Extreme Markup Languages 2007®

Montréal, Québec
August 7-10, 2007

Streaming Validation of Schemata: the Lazy Typing Discipline

Paolo Marinelli
University of Bologna

Fabio Vitali
University of Bologna

Stefano Zacchiroli
University of Bologna

Abstract

Assertions, identity constraints, and conditional type assignments are (planned) features of XML Schema which rely on XPath evaluation to various ends. The allowed XPath subset exploitable in those features is trimmed down for streamability concerns partly understandable (the apparent wish to avoid buffering to determine the evaluation of an expression) and partly artificial.

In this paper we dissect the XPath language in subsets with varying streamability characteristics. We also identify the larger subset which is compatible with the typing discipline we believe underlies some of the choices currently present in the XML Schema specifications. We describe such a discipline as imposing that the type of an element has to be decided when its start tag is encountered and its validity has to be when its end tag is.

We also propose an alternative *lazy* typing discipline where both type assignment and validity assessment are fired as soon as they are available in a best effort manner. We believe our discipline is more flexible and delegate to schema authors the choice of where to place in the trade-off between using larger XPath subsets and increasing buffering requirements or expeditiousness of typing information availability.

Extreme Markup Languages®

Streaming Validation of Schemata: the Lazy Typing Discipline

Table of Contents

Streaming Validation of Schemata: the Lazy Typing Discipline

Paolo Marinelli, Fabio Vitali, and Stefano Zacchiroli

§ 1 Introduction

At the end of August 2006, a new draft of the upcoming 1.1 release of *XML Schema: Structures* has been published by the W3C [XMLSchema]. Among the major changes from the previous specifications we found the new concept of *assertions* (see Figure 1), one of the possible ways to express co-constraints [CoConstrWiki] in XML Schema schemata. Both identity constraints (see Figure 2), which were already present in former versions of the XML Schema specifications, and assertions exploit XPath[1] expressions [XPath2] to point to the elements and attributes that are relevant in the validity assessment of the elements they control. A further approach to co-constraints, CTA [Conditional Type Assignment], is being discussed at this moment for inclusion in subsequent drafts of XML Schema 1.1, and also heavily relies on XPath expressions to specify the elements and attributes that are relevant for the conditional assignment of type to the element or attribute being defined.

Figure 1: Assertion example

```
<xs:complexType name="T">
 <xs:attribute name="a" type="xs:string" />
 <xs:attribute name="b" type="xs:string" />
 <xs:report test="@a and @b" />
 <xs:assert test="@a or @b" />
</xs:complexType>
```

Two assertions about the attributes `a` and `b` are given: the former reports an error if both are present (i.e. `a` and `b` are mutually exclusive), the latter to impose that at least one among them is present.

Figure 2: Identity constraint definition example

```
<xs:element name="state">
 <xs:complexType>
  <xs:sequence maxOccurs="unbounded">
   <xs:element name="province">
    <xs:complexType>
     <xs:attribute name="code" type="xs:string" />
    </xs:complexType>
   </xs:element>
  </xs:sequence>
  <xs:attribute name="code" type="xs:string" />
 </xs:complexType>
 <xs:key name="provinceKey">
  <xs:selector xpath="./province" />
  <xs:field xpath="@code" />
 </xs:key>
</xs:element>
```

Enforcing that a province code is a key indexing the provinces of a state

Both identity constraints and assertions do not specify that the full syntax of XPath 2.0 can be used, but only limited subsets of it. In particular, they both exclude a large number of functions and operators, and many axes. The reason behind these restrictions is fundamentally the need to allow the implementation of streaming XML parsers that validate documents with respect to XML Schema schemata, as explained in the draft itself.

A streaming XML parser is a parser that accesses and verifies XML documents in a serial way, i.e. as a single, unidirectional data stream whereby previously accessed data are not meant to be re-read for new information. Streaming parsers generate events (e.g. *startElement* or *endElement*) for downstream applications upon encountering relevant XML structures in the incoming stream and never generate a full memory representation of the document. SAX [Sax] is a well-known API between streaming parsers and downstream applications that standardizes the events signaling the reading of noteworthy XML structures

Figure 3: Conditional declaration example

```
<xs:element name="x" type="T">
 <xs:alt cond="@a='foo'" type="Ta" />
 <xs:alt cond="@a='bar'" type="Tb" />
</xs:element>
```

An element validated against this declaration is assigned type `Ta` if it has an attribute `a` with value `"foo"`, `Tb` if `a` has value `"bar"`, type `T` otherwise.

in the source stream. Streaming parsers are extremely fast and require a much smaller memory footprint than with other tree-based parsing approaches like DOM [Dom].

Validating streaming parsers are able to assess the validity of elements and attributes while reading them in the source stream, and are designed to be able to do so as soon as possible, with a minimal amount of memory, and without the need to look ahead or behind in the incoming stream. One such parser is Xerces [Xerces2], the XML parser of the Apache project.

XPath expressions do not in general allow streaming implementations. In fact, XPath expressions associated to a specific XML context node can freely span to structures before or after the context node itself, which requires continuous and complete access to the full document tree for evaluation. Therefore, the subset of XPath included in the current draft of XML Schema contains only those constructs that are considered to behave safely in a streaming environment, for instance because they can be guaranteed to end their evaluation in a reasonable amount of time and using a reasonable amount of memory.

This paper makes two important claims, as well as a few minor ones. First of all, it claims that the subset of XPath constructs included in the XML Schema draft is even more restrictive than necessary for naive streaming implementations of XML Schema engines.[2] Second, it proposes a different validation concept that would further widen the allowable subset of XPath expressions and would allow implementation to tune finely the allowed subset to the memory and time requirements of the validation engine.

In order to prove these points, we have first analyzed the time and memory requirements of each structure of XPath 2.0, and have organized them in several classes whose stratification and characteristics are explained in Section 2.

Next we consider the validation model that is implicit in current implementations of streaming XML Schema engines. The validation of XML documents against an XML Schema has two very specific actions: first, a type is chosen for the element (or attribute), and next its content is evaluated against the content model specified in the type, in order to determine its validity.

The current rule of thumb for the evaluation of what is acceptable for streaming validation is what we call the STEVE [Start tag: Type; End tag: Validity Evaluation] discipline. STEVE imposes that the type is determined as soon as the element name is read from the input stream (or, with generous flexibility, at the end of the start tag that contains the name itself), and that the validity is determined as soon as the content of the element has been completely examined, i.e., at the end of the element itself, when reading its end tag.

Thus, in STEVE the relevant XPath expressions cannot point to structures before the start tag (because only at its occurrence we know that they are needed), and they also cannot point to structures after the end tag, since the validity needs to be assessed at the end tag of the element. As such, the only remaining XPath subset is whatever points downward, i.e. along the "attribute", "child", and "descendant" axes.

The STEVE discipline is even more harmful for CTA as introduced with SchemaPath [Marinelli04]. In CTA, the typing of an element is conditional to a set of XPath expressions, so that the same element can be typed in a way or another depending on their truth values (see Figure 3). Therefore, according to STEVE, typing must both start *and* end when reading the start tag of the element. Thus, the only remaining XPath axis for CTA is "attribute". This means that if XML Schema were to adopt CTA, the STEVE discipline would allow types only to depend on the values of the attributes of an element or a comparison between different attributes of the same element.

Yet, the usefulness of co-constraints (both in terms of assertions and, even more so, of conditional type assignments) can be easily determined if we consider the number of co-constraints that are currently expressed informally as plain text in appendices or comments of the fully formal schemata describing and

detailing XML vocabularies. Examples from XHTML [Extensible HyperText Markup Language] [XHTML], XSLT [XSL Transformations] [XSLT] , UBL [Universal Business Language] [UBL], FpML [Financial products Markup Language] [FpML], XML Schema itself [XMLSchema], and hundreds of other specialized vocabularies exist to prove this point.

