. The function, `exists` on line 34 is defined in a similar way, but is true when the result is not empty. The function `?` of class `Query`, is SPath's analog of XPath's predicate, and is defined in terms of `exists`, on line 26 of Figure 11.

7.1 Extending SPath's fluent interface.

Adding new methods to the `Query` class is possible by using Scala's implicit conversions. For example, a new operation, `\\+`, which is syntactic sugar for one-or-more steps along the child-axis is defined in a new class `EnhancedQuery`:

```
class EnhancedQuery(q:Query) {
 def \\+ (q2 :  Query) = q\(child)\\(child, q2)
}
```

An implicit conversion from `Query` to `EnhancedQuery` is defined as follows:

```
implicit def enhancedQuery(q:Query) = new EnhancedQuery(q)
```

An SPath query can now be written with the new operation: `\(A)\\+ B`

```
1   trait SPath[T <: AnyRef]
    extends QueryExpression[T] with LltAlgorithm[T] {
2    def parent: axis
3    def children: T => IndexedSeq[T]
4    override def defaultAxis = children
5    def ?(p: predicate) = Predicate(p)
6    val leftSibling : axis = n => sibling(position(n) - 1)(n)
7    val rightSibling : axis = n => sibling(position(n) + 1)(n)
8    val self : axis = n => List(n)
9    val child = children
10   val descendant = $(\(child)\\child)
11   val descendantOrSelf = $(\\(child))
12   val ancestor = $(\(parent)\\parent)
13   val ancestorOrSelf = $(\\(parent))
14   val followingSibling = $(\(rightSibling)\\rightSibling)
15   val precedingSibling = $(\(leftSibling)\\leftSibling)
16   val following = $(\\(parent)\rightSibling\\rightSibling\\child)
17   val preceding = $(\\(parent)\leftSibling\\leftSibling\\child)
18   def root : Predicate = ?(n => parent(n).size == 0)
19   def ~\\ (e : Query)= \\(parent, root)\\e
20   def ~\ (e : Query)= \\(parent, root)\e
21   def $(n : T, e: Query) : Iterable[T] = $(e)(n)
22   def $(e: Query) : T => Iterable[T] = {
23    if (!SPath(e))
24     throw new Exception("SPath expression contains branching conflicts.")
25    val q = buildAutomaton(e)
26    (o:T) => {
27     val r = q(o);
28     if (depth == 0)
29      documentOrder(r, o)
30     else r
31    }
32   }
33   override def not = (e : Query) => ?(n => $(n, e).size == 0)
34   override def exists = (e : Query) => ?(n => $(n, e).size > 0)
35  }
```

Figure 13 – Partial listing of trait SPath.

7.2 Instantiating SPath with concrete documents.

`SPath` uses the template pattern [GHJV95] and Scala's parametric polymorphism to enable SPath to be reused with any kind of sibling-ordered, tree data-structure. The abstract class `SPath` must therefore be extended before querying specific types of trees. `SPath` is concretised by instantiating its parametric type `T` with the type of nodes of a specific document API, and implementing both of its abstract methods, `children` and `parent`. The implementation of these abstract methods require the API's tree-nodes to have pointers to their children *and* parent nodes. Concrete extensions of `SPath` are immediately ready for querying the selected type of documents.

However, to enable SPath to be used with tree data-structures without pointers to parent nodes, a further abstract extension to `SPath` has been created. This abstract extension of `SPath` is called `SPathLite`, because it is lighter in terms of its dependencies on abstract methods that need to be overridden by users. A concrete extension of `SPathLite` needs only to override the abstract `children` method of `SPath` and instantiate the parametric type `T`. `SPathLite` has its own implementation of the `parent` method.

`SPathLite` does some book keeping to generate the parent relation. The missing parent-relation can be generated completely before evaluation or on-the-fly during the evaluation. The latter is possible since when evaluation starts at the root of the document, every evaluation-step along the parent-axis from a node `n`, is preceded by a step along the child-axis from `n`'s parent to `n` or one of `n`'s siblings. `SPathLite` counts the number of evaluations on the stack and clears the parent-relation when the `depth` counter reaches zero.

The instantiation of `SPathLite` on `scala.xml.Node` has been named `XSPathLite`. This extension is used in all of the queries in this article. SPath has also been successfully instantiated on `org.w3c.dom.Node` and `java.awt.Component`. The `XSPathLite` extension of SPath has an implementation of the XPath function `id`. The XPath function call `id(QName)` returns the element node of the document that has its `id` attribute value equal to the value of the context node's attribute, `QName`. `XSPathLite` indexes each element node by its `id` attribute. The SPath function `$id(a:Attribute)` creates an axis that maps the context node to the referenced node by retrieving it from the index in constant time. Examples of `$id` appear in Appendix A.

8 Conclusion and related work.

Modal and temporal logic has been widely used as a foundation for query-languages on semi-structured data [Mar04b, BJ07, Mar04a, CG04]. LTL in particular, is expressively equivalent to first-order logic over linear sequences [GPSS80]. Automata-theoretic model checking algorithms [GPV$^+$95, Var07] for LTL are readily available for implementing simple query-evaluation with the prospect of a worst-case running-time that is $O(|\mathcal{D}| \times 2^{|\phi|})$ where $|\phi|$ is the size of the query and $|\mathcal{D}|$ the size of the document-tree. It has been straightforward to implement query evaluation in SPath, by an explicit-state enumeration that is instrumented with SPath's user-defined axis-functions. Linear-time query evaluation for Regular XPath [CDGLV09] uses alternating tree-automata. SPath could also benefit from this algorithm but its application would not improve on the exponential-time evaluation with user-defined axes in SPath.