This leads us to discuss some considerations (including forward-looking rewriting of backward-looking XPath expressions as first introduced by [Olteanu02]) about Full STEVE XPath, the widest allowable subset of XPath structures to be fruitfully evaluated in a STEVE-controlled streaming environment. In this paper we present a syntactic approximation of Full STEVE XPath which we believe is easily understandable for XML Schema users.

Next, we propose a new streaming evaluation discipline for XML Schema, which we call LATVIA [Lazy Assignment of Typing and Validity Infoset Aspects] that extends STEVE by allowing both typing and validity events to happen at any moment during the streaming processing of the input document, thus allowing the relevant elements and attributes to be anywhere in the document, and therefore the corresponding expressions to contain arbitrary XPath constructs.

A LATVIA-driven streaming environment would identify in advance all relative XPath expressions, converting them to forward-looking absolute expressions, and constantly verify their truth values throughout the parsing of the document. Whenever an element controlled by one such expression is found, an independent evaluator is detached and associated to the element, which returns a "Success" or "Failure" event as soon as the relevant elements and attributes are found in the document. If the original expression pointed upward or before the relevant element, it would have already succeeded or failed when meeting the element, and no further evaluation would be necessary. If the original expression pointed downward, then it either would succeed or fail before or at the end of the element, and the evaluation would stop in time for the end of the element. If the original expression pointed to structures after the end of the element, then the relevant expression could not be conclusively evaluated at that time, and its validity and type would remain in an undecided state up to the moment in which the relevant element would be read and evaluated.

LATVIA thus is adding a new possible state for the type and validity properties of an XML element (the value "not decided yet") and does not necessarily considers the typing at the start of the element and the validation at the end of the element as strict and final assertions, but rather as lazy assertions that may need to be verified and confirmed later on, when the appropriate document structures are found and verified.

We believe that LATVIA is the best possible approach to perform streaming validation in the presence of arbitrary co-constraints using assertions and conditional type assignments. We believe no alternative approach can provide a solution with lesser requirements in terms of memory occupation and expeditiousness in determining invalidity given the same subset of XPath supported.

1.1 *Paper Structure*

The paper is structured as follows. In Section 2 we provide the classification of XPath expressions. In Section 3 we introduce the LATVIA model. In Section 4 we describe our implementation of a streaming XML Schema validator supporting CTA and following the LATVIA discipline. In particular, our implementation is also able to evaluate the conditions used within conditional declarations. Thus, in Section 4 we also describe our implementation of a streaming XPath evaluator.

§ 2 Streaming XPath: a Stratification

In this section we describe several classes of XPath expressions. For each class we discuss streamability issues, namely: memory requirements and the amount of time that should be waited for to know the evaluation result. Memory requirements are measured on some metrics of the input document (e.g. the number of nodes, document depth, etc).

Our interest on XPath streamability derives from a more general interest on the streamability of XML Schema 1.1. In XML Schema schemata, XPath is always used within a *relative* context, i.e.:

- assertions are evaluated relatively to the element nodes assigned to the schema types which they are defined in;
- identity-constraints are evaluated relatively to the nodes matching the element declarations which they are defined in;
- CTA conditions are evaluated relatively to the nodes matching the conditional declaration which they are defined in.

Therefore it is natural to perform our stratification on the expressions that can be used within a location step predicate.

For each class we are going to consider, the complexity in the evaluation of its expressions is dominated by the *axes* allowed within location path sub-expressions, the *kind of predicates* allowed within their steps, and the set of allowed *functions*.

We will partition functions into two classes: *simple* and *complex* functions. The former class contains functions whose evaluation requires an amount of memory at most linear in the amount of memory required by the evaluation of its arguments; the latter functions whose evaluation requires to buffer nodes of the input document regardless of the memory required to evaluate its arguments. At the end of the section we will provide the list of complex functions.

Moreover, we operate a distinction also on simple functions. Indeed, we distinguish among boolean operators (which are simple functions) and the other simple functions. Such a distinctions will be useful when we will discuss about the evaluation of reverse paths.

Finally, we also distinguish among tail and inner predicates. We define a *tail predicate* as a predicate appearing within the predicate list of the last step of a location path. We call the other predicates *inner predicates*.

The main claim of this section is grasped by the following table (referenced as the *stratification table* in the remainder of the paper).

Table 1: XPath stratification. Each column selects a different set of axes, while each row identifies a different XPath feature. Within each cell, *startElement*, *endElement*, and *endDocument* indicate when the evaluation result is available w.r.t the context node; "constant", "linear-depth", and "linear-size" indicate if the amount of required memory is respectively constant, linear in the document depth, linear in the document size. Side numbers are used in the remainder of the paper for referencing this table cells.

	Backward Axes	**"attribute" Axis**	**Downward Axes**	**Forward Axes**	**Absolute Paths**	
Boolean ops	*startElement*, linear-depth	*startElement*, constant	*endElement*, linear-depth	*endDocument*, linear-depth	*endDocument*, linear-depth	1
Simple Functions	*startElement*, linear-size	*startElement*, constant	*endElement*, linear-depth	*endDocument*, linear-depth	*endDocument*, linear-depth	2
Tail Predicates	*startElement*, linear-depth	N/A[3]	*endElement*, linear-depth	*endDocument*, linear-size	*endDocument*, linear-depth	3
Inner Predicates	*startElement*, linear-depth	N/A[3]	*endElement*, linear-size	*endDocument*, linear-size	*endDocument*, linear-size	4
Complex Functions	*startElement*, linear-size	*startElement*, linear-size	*endElement*, linear-size	*endDocument*, linear-size	*endDocument*, linear-size	5
	1	2	3	4	5	

We identify the Full STEVE XPath subset *approximation* as the union of the XPath subsets contained in the first two columns of the stratification table. A precise, formal definition of Full STEVE XPath should take into account not only syntactic aspects (as those we are mainly concerned with in this paper) but also semantic ones. A precise semantic definition of Full STEVE XPath should be given in term of which nodes are touched by the evaluation of a given XPath expression: if and only if all of them fall before the context node the expression is part of Full STEVE XPath. Otherwise stated, it should be possible to

complete the evaluation of a Full STEVE XPath expression before the start tag of the context node is encountered. To grasp this idea consider the expression `./preceding::x/descendant::y/following-sibling::z` which escapes from our approximation. Still, its evaluation finishes before the context node and hence it is arguable that such an expression should be part of a precise Full STEVE XPath definition. However a trade-off between precise definition and understandability of the constraint by XML Schema authors should be considered: we believe our choice (choosing a syntactic constraint instead of a more involved semantic one) to be reasonable.

Other works such as [BarYossef04] and related works have studied formally the theoretical memory requirements for the streaming evaluation of XPath expressions. We compare to those works attempting a less formal (for the time being) analysis yet more complete not restricting ourselves to artificially small subset of the XPath language. Moreover, we also consider in our analysis the moment after which the evaluation of the expression can be considered completed (wrt a context node), scenario not analyzed in previous works which were not particularly biased toward the use of XPath in XML Schema.

The remainder of this section explains the most interesting placement choices, skipping the obvious ones. Each section title is equipped with "(row, column)" reference to the cell of the table it describes.

2.1 *The Evaluator API*

For each class we define the issues to be faced in designing a generic streaming XPath evaluator. We assume that the evaluator works in the following setting (note how the input requirements can be easily fulfilled by any SAX-compliant parser):

1. task of the evaluator is to evaluate a single XPath expression on a single streamed XML document;
2. the evaluator receives a sequence of input events describing the logical structure of the document to evaluate. The input events we are interested in are:
   - *startDocument*: it signals the beginning of the document;
   - *endDocument*: it signals the end of the document, i.e. the document does not contain further nodes;
   - *startElement*: it signals the start-tag of an element. We assume the event has some parameters specifying: the element name, a unique and progressive numeric identifier of the node (increasing following the document order), a number indicating the depth-level. Moreover, we assume this event is also parameterized in the attribute list of the element.
   - *endElement* it signals the end-tag of an element. Except for the attribute list, the event has the same parameters of *startElement*.
   - *text*: it signals the presence of a text node. The event is parameterized in a string specifying the text content.
3. the output of the evaluator depends on the semantics defined for the expression to evaluate. If the output is an atomic value, the output is an event specifying the result of the evaluation; if the result of the evaluation is a node sequence, than the evaluator produces a sequence of output events: an event for each node of the sequence. Each event contains the numeric identifier of the selected node. There is also an extra output event indicating the end of the sequence.

2.2 *Boolean Expressions over Attributes (1,2)*

In this section we consider the set of expressions built over relative paths without predicates, where boolean operators (`and`, `or`, and `not()`) can be used and where the only allowed axis is "attribute". An instance of such expressions is

```
(@a or @b) and @c
```

Given a context node e, any expression of this subset can be easily evaluated processing the attribute list specified in the *startElement* event of e. Thus, the evaluation of such an expression starts at the *startElement* event of the context node and ends at the same event: it is not necessary to wait for further events, in order to decide the result.

It is easy to observe that the evaluation of such expressions requires a *constant* amount of memory.

2.3 *Simple Functions over Attributes (2,2)*

Here we consider an XPath fragment extending the one defined in Section 2.2, allowing the usage of simple functions. An instance of this fragment is

```
(@a eq '8' or @b) and @c
```

Again, given an expression of this fragment and given a context node *e*, the evaluation of the expression has to start at the end *startElement* of *e*, and it ends by the same event. Indeed, it is sufficient to process the attribute list specified within the input event.

2.4 *Complex Functions over Attributes (5,2)*

In this section we consider the XPath fragment obtained from that defined in Section 2.3, allowing the usage of complex functions. As the evaluation of a complex functions may require an amount of memory linear in the number of document nodes, the evaluation of an expression of this fragment may require the same amount of memory.

2.5 *Boolean Expressions over Downward Paths (1,3)*

In this section we consider an extension of the fragment described in Section 2.2. The extension is obtained allowing the use of relative paths with no predicates, and with the following set of allowed axes:

- "attribute"
- "child"
- "descendant"

An expression belonging to this XPath fragment is

```
a//b or b/c
```

Given a context node *e*, in order to evaluate an expression of this fragment, the *startElement* event of *e* has to be reached. Moreover, each relative path can be evaluated using a simple stack whose elements maintain the node-test set that can be matched at the current depth level of the document. Each time the last node-test is matched by a node *n*, we can assert that *n* is selected by the path, and thus that the path is satisfied.

It can be observed that, as the allowed axes are just "attribute", "child" and "descendant", the nodes selected by the paths of the current XPath fragment are necessarily contained by the context node *e*. As a consequence, if by the *endElement* of *e* no node has been selected by the path, a streaming evaluator can conclusively decides that the path does not select any node using *e* as context node.

As a consequence, the result of the evaluation of this XPath subset expressions is always available by the end-tag of the context node.

Moreover, the amount of memory required to evaluate these expressions is linear in the depth of the tree fragment rooted at the context node. Such an amount of memory is the same required to decide the well-formedness of an XML document.

2.6 *Simple Functions over Downward Paths (2,3)*

In this section, we extend the XPath fragment considered in Section 2.5, allowing the use of simple functions. An expression belonging to this subset is

```
(count(.//a//b) = 5) and c
```

As it can be observed in the above example, the use of simple functions may require location paths to evaluate to node sequences, rather than booleans. However, for the evaluation of these expressions, such a requirement does not imply the usage of an amount of memory greater than that used to evaluate the expressions of the previous fragment.

Moreover, by definition the evaluation of simple functions does not require an amount of memory greater than that required to evaluate its arguments. In the XPath fragment we are considering, the only expressions that needs to visit the input document in order to be evaluated are the location paths. As the evaluation of such expressions requires an amount of memory linear in the depth of the tree rooted at the context node, the evaluation of simple functions cannot require more memory.

Thus, the memory required to evaluate this XPath fragment expressions is linear in the depth of the tree rooted at the context node. The evaluation of any of these expressions terminates by the end-tag of the context node (given that the set of allowed axes are "attribute", "child" and "descendant").

2.7 *Tail Predicates over Downward Paths (3,3)*

In this section we consider the XPath fragment obtained from the previous one, allowing the use of *tail predicates* within relative paths. An expression of this XPath fragment is

```
.//a/b//c[d or e]
```

The only features we have added w.r.t the previous XPath subset, is the usage of tail predicates. Thus we have to study their evaluation. Consider the example expression shown above. When a streaming evaluator receives the *startElement* of a node *n* found to match the last step of the location path, it cannot decide if *n* satisfies the predicate or not since the *n* children have not been visited yet. Thus, the evaluator cannot decide if *n* is selected by the main location path or not.

However, by the end-tag of *n* all the children of *n* have been visited, and thus it is possible to know if the predicate has been satisfied or not. Thus, at the end-tag of *n* it is possible to decide if it is selected by the location path or not.

Note that, when the *startElement* event of *n* is received, a streaming evaluator is not required to store it in memory, i.e. to *buffer* it.

Thus, also for this XPath fragment we can state that: all the expressions are conclusively evaluated by the end-tag of the context node *e*; the memory amount required to evaluate these expressions is linear in the depth of the tree fragment rooted at the context node.

2.8 *Inner Predicates over Downward Paths (4,3)*

We extend the XPath fragment considered in Section 2.7, adding *inner predicates*. An expression belonging to the current XPath fragment is

```
count(.[a]//*)
```

Given a context node *e*, in order to evaluate an expression of this XPath fragment, a memory amount linear in the *number* of nodes contained by *e* might be required. Indeed, consider the example expression shown above and suppose that the context node *e* does *not* contain any `a` elements. In such a case, all the nodes contained by *e* have to be selected only if *e* has a child named `a`. However, given a node *n* contained by *e*, neither by its *startElement* nor by its *endElement* events, it is possible to know if *n* is a node selected by the location path or not. Indeed, it is not yet possible to know if *e* has a child `a` or not.

As a consequence, it is necessary to buffer *n* until the end-tag of *e* is reached. At that time, a streaming evaluator can assert that *n* is not selected by the location path. Such a buffering is clearly required for any node contained in the context node *e*.

Finally, also for the current XPath fragment, in order to evaluate an expression it is necessary to visit the nodes contained by the context node: the evaluation process starts at the start-tag of the context node and terminates by its end-tag.

2.9 *Complex Functions over Downward Paths (5,3)*

Now we extend the XPath fragment defined in Section 2.8, allowing the use of complex functions.

Complex functions may require a memory amount linear in the number of document nodes, regardless of the memory required to evaluate its arguments.

However, we can observe that the evaluation the current subset expressions, terminate by the end-tag of the context node.

For the remaining XPath subsets, we will not discuss about the evaluation of complex functions.

2.10 *Boolean Expressions over Forward Paths (1,4)*

In this section we extend the XPath fragment defined in Section 2.5 (note that predicates and functions are avoided), allowing the use of forward axes. i.e. "following-sibling" and "following". An expression of the current subset is

```
descendant::a/following-sibling::b or following::a/child::b
```

Consider the above expression. We can observe that, given a context node *e*, the evaluation result might be available *after* the end-tag of *e* (e.g. if *e* does not contain any `a` nodes), and possibly at the *endDocument* (e.g. if the input document does not contain any `a` nodes).