The conditional axes of Core XPath [Mar04b] allow transitive closures over single, atomic axis-relations but disallows transitive closure of sequential compositions of

axis-relations, which is also not expressible in χ_{until}. Consequently, queries such as selecting the nodes that are an odd number of steps away from the context-node, cannot be expressed in Core XPath. In [Mar04b], Core XPath is shown to be equal in expressiveness to χ_{until}, a temporal logic with branching-time reasoning. The χ_{until} formula $\pi(\phi, \psi)$ is satisfied whenever there exists a path along the transitive closure of the primitive axis π, on which ψ holds until ϕ. Although the formal language SPath is based on LTL and can reason about a single order of time, it is possible for branching-time reasoning to be expressed in the Scala DSL for SPath using propositional functions that evaluate queries. For example, the predicate `not(exists(\(not(A))))` is true at a node `n` when all of `n`'s children are `A` nodes.

TQL [CG04] is a query-language for semi-structured data that is based on the modal logic for the ambient calculus. The `Until` formula of LTL, `a U b`, can be expressed in TQL as the recursive formula `rec $x.  a.$x Or b`. For example, selecting the nodes that are an even number of steps away from the context-node can be written in TQL using a query comprehension:

```
from $doc |= rec $x.  .%.$a.$x Or 0
select $a
```

which returns the set of all the bindings to `$a` for the required nodes. However, it is not straightforward to express the query from Example 4.2 in TQL. A employee's manager can be selected using a nested query comprehension, with a variable bound to the `mgrId` attribute, but TQL's syntax does not allow recursive query comprehensions, which is what this query seems to require in TQL. SPath, in contrast, uses variable bindings of the host language, Scala, to define the relations between the attributes of nodes. For example, the following query from Example 4.2,

```
$(company, \\(employee(id == "1"))\reports\\reports)
```

bootstraps the host language and the `Until` operator applies the recursion along the `reports` axis, as needed.

The Scrap Your Boilerplate (SYB) pattern [LJ03, LÖ7] for Haskell alleviates programmers from writing the boilerplate-code associated with writing queries on tree data-structures. The SYB framework relies on rank-2 polymorphism for traversal over tree-nodes and provides a variety of combinators for rich traversal strategies. SYB is more general than SPath in this respect, as different queries may require different traversal combinators e.g. top-down, bottom-up and partial traversals. In contrast, users of SPath write declarative queries in an XPath-like syntax and the framework applies a general search algorithm that can be fine tuned with optimisations.

XQuery 1.0 [W3Cd] incorporates XPath 2.0 featuring function calls as location-path steps. For example, the user-defined function `f` may be applied to the context-node in the path expression `/A/B/f(.)/C`. When combined with the abbreviated syntax of XPath 2.0, the expression `B//f(.)` is effectively expanded to

```
B/descendant-or-self::node()/f(.).
```

However, the function call `f(.)` is applied only once, *along the depth of the axis*, compared to an SPath axis as a location-path step, `\\(f)`, which applies `f` many times. The implementation of the query from Example 4.2, in XQuery, requires boilerplate for tree-traversal and cycle-detection to ensure termination such as in [Kaya]. SPath terminates on cyclic evaluation-paths.

The symbol /, which separates location steps, can be interpreted as a function that has a type that is similar to the bind function in the list monad of Haskell [Wad92]. The type of / could be interpreted as:

```
/ :  seq a => (a => seq b) => seq b
```

That is, / takes a sequence of nodes, of type `seq a`, with each node of type `a` and a function that maps values of type `a` to a sequence of nodes in which each node has type `b`. The map is applied to all nodes of type `a` and the result is the *flattening* of the many sequences of nodes of type `b` into one sequence of nodes of type `b`. An implementation of / could be formulated as:

```
as / f = as.flatMap(f)
```

In this interpretation of XPath queries, query-evaluation is based on chaining together a series of function applications. Each function may be either a predefined axis of XPath or any user-defined function on tree-nodes. Both C# and Scala allow a functional style for the evaluation of XPath queries. In each case, user-defined functions within a location step can be emulated through language-specific embeddings, using either extension methods of C# 3.5 or Scala's implicit conversions.

Scala's API for XML implements a subset of XPath for navigating XML documents. In particular, the class `scala.xml.Node` has methods for the child and descendant axes. The parent-axis is absent because nodes do not contain links back to their parent nodes. The reason for this is most likely due to performance. For example when a node is shared between documents, the subtree rooted at the shared node does not need to be copied in memory.

Cω [BMS05] features a special dot-operator for generalised member-access. In particular, the primitive operator `...` for transitive member-access is analogous to the descendant-or-self axis of XPath. The transitive member-access operator can also be combined with filter expressions allowing downward queries directly on object graphs. However, primitive axis-operators preclude the opportunity for extending the axis-relations in a consistent way without extending the language design. Cω's apply-to-all expression resembles XPath's syntax for location-path steps, enabling new axis-relations to be defined through extension methods.

LINQ-to-XML and Cω both share a similar semantics for location-path steps with XPath 2.0. Cω's streams are always flattened-out and the generalised dot-operator, when applied to a stream, applies the method-call or code-block to all elements in the stream.

The XPath 2.0 syntax for function calls within location-path steps could be more more closely aligned with the syntax of the core axes of XPath 1.0 [Kayb, Jel]. However, conditional axes could also be included in XPath by using a new syntax similar to Core XPath [Mar04b, Mar04a], rather than altering the current semantics of `//f(.)`.

9 Further work.

Future work on SPath includes optimising its performance in terms of execution time and memory usage. Query rewriting, based on semantic equivalence could reduce the size of queries, thereby reducing the size of automata. The evaluation algorithm stores each processed state of the product automaton in a cache but only the states that are reachable from the frontier of the search need to be retained. SPath's core axes are implemented as SPath axes, but they could be replaced with more

efficient implementations. For example, the axes `following` and `preceding` can be implemented simply, with Scala's non-strict views of an indexed sequence that stores all the nodes in document order. The automata in SPath are not constructed on-the-fly because the queries used with SPath so far tend to be small. However, SPath could be modified to construct the automata on-the-fly, if necessary. Query-evaluation algorithms with better complexity bounds could also be investigated.

The `XSPathLite` extension of SPath can be scaled-up to cover more features of XPath such as its full set of functions and operators.

The domain-specific predicates for XML elements and attributes are currently written by hand but an IDE-plugin could generate them automatically from an XML document or schema.

SPath can also be applied to semi-structured documents that are represented by edge-labeled trees, such as YAML and JSON.

The source code for SPath including the examples in this article, is available from [spa].

10 Acknowledgment.

I am grateful to the anonymous reviewers for their helpful comments about this work.

References

[BJ07] Michael Benedikt and Alan Jeffrey. Efficient and Expressive Tree Filters. In *FSTTCS'07: Proceedings of the 27th international conference on Foundations of software technology and theoretical computer science*, pages 461–472, Berlin, Heidelberg, 2007. Springer-Verlag. `doi:10.1007/978-3-540-77050-3_38`.

[BMS05] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The Essence of Data Access in Cω. In *Proc. ECOOP 2005*, pages 287–311, 2005. `doi:10.1007/11531142_13`.

[CDGLV09] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. An Automata-Theoretic Approach to Regular XPath. In *Proc. of the 12th Int. Symposium on Database Programming Languages (DBPL 2009)*, volume 5708 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2009. `doi:10.1007/978-3-642-03793-1_2`.

[CG04] Luca Cardelli and Giorgio Ghelli. TQL: a Query Language for Semistructured Data Based on the Ambient Logic. *Mathematical. Structures in Comp. Sci.*, 14:285–327, June 2004. `doi:10.1017/S0960129504004141`.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[Fra07] Massimo Franceschet. XPathMark: Functional and Performance Tests for XPath. In *XQuery Implementation Paradigms*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Available from: `http://drops.dagstuhl.de/opus/volltexte/2007/892`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GKP02] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. In *VLDB*, pages 95–106, 2002. doi:10.1145/1071610.1071614.

[GPSS80] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the Temporal Analysis of Fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '80, pages 163–173, New York, NY, USA, 1980. ACM. doi:10.1145/567446.567462.

[GPV+95] Rob Gerth, Doron Peled, Moshe Y. Vardi, R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi, and Pierre Wolper. Simple On-the-Fly Automatic Verification of Linear Temporal Logic. In *Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[Hol03] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[Jel] Rick Jelliffe. XPath needs virtual axes. Available from: http://broadcast.oreilly.com/2010/02/xpath-needs-virtual-axes.html.

[JGP99] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[jQu] jQuery. Available from: http://jquery.com/.

[jsr04] JSR 206: Java API for XML Processing (JAXP) 1.3., 2004. Available from: http://jcp.org/en/jsr/detail?id=206.

[Kaya] Michael Kay. Defining your own Functions in XQuery. Available from: http://www.stylusstudio.com/xquery/xquery_functions.html.

[Kayb] Michael Kay. Pipedreaming: Could XPath have been better? Available from: http://saxonica.blogharbor.com/blog/_archives/2010/1/5/4420740.html.

[Kay04] Michael Kay. *XPath 2.0 Programmer's Reference*. Wrox, 2004.

[LÖ7] Ralf Lämmel. Scrap Your Boilerplate with XPath-like Combinators. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 137–142, New York, NY, USA, 2007. ACM. doi:10.1145/1190216.1190240.

[LJ03] Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: a Practical Design Pattern for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM. doi:10.1145/604174.604179.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '85, pages 97–107, New York, NY, USA, 1985. ACM. doi:10.1145/318593.318622.

[Mar04a] Maarten Marx. Conditional XPath, the First Order Complete XPath Dialect. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 13–22, New York, NY, USA, 2004. ACM. doi:10.1145/1055558.1055562.

[Mar04b] Maarten Marx. XPath with Conditional Axis Relations. In *EDBT*, pages 477–494. Springer, 2004. doi:10.1007/978-3-540-24741-8_28.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[Mic] Microsoft. .NET Language-Integrated Query for XML Data. Available from: http://msdn.microsoft.com/hi-in/library/bb308960(en-us).aspx.

[MSB03] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Programming with circles, triangles and rectangles. In *XML Conference and Exposition*, 2003. Available from: http://research.microsoft.com/en-us/um/people/emeijer/papers/xml2003/xml2003.html.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 2008.

[spa] SPath. Available from: https://github.com/nicnguyen/SPath.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: a benchmark for XML data management. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 974–985. VLDB Endowment, 2002. Available from: http://dl.acm.org/citation.cfm?id=1287369.1287455.