For what concern the memory footprint, it is not difficult to find an algorithm using a memory linear in the document depth.

2.11 *Simple Functions over Forward Paths (2,4)*

Now, we extend the XPath subset defined in Section 2.10 allowing the use of simple functions.

Similarly to what seen in Section 2.6 for downward paths, we have that the streaming evaluation of expressions of the current subset requires a memory amount linear in the depth level of the document.

On the other hand, it is possible that the evaluation result is not available before the *endDocument*.

2.12 *Tail and Inner Predicate over Forward Paths (4,4)*

Now, we extend the XPath subset defined in 2.11 allowing the use of predicates. A current-subset expression is

```
following::*[count(following::a) lt 2]
```

We do not distinguish among tail and inner predicates, because also for the evaluation of expressions with tail predicates only, a relevant amount of memory can be required.

Indeed, consider the expression shown above, and suppose it has to be evaluated using a context node *e*. In order to count all the `a` nodes, it is necessary to visit all the nodes following *e*. Thus, for each node *n* following *e*, neither by its *startElement*, nor by its *endElement*, it is not possible to know if it is selected by the location path. Thus, a streaming evaluator is forced to buffer *n*. It is not difficult to find an input document s.t. the number of buffered nodes is linear in the number of nodes following the context node.

2.13 *Booleans Expressions over Forward Absolute Location Paths (1,5)*

Now, we consider the set of XPath expressions obtained combining boolean connectors over both relative and absolute forward paths without predicates.

We can observe that, given a context node *e*, an absolute location path evaluation has to start at the beginning of the document rather than at the *startElement* of *e*. Moreover, in order to find all the nodes selected by an absolute location path, a streaming evaluator is forced to visit all the document nodes. However, if a streaming evaluator is interested in the *effective boolean value* [XPath2], it is possible that the location path is satisfied even *before* the *startElement* of *e*.

In the general case it is necessary to wait for the end-document in order to know the evaluation result of an absolute path. Finally, since we are considering paths with no predicates, the required memory amount is linear in the document depth.

The same characteristics hold also if we extend the current XPath fragment with simple functions.

2.14 *Tail and Inner Predicates over Forward Absolute Location Paths (4,5)*

Now we consider inner and tail predicates. Consider the expression

```
count(//*[//a]) eq 0
```

and suppose it has to be evaluated on a document without any `a` element. The predicate is found to be false only at the end document. Before that, all the document elements have to be buffered. Clearly the required memory is linear in the size of the input document.

2.15 *Boolean Expressions over Backward Paths (1,1)*

In this section we extend the XPath subset discussed in 2.2 allowing the use of relative paths with *backward axes* only: "parent", "ancestor", "preceding-sibling", and "preceding". Note, we are not allowing forward paths. An example of such expressions is

```
preceding::a and parent::b
```

Clearly, given a context node *e*, the evaluation of an expression of the current subset cannot start at the *startElement* of *e*. However, suppose for a moment that we have a mechanism to evaluate reverse axes. Then, we can observe the evaluation result is available by the *startElement* of *e*.

Now, we propose a mechanism to handle reverse axes. We describe it by an example. Consider the expression above. Before the evaluation process starts, the expression is transformed into the absolute location path

```
//*[preceding::a and parent::b]
```

Such a location path selects all the nodes of the document satisfying the original expression. Then the obtained path is rewritten into an equivalent location path with forward axes only (see Section 2.18): [4]

```
//*[simple_let $var in . return //a[following::node() == $var] and //b[child::node() == $var]]
```

The rewritten expression may contain absolute paths with predicates. We have stated in Section 2.13 that such expressions may require the buffering of relevant portions of the input document. However, if we consider boolean operators only, the buffering can be avoided. Indeed, in such cases we are interested in th effective boolean values of location paths. For instance, consider the expression

```
//a[following::node() == $var]
```

It is not necessary to find all the `a` elements, but rather that an `a` has been found. Thus, it is equivalent to the expression

```
//a/following::node() == $var
```

which does not have any predicates and thus requires no buffering.

2.16 *Simple Functions over Backward Paths (2,1)*

In this section we extend the XPath fragment described in Section 2.15, allowing the use of simple functions. Consider the expression:

```
count(preceding::a) > 0
```

Following the ideas shown in Section 2.15, the expression is rewritten into

```
//*[simple_let $var in . return count(//a[following::node() == $var]) > 0]
```

The nodes selected by the expression `//a[following::node() == $var]` must be exhaustively determined. It implies the buffering of `a` elements. The use of the `$var` impose an upper-bound on the buffered nodes. Indeed, if we are interested in the evaluation of the original expression using e as context node, than the number of buffered nodes is linear in the number of nodes preceding e, rather than in the number of nodes of the entire document.

2.17 *XPath Functions*

XPath describes its built-in functions and operators in [XPath2Op]. The streaming evaluation of some of them requires a relevant amount of memory regardless of the memory required for the evaluation of their arguments. We call such functions *complex functions*. For the other functions, the streaming evaluation does not require an amount of memory larger than that required by the evaluation of their arguments. We call them *simple functions*.

In a function call, an argument is either a constant value or a location paths. The former is not interesting for our discussions. Thus we consider only the latter. Moreover, as we can get rid of backward axes through a rewriting process, we consider forward paths only.

2.17.1 Simple Functions

Most XPath functions are simple. Here, we discuss some of them.

For instance, consider value comparisons: it applies to two sequences, but impose restrictions on the cardinalities of both arguments. Indeed, each argument sequence must contain either zero or one item: if it contains more than one item, an error has to be thrown.

Thus, suppose we have to evaluate, using a generic node e as context node, the expression `p1 valueComp p2`, where `valueComp` is a generic value comparison operator and `p1` and `p2` are location paths. In order to evaluate this expression, there is no need to store the values of all the nodes selected by `p1` and `p2`. Indeed, a streaming evaluator can store just the value of the first node selected by `p1`, and the value of the first node selected by `p2`: the comparison is then performed on these values.

Now, consider the expression `p1 intersect p2`, where `p1` and `p2` are location paths. Suppose that in order to evaluate `p1` and `p2`, buffering is not required (e.g. predicates are disallowed). If, given a context node e, there is a node n selected by both paths, then at the *startElement* of n a streaming evaluator finds that n belongs to the intersection. Thus, the evaluator can send an output event signaling that n belongs

Figure 4: Complete list of complex functions

- fn:codepoint-to-string
- fn:string-join
- op:concatenate
- fn:index-of
- fn:distinct-values
- fn:insert-before
- fn:remove
- fn:reverse
- fn:subsequence
- fn:deep-equal
- fn:id
- fn:idref
- fn:doc
- fn:collection
- fn:position
- fn:last
- `=` (general equality)
- `!=` (general inequality)
- `for` expressions
- conditional expression

to the output sequence. Similarly, if *n* does not belong to the intersection, at its *startElement* this can be immediately verified.

On the other hand, if p1 and/or p2 requires buffering for their evaluation, also the evaluation of the intersection operator requires buffering.

In Figure 4 we provide a complete list of *complex* functions. All the other XPath functions are simple.

2.17.2 Complex Functions

Here we discuss some complex functions. Consider the expression `following::*, descendant::*` and suppose it has to be evaluated w.r.t. a context node *e*. The output of the expression must be the sequence of nodes obtained concatenating, in order, the nodes selected by the first location path, and the nodes selected by the second location path. Since the nodes selected by the second argument are visited before the nodes selected by the first argument, the nodes descending from *e* have to be buffered until all the nodes following *e* have not been visited.

Also the `for` statement can require an amount of memory larger than the memory required to evaluate its arguments. Consider the expression `for $var := //* return //*`. The resulting sequence is the sequence of all the nodes of the document concatenated with itself as many times as many nodes are in the document. The buffering of the nodes selected by the return expression is unavoidable.