[Var07] Moshe Y. Vardi. Automata-Theoretic Model Checking Revisited. In *Proceedings of the 8th international conference on Verification, model checking, and abstract interpretation*, VMCAI '07, pages 137–150, Berlin, Heidelberg, 2007. Springer-Verlag. doi:10.1007/978-3-642-01702-5_2.

[W3Ca] W3C. Document object model (DOM) technical reports. Available from: http://www.w3.org/DOM/DOMTR.

[W3Cb] W3C. XML path language (XPath): Version 1.0. Available from: http://www.w3c.org/TR/xpath/.

[W3Cc] W3C. XML path language (XPath): Version 2.0. Available from: http://www.w3.org/TR/xpath20/.

[W3Cd] W3C. XQuery 1.0: An XML Query Language. Available from: http://www.w3.org/TR/xquery/.

[W3Ce] W3C. XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition). Available from: http://www.w3.org/TR/xpath-datamodel/.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM. doi:10.1145/143165.143169.

[xma] XMark data generator. Available from: `http://www.xml-benchmark.org/generator.html`.

[xqj09] JSR 225: XQuery API for Java (XQJ), 2009. Available from: `http://jcp.org/en/jsr/detail?id=225`.

A SPath queries based on the XPathMark performance test.

```
A1 site\closed_auctions\closed_auction\annotation\description\text\keyword
A2 \\(closed_auction)\\keyword
A3 site\closed_auctions\closed_auction\\keyword
A4 site\closed_auctions\closed_auction?(\(annotation)\description\text\keyword)\date
A5 site\closed_auctions\closed_auction?(\\(keyword))\date
A6 site\people\person?(\(profile)\gender)?(\(profile)\age)\name
A7 site\people\person?(\(phone or homepage))\name
A8 site\people\person?(\(address))?(\(phone or homepage))
                                  ?(\(creditcard or profile))\name

B1 site\regions\*\item?(\(parent, namerica or samerica))\name
B2 \\(keyword)\(ancestor, listitem)\text\keyword
B3 site\open_auctions\open_auction\(\(bidder)$rtrim(1))
B4 site\open_auctions\open_auction\(\(bidder)$ltrim(1))
B5 site\regions\*\item $rtrim(1)\name
B6 site\regions\*\item $ltrim(1)\name
B7 \\(person?(\(profile(income))))\name
B8 site\open_auctions\open_auction(\(bidder)$size(1))\interval
C1 site\people\person?(\(profile)\age >= 18 and ?(\(profile(income < 10000)))
 and ?(\(address)\city <> "Dallas"))\name
C2 site\open_auctions\open_auction?(\(bidder)\increase join \(current))\interval
C4 site\people\person(id on (person_id, \(watches)\watch\$id(open_auction)\seller))
                                       \name

C5 \\(id == "person0")
C6 site\people\person\watches\watch\$id(open_auction)\interval
C7 site\people\person?(\(watches)\watch\$id(open_auction)\itemref
                               \$id(item)\(parent, australia))\name

D1 site\open_auctions\open_auction(\(bidder)$context((s:Int) => s%2==0))\interval
D2 \\(text or bold or emph or keyword)

E1 site\open_auctions\open_auction?(
   ((((\(bidder)$first)\increase) < (\(bidder)$nth((s:Int) => (s+1)/2)\increase))
    and ((\(bidder)$nth((s:Int) => (s+1)/2)\increase) < ((\(bidder)$last)\increase))
   )\interval

E2 site\regions\europe\item\description\(\(descendant, keyword)$first)
E3 \\(keyword)\(\(ancestor, listitem)$last)\text\keyword

E4 site\open_auctions\open_auction\bidder?(
                     \->(leftSibling,bidder)\increase <= \(increase) and
                     \(increase) <= \->(rightSibling,bidder)\increase)

E5 site\regions\*\item $ltrim(100)$rtrim(100)\name
F1 \\(bidder)?(\(increase) <= 10 and \->(rightSibling, bidder and \(increase) > 10))
F2 \\(bidder)?(\(increase) <= 10 and \->(leftSibling, bidder and \(increase) > 10))

F3 \\(listitem)\\compose(x, 2)\text\keyword
val x = $(\(parlist)\item)

F4 site\open_auctions\open_auction $range(1,5)\\F4axis\interval
val F4axis = $(\(seller)\$id(person)\watches\watch\$id(open_auction))

F5 site\people\person $range(1,5)\\F5axis\name
val F5axis = $(\(watches)\watch\$id(open_auction)\bidder\personref\$id(person))

F7 site\catgraph\edge(from == "category0")\\F7axis\$id(to)\name
val F7axis :  axis = n => $(n, ~\(catgraph)\edge(from == to @@ n))
```

B Automaton construction algorithm adapted from [GPV+95].

The states of the automaton are defined by the class `Node`, which has six fields:

```
class Node(  name:String, father:String, old:Set[Expr],
             new:Set[Expr], next:Set[Expr], incoming:Set[String])
```

The function `expand` constructs and links the states of the automaton:

```
expand:  (Node, Set[Node]) => Set[Node]
```

The automaton states, `ns`, for an expression `e` are created by the call:

```
expand(new Node(new_name(), null, ∅, {e}, ∅, {"init"}), ∅) = ns
```

The initial states are `n` ∈ `ns` with `"init"` ∈ `n.incoming`. The transition `n` ⟶ `n'` exists between `n`, `n'` ∈ `ns` when `n.name` ∈ `n'.incoming`. Let `n'=n[old+=` η`]` abbreviate `n' = n.clone; n'.old +=` η where `clone` returns a deep clone of `n`. The algorithm is presented in the form of the inference rules, `R1` to `R6`.