In Figure 4 we list all the complex functions.

2.18 *The Rewriting Algorithm*

For the sake of brevity in this paper we do not discuss our rewriting algorithm, here we give some hints about it. It is strongly based on the ideas shown in [Olteanu02], where Dan Olteanu et al propose an algorithm to rewrite expressions of limited subset of XPath 1.0 [XPath1] into forward only location paths. We also adopt some ideas of [Olteanu01], an extended version of [Olteanu02], where some hints to handle

the complete XPath 1.0 language are provided. Indeed, our rewriting algorithm is able to handle all the expressions of the XPath 2.0 fragment equivalent to XPath 1.0, and extended with XPath 2.0 specific functions.

The expressions produced by the rewriting process use two special constructs: `simple_let` and `==`. The former is equivalent to a XPath `for` statement whose binding expression is either `self::node()` or `/`. The latter is an operator over node sequences, and it returns true iff their intersection is not empty.

§ 3 The LATVIA Typing Discipline

Here we propose an approach to the streaming validation of CTA, where the conditions used to assign types are expressions taken from the complete XPath language. The idea is to evaluate all the conditions used within a schema using a lazy XPath evaluator which emits validation outcomes as soon as they are available, according to the timing constraints of the chosen XPath subset. The events produced by all XPath evaluators, one for each condition in the Schema, are listened to by the streaming validator.

As observed in Section 1 the use of the complete XPath language to express conditions does not conform, in the general case, to the STEVE discipline currently followed by XML Schema. In this section we propose another discipline called LATVIA (*Lazy Assignment of Typing and Validity Infoset Aspects*). In LATVIA the type and the validity of an element are allowed to be decided *lazily*, i.e. the type is allowed (in the general case) to be decided at any moment after the start tag of the element, and the validity is allowed to be decided even after the end tag of the element.

Here we describe the behavior that an XML Schema validator adopting CTA may provide when following the LATVIA discipline. When the start tag of an element e is read, the validator looks for an appropriate declaration which e can be validated against. Let d be such a declaration. If d is a conditional declaration, we say that e is a *conditional element*. In such a case, d specifies several alternative type definitions. The actual type e has to be assigned to is one of such type definitions. Let T_e be the set of type definitions that can be assigned to e. If e is not conditional, we can think of T_e as a singleton, because it contains just one type definition.

If T_e contains more than one type, the STEVE discipline cannot be applied. In such a case the validator informs the downstream application about the set of all the types that may be assigned to e. Such a communication is accomplished through the emission of the output event *possibleTypes*(e, T_e). This implies that the content of e has to be validated against all the types of T_e.

During the processing of e, all expressions whose evaluations fail have the effect of removing the corresponding type from the set T_e. If at the end tag of e T_e still contains more than one type definition, the validator cannot assert the validity of e, and thus the STEVE discipline is not respected. However, for each type t of T_e the streaming validator decides whether e validates against t or not. In this way, the validator obtains a validation result for each type in T_e. The obtained results are collected in a set we call V_e. Each member of V_e is a pair (t, v), where t is a type definition and v is the result of validating e against t. V_e is communicated to the downstream application through the emission of the output event *possibleValidities*(e, V_e). In this manner, the downstream application is informed about all the possible validation results of e.

Based on the events received by the streaming evaluators, the validator decides which type a conditional element e has to be assigned, and which types e cannot be assigned. If the validator decides that an element e cannot be assigned a type t, it removes such a type from T_e and informs the downstream application about such a removal. The communication is performed through the output event *removeType*(e, t). Similarly, if the validator decides that an element e has to be assigned type t, it removes all the other types from T_e and informs the downstream application about the type assignment. Again, such a communication is accomplished through the output event *assignType*(e, t).

Figure 5: (Yet another) Conditional declaration example

```
<xs:element name="x" type="xs:anyType">
 <xs:alt cond="following::a" type="xs:decimal" />
 <xs:alt cond="child::a">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="a" type="xs:string" />
   </xs:sequence>
  </xs:complexType>
 </xs:alt>
 <xs:alt cond="preceding::a" type="xs:integer" />
</xs:element>
```

For instance, consider the element declaration of Figure 5, and suppose it has to be used to validate the element `x` of the document:

```
<r>
 <x>
  <a></a>
 </x>
</r>
```

Before the validation process starts, the validator transforms the three conditions of the declaration into three absolute location paths p1, p2, and p3, and it instantiates a streaming evaluator for each of them. We call the evaluators $eval_{p1}$, $eval_{p2}$, and $eval_{p3}$.

When the start tag of the element `x` is received by the validator, it forwards it to the three evaluators. As result, $eval_{p3}$ sends an output event to the validator, informing it that `x` is not selected by p3. It means that `x` does not satisfy the third condition of the declaration, i.e. `preceding::a`. Consequently, `x` cannot be assigned type `xs:integer`. Thus, the set of possible types of `x` T_x is `xs:decimal`, `xs:anonymousType`, where `xs:anonymousType` is the anonymous type definition. The validator sends the output event *possibleTypes*(`x`, T_x).

When the start tag of the element `a` is reached, $eval_{p2}$ informs the validator that `x` satisfies the second condition of the declaration. However, the validator cannot assert the type of `x` yet, because it has to verify the first condition.

At the end tag of `x`, T_x contains more than one type. Thus, the validator validates `x` against both types of T_x, and constructs the set V_x with the members (`xs:decimal`, `invalid`) and (`anonymousType`, `valid`). Then, it sends the event *possibleValidities*(`x`, V_x).

Finally, at end of the document, $eval_{p3}$ informs the validator that `x` does not satisfy the first condition. Consequently, the validator assigns `xs:anonymousType` to `x`, and removes `xs:decimal` from `x`. The events generated by the validator are *removeType*(`x`, `xs:decimal`) and *assignType*(`x`, `xs:anonymousType`).

The last event implicitly states that `x` is valid. Indeed, the validator previously informed the downstream application that `x` validates against `xs:anonymousType`.

§ 4 Implementation

As an assessment of the feasibility of a LATVIA-driven validator we have implemented our own prototype. It is a streaming validator of XML Schema schemata which can exhibit conditional type declarations in the syntax of Figures 3 and 5. It consumes as input a stream of SAX-like events and produces as output a stream of validation events like those discussed in Section 3.

Such a prototype is necessarily composed of two distinct parts: a frontend consisting of a streaming evaluator of XPath conditions and a backend consisting of an XML Schema validator driven by the validation events produced by the frontend. In this section we describe the two parts in turn, emphasizing our novel algorithm for the streaming evaluation of XPath conditions [Marinelli06].

Figure 6: Architecture of the streaming condition evaluator

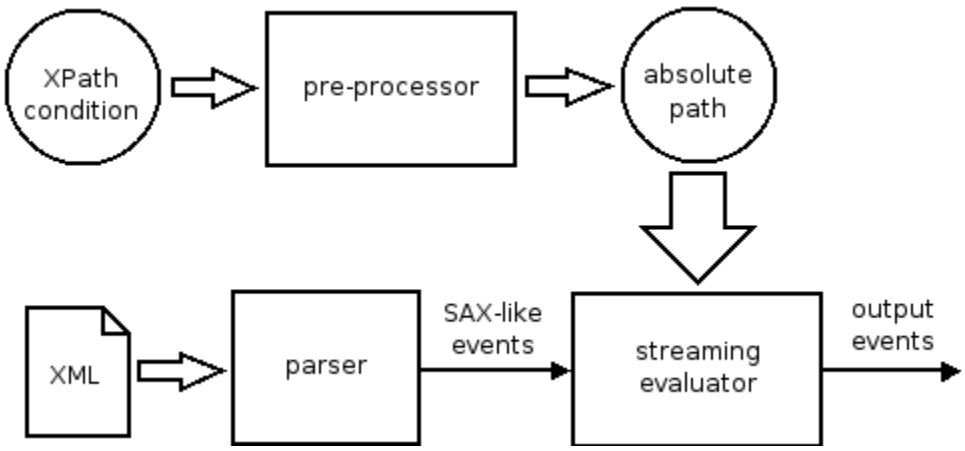


Figure 7: A conditional declaration with a backward condition.

```
<xs:element name="coordinate" type="2DCoordinate">
 <xs:alt cond="parent::geometry/@dimension eq '3D' type="3DCoordinate" />
</xs:element>
```

4.1 *Our Streaming XPath Evaluator*

Here we describe our streaming evaluator of the XPath conditions specified by a conditional declaration.

The evaluation of an XPath condition of a conditional declaration is based on the architecture depicted in Figure 6. Let c be the condition, and e the name of the conditional declaration. The condition passes through a pre-processing phase consisting of two steps. The first step transforms the condition c into the absolute location path `/descendant::` $e[c]$ selecting all the nodes whose name matches e and satisfying c. The second step transforms the obtained location path into a semantically equivalent expression with no reverse axes. The second step is performed applying the rewriting algorithm partially described in Section 2.18. Thus, the output of the pre-processing phase is an absolute location path of the form `/descendant::` $e[c']$ where c' is an expression with no reverse axes.

For instance, consider the declaration of Figure 7. An element validated against such a declaration is assigned type `3DCoordinate` if three-dimensional coordinates are required; the default type `2DCoordinate` otherwise. The condition defined within the declaration is rewritten into

```
//coordinate[simple_let $var in . return //geometry[child::node() == $var]/@dimension eq '3D''']
```

The evaluation is performed on the location path resulting from the pre-processing phase. Thus, relatively to the XPath stratification outlined in Section 2, our streaming evaluator works on expressions of the subset identified by the cell (5, 5) of the stratification table. Indeed, the rewriting process may produce expressions with absolute location paths with predicates. Moreover, general comparisons are allowed, as well as inner predicates.

The algorithm used to evaluate the location path is event-based, i.e. it receives a sequence of SAX-like events describing the XML document which the location path has to be evaluated on. The output of the algorithm is a sequence of events emitted during the evaluation process and signaling which nodes `e` are and which nodes `e` are not selected by the location path. Note that through such output events, the evaluator indirectly informs about which nodes `e` satisfy and which nodes `e` do not satisfy the original condition `c`. The input and output events are those defined 2.1.

In order to evaluate in streaming the location path resulting from the pre-processing phase, we propose an algorithm that extends χαος [Barton03] (an algorithm for the streaming evaluation of absolute location paths). We do not directly use χαος because it does not support all the features of our XPath fragment.

In the next sections we give a description of our algorithm for the streaming evaluation of an absolute location path with no reverse axes. Before that we briefly describe χαος to facilitate the reader in understanding our algorithm.

4.1.1 A Gentle Introduction to χαος

χαος is an event-based algorithm for the streaming evaluation of an absolute location path possibly with reverse axes. Its purpose is to construct a compact representation of the node-set selected by the location path.

χαος is based on a restricted subset of XPath 1.0.[5] Indeed, it does not allow axes other than "ancestor", "parent", "child", and "descendant"; no function is handled and just the `and` operator is supported. Moreover, χαος does not allow absolute location paths within a predicate of a location step.

Before the evaluation process starts, χαος creates two representations of the location path: an x-tree and an x-dag. The *x-tree* is a tree data-structure similar to an AST [Abstract Syntax Tree] of the XPath expression. Each node of the tree is called *x-node* and corresponds to the node-test of a location-step. The edges connecting the x-nodes are called *x-edges*. An x-edge connecting an x-node u to a child x-node v represents a location step from u to v, and it is denoted by (u, v). Each x-edge is labeled with the axis specifier of the corresponding location step. The *x-dag* is a *Directed Acyclic Graph* (DAG) DAG [Direct Acyclic Graph] obtained from the x-tree. It is mainly used to handle the reverse axes.

For what concerns the streaming evaluation of the location path, χαος progressively constructs the node-set it selects. As soon as a node e matches the node-test represented by an x-node v, χαος creates a data-structure called *matching-structure*, which is a pair associating v to e, denoted by (v, e). In order to know when a matching-structure has to be created, χαος makes use of the *looking-for set*. It is a set of pairs (v, l) where v is an x-node and l is either an integer or a `*`. The presence of a member (v, l) within the looking-for set means that at the depth level l a node matching v is expected.[6] Thus, if χαος receives the start-tag of an element e and within the looking-for set there is a pair (v, l) such that e matches v and its depth level satisfies l, then a matching-structure (v, e) is created. The looking-for set is then updated inspecting the outgoing edges of v and based on the depth level of e.

The looking-for set is initialized to the pair $(\mathrm{Root}, 0)$ which states that at depth level 0, the root of the document is expected. Thus, at the start document event the matching-structure $(\mathrm{Root}, 0)$ is created (0 is the identifier of the root of the document).

The matching-structures are interconnected in parent-child relationships. Given an x-edge (v, v') labeled with "descendant" and a matching-structure (v, e), if the algorithm finds that e has a descendant node e' matching the node-test v', a matching-structure (v', e') is created, and it is added as child matching-structure to (v, e). [7]

A fundamental concept of χαος is that of *total matching*. A matching-structure (v, e) is said to be a total matching if for each outgoing edge (v, v') of v, (v, e) has at least one sub-matching (v', e') that in turn is a total matching. If v has no outgoing edges, trivially (v, e) is a total matching.

In the case reverse axes are *not* present, at the end tag of an element e, it is always possible to decide if a matching-structure (v, e) is a total matching or not, because the allowed axes are downward, and thus the nodes potentially matching the x-nodes descending from v have been read. If at the end tag of e (v, e) is not a total matching, (v, e) is removed, and thus cannot figure as a sub-matching of its parent-matching.

The presence of reverse axes involves the procedure to decide if a matching-structure is a total matching. However, we do not discuss the implications of their presence, because our streaming evaluator works on a reverse-axes-free location path, and thus they are not interesting for the discussion of our algorithm.

Finally, if at the end document event the matching-structure $(\mathrm{Root}, 0)$ is found to be a total matching, then the location path has selected at least one node. The nodes selected can be obtained from the matching-structures associated to the x-node corresponding to the last location step of the input location path.

In the next sections we present our algorithm for the evaluation of location paths, emphasizing the differences with respect to χαος.

4.1.2 Absolute Location Paths in Predicates

Our algorithm supports the use of absolute location paths within a predicate. All absolute location paths are evaluated in parallel, and are represented by distinct x-trees. For this reason, we say that our algorithm represents the input location path through an *x-forest*. Moreover, if the node-test represented by an x-node v has an absolute location path p as predicate, v is connected to the x-tree t representing p through a special x-edge (v, t) called *x-tree-edge*. An x-tree-edge has no label.

In order to handle reverse axes, χαος resorts to a graph data-structure called *x-dag* representing the location path. Since our algorithm works on a reverse-axes-free location path, the x-dag is not necessary.

4.1.3 The Use of DFAs

Although our algorithm works on a location path with no reverse axes, it accepts all the forward ones (including "following" and "following-sibling"). This has some implications in deciding whether a matching-structure has a sub-matching or not. In fact, let v be an x-node with an outgoing edge (v, v') labeled with "following-sibling", and let e be a node matching v, whose depth-level is l. In order to decide that the matching-structure (v, e) has no sub-matching (v', e') it is not sufficient to wait for the end-tag of e, but rather for the end tag of the parent of e.