$$\textbf{R1}\quad \frac{\texttt{expand(n[old += }\eta\texttt{; new -= }\eta\texttt{], ns) = ns'}}{\texttt{expand(n, ns) = ns'}}\ \eta = \texttt{p} \in \texttt{n.new}$$

$$\textbf{R2}\quad \frac{\texttt{n' = n}\left[\begin{array}{l}\texttt{old += }\eta\texttt{;}\\ \texttt{new -= }\eta\texttt{ += (\{}\mu\texttt{,}\psi\texttt{\} -= old)}\end{array}\right]\quad \texttt{expand(n', ns) = ns'}}{\texttt{expand(n, ns) = ns'}}\ \eta = \mu \wedge \psi \in \texttt{n.new}$$

$$\textbf{R3}\quad \frac{\texttt{expand(n[old += }\eta\texttt{; new -= }\eta\texttt{; next += }\mu\texttt{], ns) = ns'}}{\texttt{expand(n, ns) = ns'}}\ \eta = \texttt{X}_\chi\mu \in \texttt{n.new}$$

(`new`$_i$, `next`$_i$ are defined in Figure 14.)

$$i \in \{1,2\},\ \texttt{n}_i\texttt{ = n}\left[\begin{array}{ll}\texttt{name = new\_name();} & \texttt{father = name;}\\ \texttt{new -= }\eta\texttt{ += (new}_i\texttt{(}\eta\texttt{) -= old);} & \texttt{old += }\eta\texttt{;}\\ \texttt{next += next}_i\texttt{(}\eta\texttt{)} & \end{array}\right]$$

$$\textbf{R4}\quad \frac{\texttt{expand(n}_1\texttt{,ns) = ns'' expand(n}_2\texttt{, ns'') = ns'}}{\texttt{expand(n, ns) = ns'}}\ \eta = \mu\,\texttt{U}_\chi\psi\texttt{,}\mu \vee \psi \in \texttt{n.new}$$

$$\textbf{R5}\quad \frac{\texttt{expand(n'', ns} \cup \texttt{\{n\}) = ns'}}{\texttt{expand(n, ns) = ns'}}\quad \begin{array}{l}\texttt{n.new = }\emptyset\\ \nexists\ \texttt{n'} \in \texttt{ns.}\\ \texttt{n'.old = n.old}\\ \texttt{n'.next = n.next}\end{array}\quad \texttt{n'' = }\left[\begin{array}{l}\texttt{name = father = new\_name();}\\ \texttt{incoming = \{n.name\};}\\ \texttt{new = n.next;}\\ \texttt{old = }\emptyset\texttt{; next = }\emptyset\end{array}\right]$$

$$\textbf{R6}\quad \frac{}{\texttt{expand(n, ns) = ns[n'.incoming += n.incoming]}}\quad \begin{array}{l}\texttt{n.new = }\emptyset\\ \exists\ \texttt{n'} \in \texttt{ns.}\\ \texttt{n'.old = n.old}\\ \texttt{n'.next = n.next}\end{array}$$

η	$\texttt{new}_1(\eta)$	$\texttt{new}_2(\eta)$	$\texttt{next}_1(\eta)$	$\texttt{next}_2(\eta)$
$\mu\ \mathtt{U}_\chi \psi$	$\{\mu\}$	$\{\psi\}$	$\{\mu\ \mathtt{U}_\chi \psi\}$	$\emptyset$
$\mu \vee \psi$	$\{\mu\}$	$\{\psi\}$	$\emptyset$	$\emptyset$

Figure 14 – $\texttt{new}_i$, $\texttt{next}_i$ for $i \in \{1, 2\}$ - used in R4.

C Proof sketches.

Expressions $\texttt{e} \in \texttt{Expr}$ are annotated with labels from a set of countably infinite labels $l \in$ Labels, such that each label in e occurs exactly once.

$$\texttt{e} \in \texttt{LabeledExpr} ::= \texttt{p}^l \mid (\texttt{X}_\chi\ \texttt{e})^l \mid (\texttt{e}\ \texttt{U}_\chi\ \texttt{e})^l \mid (\texttt{e} \vee \texttt{e})^l \mid (\texttt{e} \wedge \texttt{e})^l$$

Subexpressions of the expression e are thus uniquely identified and the labels will be omitted from here on.
An expression context ε is a placeholder for subexpressions:

$$\varepsilon ::= \texttt{[•]} \mid \texttt{X}_\chi\ \varepsilon \mid \varepsilon\ \texttt{U}_\chi\ \texttt{e} \mid \texttt{e}\ \texttt{U}_\chi\ \varepsilon \mid \varepsilon \vee \texttt{e} \mid \varepsilon \wedge \texttt{e} \mid \texttt{e} \vee \varepsilon \mid \texttt{e} \wedge \varepsilon$$

The expression $\texttt{e}_2$ is a subexpression of $\texttt{e}_1$ when there exists a context ε such that $\texttt{e}_1$ and $\varepsilon\texttt{[e}_2\texttt{]}$ are syntactically identical.
Lemma 1. If SPath(ε[e]) then SPath(e).
Proof. By induction on the size of ε.
Definition. *The expressions* $\texttt{e}_1$ *and* $\texttt{e}_2$ *are distinct when there does not exist* ε *such that either* $\texttt{e}_1 = \varepsilon[\texttt{e}_2]$ *or* $\texttt{e}_2 = \varepsilon[\texttt{e}_1]$.
Let Π be the fully expanded tree for the formula $\texttt{e}_\texttt{I}$ with $\texttt{SPath}(\texttt{e}_\texttt{I})$:

$$\Pi \quad \frac{\vdots}{\texttt{expand}(\texttt{n}_0, \texttt{ns}_0) = \texttt{ns}}$$

where $\texttt{ns}_0 = \emptyset$ and $\texttt{n}_0$ is the initial node of the construction:

$$\texttt{new Node(new\_name(), null, } \emptyset\texttt{, \{e}_\texttt{I}\texttt{\}, } \emptyset\texttt{, \{"init"\})}$$

Let Γ be a subpath in Π of height k, starting at the root of Π, in which $\texttt{n}_k$ is a fully constructed initial node that is added to the node set by R5:

$$\Gamma \quad \frac{\dfrac{}{\texttt{expand (n}_k\texttt{, ns}_k\texttt{) = ns'}}\ \texttt{R5}}{\dfrac{\vdots}{\texttt{expand}(\texttt{n}_0\texttt{, ns}_0\texttt{) = ns}}}$$

Each $\texttt{n}_i$ for $k \geq i \geq 1$ are the *same-time* descendants of $\texttt{n}_0$, such that, $\texttt{n}_i\texttt{.name = n}_{i-1}\texttt{.name}$ or $\texttt{n}_i\texttt{.father = n}_{i-1}\texttt{.name}$. Γ is constructed by the rules R1, R2, R3 and R4.
Lemma 2. If $\texttt{e} \in \texttt{n}_i\texttt{.new}$ then $\exists\texttt{e'} \in \texttt{n}_{i-1}\texttt{.new}$, $\varepsilon\ \texttt{:e'} = \varepsilon[\texttt{e}]$.
Proof. Immediately by the rules R1, R2, R3 and R4.
Lemma 3. If $\texttt{e} \in \texttt{n}_k\texttt{.old}$ then $\exists\varepsilon\texttt{:e}_\texttt{I} = \varepsilon[\texttt{e}]$.
Proof. By Lemma 2, take $\varepsilon = \varepsilon_1[\ldots\varepsilon_k[\bullet]\ldots]$.
Proof of Proposition 6.2. The proof proceeds to show by contradiction that for $\texttt{n}_k$ in Γ: $|\Theta(\texttt{n}_k)| \leq 1$ and $|\texttt{n}_k\texttt{.next}| \leq 1$.

Suppose that, in Γ, $|\Theta(\mathtt{n}_k)| > 1$. Then there exists $\mathtt{e}_1$, $\mathtt{e}_2 \in \mathtt{n}_k\mathtt{.old}$, both of the form $\mathtt{X}_\chi\mathtt{e}$ or $\mathtt{e}\ \mathtt{U}_\chi\mathtt{e'}$ with $\mathtt{e'} \notin \mathtt{n}_k\mathtt{.old}$. The expressions $\mathtt{e}_1$ and $\mathtt{e}_2$ must be either distinct or not distinct.

Suppose $\mathtt{e}_1$ and $\mathtt{e}_2$ are distinct. The rules R1, R3 and R4 cannot introduce distinct subexpressions into an $\mathtt{n}_i\mathtt{.new}$ in Γ. So R2 must have been applied with η of the form $\varepsilon_1[\mathtt{e}_1] \wedge \varepsilon_2[\mathtt{e}_2]$. By Lemma 3, there exists an ε such that $\mathtt{e}_\mathtt{I} = \varepsilon[\eta]$. By Lemma 1, $\mathtt{SPath}(\eta)$. But $\mathtt{Axes}(\varepsilon_1\mathtt{[e_1]}) \neq \emptyset$ and $\mathtt{Axes}(\varepsilon_2\mathtt{[e_2]}) \neq \emptyset$. So $\mathtt{SPath}(\eta)\mathtt{=false}$.

Suppose $\mathtt{e}_1$ and $\mathtt{e}_2$ are not distinct. Then R4 must have been applied in Γ with η of the form $\varepsilon_1\mathtt{[e_1]}\ \mathtt{U}_\chi\ \mathtt{e'}$. A similar contradiction now occurs, because by definition of SPath, it is required that $\mathtt{Axes}(\varepsilon_1\mathtt{[e_1]}) = \emptyset$. So $|\Theta(\mathtt{n}_k)| \leq 1$.

Let $|\Gamma|_{\mathtt{R2}}$ equal the number applications of R2 in Γ. Let $|\Gamma|_{\mathtt{R4}}$ equal the number of applications of R4 in Γ, which take the LHS branch of R4. If $|\Gamma|_{\mathtt{R2}} + |\Gamma|_{\mathtt{R4}} > 1$ then $|\Theta(\mathtt{n}_k)| > 1$. But $|\Theta(\mathtt{n}_k)| \leq 1$. So $|\Gamma|_{\mathtt{R2}} + |\Gamma|_{\mathtt{R4}} \leq 1$. New subformulas are placed into a node's next formulas only by the rules R3 and the LHS branch of R4. So $|\mathtt{n}_k\mathtt{.next}| \leq 1$. By definition of the construction and Lemma 3, $\mathtt{e} \in \mathtt{n}_k\mathtt{.next}$ is a subformula of $\mathtt{e}_\mathtt{I}$. By Lemma 1, $\mathtt{SPath(e)}$. By R5, the expansion of the successor of $\mathtt{n}_k$ is rooted at the premise of R5 and has the same form as Γ. The same proof is applied to all fully constructed nodes in Π. ■