This behavior is implemented by our algorithm as follows. When the matching-structure (v, e) is created, a DFA able to recognize all the nodes within the "following-sibling" axis of e and matching v' is created. We say that (v, e)*depends on* such a DFA, which is deallocated when the end-tag of the parent of e is reached. If during its life-cycle the DFA has not recognized any node, then it is possible to assert that (v, e) has no sub-matching relative to the x-edge (v, v').

In general, when a matching-structure (v, e) is created, the outgoing edges of v within the x-tree are analyzed. For each edge (v, v') labeled with an axis specifier a, the algorithm instantiates a DFA able to recognize all the nodes of the axis a of e and matching v'. Such a DFA is described by the tuple (a, v', l), where l is the depth-level of e. The depth-level l is necessary for computing the transitions among the states of the automata.

Each DFA has an initial state, some intermediate states, one or more final states, and a sink state. Once the sink state is reached, the DFA is deallocated. When a DFA (a, v', l) receives the start element event of an element e', and through such input it transits into a final state, a new matching-structure (v', e') is created. Our DFAs replace the looking-for set of χαος.

4.1.4 Operators and Functions

Every operator and function present within the location path to evaluate, is represented through an x-node.

If the x-node o represents an operator or a function, the edge connecting it to its parent v is a special x-edge with no label, called *x-pred-edge*.

Now, let v be an x-node representing a location step node-test, and o be an operator used at the top-level of a predicate of such location step. If during the evaluation process a matching-structure (v, e) has to be created, the algorithm also creates a special matching-structure (o, e) called *evaluator-structure*. Such structure has the purpose of evaluating the operator o with respect to the context node e.

For an evaluator-structure, talking about total matching does not make sense. Rather, each evaluator-structure has a current value, that is updated during the evaluation process, based on the operators semantics. When the current value of an evaluator-structure cannot change anymore, we say it is *stabilized*. Once its current value is stabilized, the evaluator-structure communicates it to the parent-matching.

Now, we are going to explain the stabilization of the value of an evaluator-structure. Since the stabilization depends on the semantics of the relevant function or operator, we take into consideration a single case, i.e. that of the operator `or`. Other operators and functions are handled similarly. Consider the expression `child::a or following-sibling::b`.

Let o be the x-node representing the boolean operator `or` It has two outgoing edges: (o, a) (labeled with "child") and (o, b) (labeled with "following-sibling"), where a is the x-node representing the node-test `a` and b is the x-node representing the node-test `b`. When an evaluator-structure (o, e) is created, its value

is set to `false`. The algorithm also creates two DFAs: d_1 ("child", a, l) and d_2 ("following-sibling", b, l), where l is the depth level of e.

Now, suppose that during the evaluation process at least one of d_1 and d_2 transits into a final state. For instance, suppose that d_1 finds an element `a`. For the semantics of the `or` operator applied to node-sets, it is sufficient to assert that the current value of (o, e) is `true`, and that it is to be considered stabilized. Consequently, it is not necessary to hold d_1 and d_2 active, and thus they are deallocated.

On the other hand, in order to decide that the stabilized value of (o, e) is stabilized, it is necessary to wait that both d_1 and d_2 transit into their sink states, without having been transited into a final state.

4.1.5 Total Matching and Propagation

In our algorithm, a matching-structure has a state indicating whether it is either a *total matching*, a *potential total matching*, or a *no total matching* (i.e. it will never be a total matching). When a matching-structure becomes a total matching, it propagates its state to its parent-matching. Analogously, when it is a no total matching, its state is propagated to the parent-matching.

Let (v, e) be a matching-structure,[8] and let l be the depth level of e. (v, e) is a total matching if the appropriate case holds:

1. v has no outgoing edge;
2. for each outgoing edge (v, v'):
    1. if (v, v') is an x-pred-edge, the value of the sub-matching (v', e) has to be stabilized, and its conversion into a boolean value [9] has to return `true`;
    2. otherwise, if (v, v') is an x-tree-edge (i.e. v' is an x-tree), the matching-structure $(\mathsf{Root}_{v'}, 0)$ (where $\mathsf{Root}_{v'}$ is the x-node representing the root of the x-tree v') has to be a total-matching.
    3. otherwise (i.e. (v, v') is an x-edge as those of $\chi\alpha o\varsigma$), there is at least one sub-matching (v', e') representing a total matching.

On the other hand, (v, e) is a no total matching if v has at least one outgoing edge (v, v') such that the appropriate case holds:

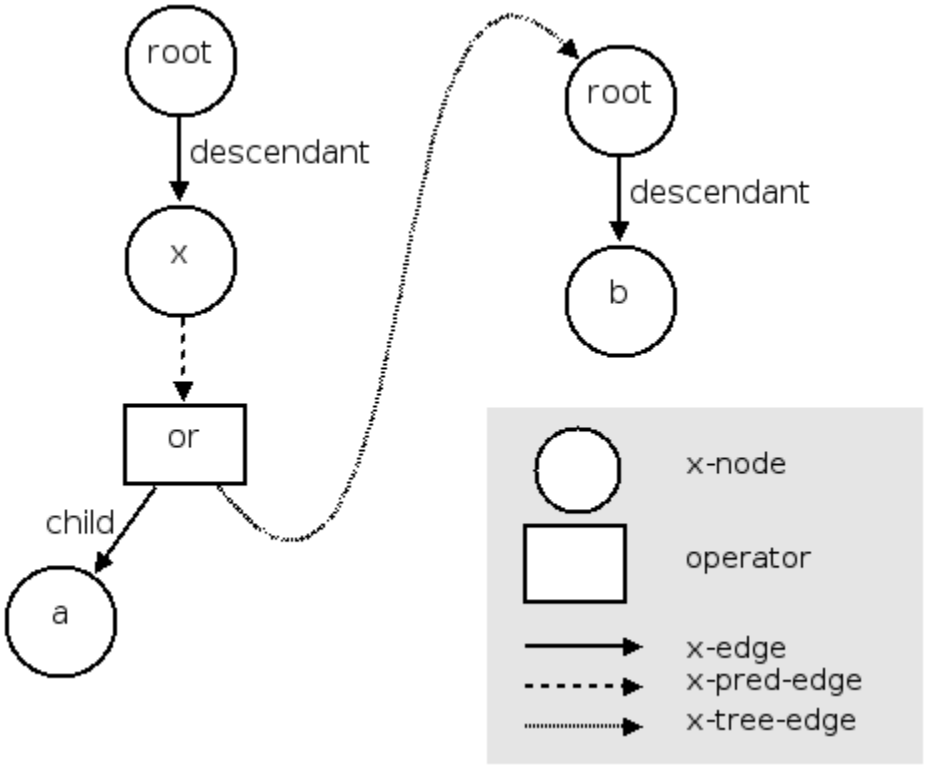
1. if (v, v') is an x-pred-edge (i.e. v' is an operator or a function), the value of the evaluator-structure (v', e) is stabilized, and its conversion into a boolean value returns `false`.
2. if (v, v') is an x-tree-edge (i.e. v' is an x-tree), the matching-structure $(\mathsf{Root}_{v'}, 0)$ is a no total matching.
3. if (v, v') is an x-edge labeled with an axis specifier a, there is no sub-matching (v', e') and the DFA (a, v', l) has been deallocated.

Note that, in order to decide if a matching-structure is a total matching or a no total matching, it is not always necessary to wait for the deallocation of all its DFAs. For instance, consider the following location path: `/descendant::x[child::a]` and suppose it has to be evaluated on the instance document:

```
<doc>
 <x>
  <a />
  ...
  <a />
 </x>
</doc>
```

When the start tag of the first element `a` is processed, the algorithm can assert the element `x` has a child `a` and thus it is a node selected by the location path. It is not necessary to wait for the end tag of `x`.