Proof of Proposition 6.1. In Γ, the successor, n, of $\mathtt{n}_k$ is initialised by R5 with $\mathtt{n.new = n}_k\mathtt{.next}$. If $|\mathtt{n}_k\mathtt{.next}| > 0$ then by Proposition 6.2, there exists an e such that $\mathtt{n}_k\mathtt{.next = \{e\}}$. So $\mathtt{n.new = \{e\}}$. By application of Lemma 2 and the definition of the construction, e is a subexpression of $\mathtt{e}_\mathtt{I}$. Each successor is thus initialised with a smaller subexpression of its predecessor's first new expression. So there must exist a node n' with new initialised to {e'} such that e' does not contain a future-time formula. By definition of the construction, each same-time descendant of n' has $\mathtt{next} = \emptyset$. Furthermore, the algorithm of [GPV$^+$95] discards a node, only when it contains a contradiction - which is impossible for SPath because it is based on the negation-free subset of LTL. A descendant of n' must therefore exist in the node-set of the automaton construction. ■

Lemma 4. For any node n in the automaton construction, $\mathcal{A}$, if $\mathtt{X}_\chi\mathtt{e'} \in \mathtt{n.old}$ and $\mathtt{n} \longrightarrow \mathtt{n'}$ is a transition in $\mathcal{A}$ then $\mathtt{e'} \in \mathtt{n'.old}$.

Proof. Immediately from the construction. ■

Proof of Corollary 6.4. The proof of [GPV$^+$95] is modified by replacing each formula of the form $\mathtt{X} \bigwedge \mathtt{Next(q)}$ with $\Psi(\mathtt{q})$ where

$$\Psi(\mathtt{q}) = \begin{cases} \mathtt{true} & \mathtt{q.next} = \emptyset \\ \mathtt{X}_\chi\ \mathtt{e} & \mathtt{q.next} = \{\mathtt{e}\} \text{ and } \chi \text{ is the unique axis in } \mathtt{q.old} \end{cases}$$

and true is the propositional function $\mathtt{n => true}$. Lemma 4.4 of [GPV$^+$95] is extended with the premise $|\xi| > 1$ for the finite sequence ξ. Part 2 of Lemma 4.1 is extended as: $\exists j \geq 0\ \forall i\ 0 \leq i < j : \mu, \mu\ \mathtt{U}\ \eta \in \Delta(q_i)$ and $\eta \notin \Delta(q_i)$ and $\eta \in \Delta(q_j)$. Lemma 4.7 of [GPV$^+$95] now follows by using the amended Lemma 4.1 and Lemma 4 in the case $\mathtt{X}_\chi\mathtt{e}$. Lemma 4.9 of [GPV$^+$95] follows by the amended Lemma 4.4, Definition 6.3 and the definition of SPath's semantic relation $\models$. ■

Lemma 5. The upper bound for result.size in the function evaluate is $S = |\mathcal{A}| \times |\mathcal{D}|$.

Proof. The cache ensures that each state of the product automaton, $\mathcal{A} \times \mathcal{D}$, is processed in map at most once. Line 6 can therefore append up to $|\mathcal{D}|$ document nodes to result for each state q in $\mathcal{A}$. It follows that the size of result, when evaluate terminates on line 3 cannot be larger than $S = |\mathcal{A}|.|\mathcal{D}|$. ■

Lemma 6. `distinct(it:Traversable[T])` takes time proportional to $4n + 2$ where n `= it.size` and uses space proportional to $2.|\mathcal{D}|$.
Proof. The function `distinct` creates a `HashSet` to store unique document nodes and a `ListBuffer` for the result. The function traverses over its argument adding each item to the result if not already contained in the `HashSet` and then adding each item to the `HashSet`. The time is 2 statements to create the `HashSet` and `ListBuffer` and 4 statements per item of the `Traversable` collection. The upper bound for the space usage of the `HashSet` and the result is twice the number of document nodes in $\mathcal{D}$. ■
Proof of Proposition 6.7. By the definition of Φ, for any `e` such that $\Phi(\mathtt{e}) > 1$:

$$\Phi(\mathtt{e}) = 1 + \mathtt{Max}\{\Phi(\mathtt{e'}) \mid \mathtt{o} => \mathtt{S} \in \mathtt{UF(e)} \;\&\; \mathtt{e'} \in \mathtt{Queries(S)}\}$$

Thus, for such a maximal `e'`, $\Phi(\mathtt{e'}) = \Phi(\mathtt{e})-1$. This equality enables the approximation of the worst case running-time for the evaluation of `e`, by the recurrence R_i where $i = \Phi(\mathtt{e})$. The running time of each line of the function `evaluate` of Figure 8 is summarised as follows:

Line numbers	**Running time for R_1**	**Running time for R_i with $i > 1$**
2, 4, 5, 6, 7, 13, 14	$7.S$	$7.S$
3	$4.S + 2$	$4.S + 2$
9	$S.(4.\|\mathcal{D}\| + 2) + S.A_0$	$S.(4.\|\mathcal{D}\| + 2) + S.\|\mathtt{e}\|.R_{i-1}$
11	$S^2 + S.\|\mathtt{e}\|.P_0$	$S^2 + S.\|\mathtt{e}\|^2.R_{i-1}$
12	$S.\|\mathcal{A}\|$	$S.\|\mathcal{A}\|$

where the values P_0, A_0 and S are defined as:

Quantity			**Meaning**
P_0	=	1	*Running-time of* $\mathtt{p}$ *with* $\Phi(\mathtt{p}) = 0$.
A_0	=	$\|\mathcal{D}\|$	*Running-time of* χ *with* $\Phi(\chi) = 0$.
$\|\mathcal{A}\|$	=	$2^{\|\mathtt{e}\|}$	*Size of automaton for* `e`.
S	=	$\|\mathcal{A}\|.\|\mathcal{D}\|$	*Size of product automaton* $\mathcal{A} \times \mathcal{D}$.

Adding up the columns for R_1 and R_i gives:

$$\begin{aligned} R_1 &= 11.S + 2 + S.(4.|\mathcal{D}| + 2) + S.A_0 + S^2 + S.|\mathtt{e}|.P_0 + S.|\mathcal{A}| \\ R_i &= 11.S + 2 + S.(4.|\mathcal{D}| + 2) + S.|\mathtt{e}|.R_{i-1} + S^2 + S.|\mathtt{e}|^2.R_{i-1} + S.|\mathcal{A}| \end{aligned}$$

By induction on i, $R_i < (23.S^3.|\mathtt{e}|^2)^{\Phi(\mathtt{e})}$. By definition of S, it follows that $R_i < (23.(2^{|\mathtt{e}|}.|\mathcal{D}|)^3.|\mathtt{e}|^2)^{\Phi(\mathtt{e})}$. R_i is therefore $O((2^{|\mathtt{e}|} \times |\mathcal{D}|)^3 \times |\mathtt{e}|^2)^{\Phi(\mathtt{e})}$.

The running times are explained as follows. Lines 2, 4, 5, 6 and 13 each take constant time. Appending to the `ListBuffer` on line 7 also takes constant time. The call to `distinct` on line 3 executes just once and by Lemmas 5 and 6 takes time $4.S + 2$. An upper bound for the number of times lines 2, 4, 5, 6, and 7 execute is S i.e when each call to evaluate extends a single state of the product automaton by one transition. Lines 13 and 14 execute at most S times due to the cache-check on line 11. The running time of applying axis functions on line 9 includes $S.(4.|\mathcal{D}| + 2)$ i.e by Lemma 5, the time for `distinct` on a maximum result set of size $|\mathcal{D}|$, applied for each state in $\mathcal{A} \times \mathcal{D}$ that is of size S. For R_i, the running time of applying S axis functions of degree 0 is $S.A_0$. For R_i with $i > 0$, the running time of applying an axis function for each state of the product automaton is R_{i-1}, the degree of expressions evaluated directly by the axis, multiplied by the number of user-defined functions that are directly evaluated by the axis function, which is bounded by $|\mathtt{e}|$. This is then multiplied by the number of states S.

The running time of line 11 includes the time of applying the filter on each state `(q2, o)` where `q2` $\in$ `q.outgoing` and `o` $\in$ `ns2` with `ns2.size` $\leq |\mathcal{D}|$. Thus for a state `(q, n)` with `n` $\in$ `ns`, an upper bound for the number of caches checks is $|\mathcal{A}|.|\mathcal{D}|$ (i.e S) where $|\mathcal{A}|$ is an upper bound for the outgoing states from `q` and $|\mathcal{D}|$ is an upper bound for the size of `ns2`. The total time for cache checks is therefore S^2.

The number of propositional functions in `q2.old` is bounded by $|\mathtt{e}|$. The running time of each propositional function for R_1 of degree 0 is P_0. The running time of a propositional function in R_{i-1} is $|\mathtt{e}|.R_{i-1}$, i.e the number of expression of degree $i-1$ that are directly evaluated by the propositional function multiplied by their running time. Each propositional function is applied at most S times i.e once for each state due to the cache-check guard on line 11. Line 12 can execute up to S times for each outgoing state of $\mathcal{A}$. ■

Proof of Proposition 6.8. The proof follows a similar structure as the proof of Proposition 6.7. The space usage for evaluating an expression `e` with $i = \Phi(\mathtt{e})$ is approximated by the recurrence R_i as follows:

$$\begin{array}{rcl} R_1 & = & 3.S + 2.|\mathcal{D}| + S.2.|\mathcal{D}| + S.|\mathcal{D}| + 1 \\ R_i & = & 3.S + 2.|\mathcal{D}| + S.2.|\mathcal{D}| + S.|\mathcal{D}| + R_{i-1} \end{array}$$

R_1 is explained from left to right as follows. $3.S$ is an upper bound for memory required by the 3 arguments to `evaluate`; `map`, `result` and `cache`. By Lemma 5 and 6 the memory usage of `distinct` on line `3` of the function `evaluate` in Figure 8, is $2.|\mathcal{D}|$. The total memory usage of `distinct` on line `9` is $S.2.|\mathcal{D}|$. Each axis application can return a result with a size up to $|\mathcal{D}|$ in each state S, so the total memory required to store these results is $S.|\mathcal{D}|$. Each propositional function `p` with $\Phi(\mathtt{p}) = 0$ is assumed to use constant space 1.

R_i is explained in a similar way except that the space required for axis and propositional functions at level R_{i-1} is added to the total. Each axis application and propositional function executes sequentially so space for one axis or propositional function at degree i is required at a single moment.

It follows that R_i is $O(2^{|\mathtt{e}|} \times |\mathcal{D}|^2 \times \phi(\mathtt{e}))$. ■

About the author

Nicholas Nguyen is a software developer interested in object-oriented research and current practices. Contact him at `nnguyen@hotmail.co.uk`, or visit `http://nicnguyen.github.com`.