Figure 8: X-forest associated to the XPath `/descendant::x[child::a or /descendant::b]`



There are two *x-trees*: the right one represents the path `/descendant::b` .

4.1.6 Evaluation Outcome

The purpose of χαος is to construct the node-set selected by the input location path. On the other hand, the purpose of our algorithm is to signal through output events which nodes are and which are not selected by the location path.

The output events are *willNeverBeSatisfied*(id) and *satisfied*(id). The former tells that the node identified by `id` is not selected by the location path; the latter tells that the node `id` is selected by the location path.

4.1.7 Evaluation Example

In this section we show an example of execution of our algorithm. The location path is: `/descendant::x[child::a or /descendant::b]` The associated x-forest is depicted in Figure 8.

The instance document is:[10]

```
<r 1, 1>
 <x 2, 2>
  <a 3, 3></a 3, 3>
 </x 2, 2>
 <b 4, 2></b 4, 2>
 <x 5, 2></x 5, 2>
</r 1, 1>
```

The execution of our algorithm on such a document is shown in the table below, where we only consider element nodes. The start- and end-tag contain information about the name of the element, the node identifier and the depth level, and they are respectively denoted as $S_{n,\ i,\ l}$ and $E_{n,\ i,\ l}$.

Table 2: Sample execution of our algorithm to evaluate
`/descendant::x[child::a or /descendant::b]`

Input events	Comments
Start document.	The algorithm creates two matching-structures (Root, 0), one for each x-tree. An element x and an element b are now looked for.
$S_{r, 1, 1}$	Ignored.
$S_{x, 2, 2}$	A matching-structure (x, 2) is created. Since x has an outgoing x-pred-edge, an evaluator-structure (or, 2) is created too, and it is added as sub-matching of (x, 2). The matching-structure (Root, 0) of the second x-tree is added as sub-matching of (or, 2). Finally, the current value of (or, 2) is set to false. An element a child of node 2 is now looked for.
$S_{a, 3, 3}$	A matching-structure (a, 3) is created. Since the x-node a has no outgoing edges, (a, 3) is a total matching, and it propagates its state to (or, 2), which updates its value to true. According to the semantics of the or operator, the value of (or, 2) is considered stabilized. The stabilized value true is propagated to (x, 2), which is thus found to be a total matching. Since the x-node x represents the node-test of the last location step of the input location path, the algorithm outputs *satisfied*(2).
$E_{a, 3, 3}$	Ignored.
$E_{x, 2, 2}$	Ignored.
$S_{b, 4, 2}$	A matching-structure (b, 4) is created. (b, 4) is a total matching, and thus it propagates its state to the matching-structure (Root, 0) relative to the second x-tree. Such a matching-structure updates its state to total matching.
$E_{b, 4, 2}$	Ignored.
$S_{x, 5, 2}$	A matching-structure (x, 5) is created. An evaluator-structure (or, 5) is instanced and added as sub-matching of (x, 5). The matching-structure (Root, 0) of the second x-tree is added as sub-matching of (or, 5). Since (Root, 0) is a total matching, the value of (or, 5) is set to true and it is also considered stabilized. Consequently, it is propagated to the matching-structure (x, 5), which is then found to be a total matching. Consequently, the output event *satisfied*(5) is emitted.
$E_{x, 5, 2}$	Ignored.
$E_{r, 1, 1}$	Ignored.
End document.	Ignored.

4.2 *The LATVIA Model Implementation*

We propose here an implementation of a streaming validator for XML Schema schemata augmented with CTA.

As already stated, our implementation works on conditions in the subset (5, 5) of the stratification table proposed in Section 2. As a consequence, given a conditional element e, in order to know the type it is assigned, it might be required to wait for the evaluation end of a condition put on nodes following e. In such a case, the actual type of e is known *after* its end-tag. In other words, our CTA implementation follows the LATVIA discipline.

Given a conditional declaration, our implementation creates a streaming evaluator for each condition specified with the declaration. Each streaming evaluator signals the nodes satisfying the corresponding condition. Such events are those described in Section 4.1.6.

Each time our validator receives an event from an XPath evaluator stating that a node e satisfies a condition c, it checks whether the e type depends on such conditions. In such a case, the validator assigns e with the type corresponding to the condition c, and it communicates the type assignment to the downstream application. [11]

Similarly, if the validator receives an event from a streaming XPath evaluator stating that an element *e* does not satisfy a condition *c*, then it removes the corresponding type from the potential type list of *e*, and communicates the type removal to the downstream application.

The potential type list of a conditional element *e* is held in an hash table. The keys are the conditional element numeric identifiers.

The events generated by the streaming evaluator are those defined in Section 3. Conditional elements are identified through their numeric identifiers.

4.3 *Working Prototype*

We have implemented a streaming SchemaPath validator, patching the Java source code of Xerces2 [Xerces2], a streaming XML Schema validating parser. The streaming evaluation of the XPath conditions is performed by a Java implementation of the algorithm described in Section 4.1.

The implementation is a prototype. It has some limitations concerning both the streaming validator and the XPath evaluator. For instance, our validator does not correctly handle the default attributes when conditionally declared. Furthermore, some functions and operators are not handled. Such limitations have not been overcome yet but they do not invalidate the approach of the implementation.

Our SchemaPath validator and XPath evaluator are free software and are available on the web for downloaded at **http://vitali.web.cs.unibo.it/Progetti/LaTVia**.

4.4 *Related Works*

Validating XML Schema and evaluating XPath expressions in a streaming fashion have become an increasing need of applications. Applicative scenarios in which these needs have been previously shown include selective dissemination of information [Altinel00], notification systems [Nguyen01], and routing of XML packets [Snoeren01]. To our knowledge, a clear definition of how the term "streaming" is used in the context of XML processing is lacking. Previous attempts in the literature tend to stress two arguments: the sub-linearity of memory usage with respect to the input document, and the computational efficiency of the algorithms employed. The first argument is peculiar especially for applicative areas in which continuous and potentially infinite flows of XML-encoded information need to be processed, as in these areas the requirement of an in-memory representation of the whole input document is, trivially, impossible to satisfy.

The literature about both the need and the current solutions for streaming validation against XML Schema is starting to appear. A recent work is *XML Screamer* [Kostoulas06], a compiler for XML Schema producing extremely efficient validating parsers that groups, for optimization purposes, the phases of parsing, validation, and de-serialization. XML Screamer significantly differs from our work, because it produces parsers taking into account the processing of XML documents, from their scanning to their de-serialization. In a sense, we are more generic, focusing only on validation, enabling the reuse of other techniques for the other phases.

On the other hand, several previous works have addressed the issue of how to evaluate in a streaming fashion one or more XPath expressions, but all of them fail to address the full expressivity of XPath, for one reason or another. XFilter [Altinel00] is the precursor of the streaming evaluation of XPath expressions; it is based on the idea of transforming location path expressions in DFA [Deterministic Finite Automaton], but can only handle straight-line expressions (i.e. with no branching nor predicates). YFilter [Diao03] is an extension of XFilter that factorizes common parts of multiple expressions using non-deterministic finite state automata to achieve greater efficiency in processing large sets of XPath expressions (hundreds of thousands) at once. XTrie [Chan02] and TurboXPath [Josifovski05] relax the straight-line requirement and handle tree-shaped expressions and predicates. Nonetheless, all the mentioned approaches fail to handle all XPath reverse axes, i.e. axes required to "look backward" in document order with respect to the context node, like "preceding" or "ancestor".

χαος [Barton03] was the first approach to perform stream evaluation of XPath with branching, predicates, and dealing with reverse axis. It is the basis of our approach and it has been described in Section 4.1.1. In spite of being a great improvement in the field χαος still fails to address the whole range of XPath, restricting the supported axis to a strict subset of what described in the full language, handling no functions, and supporting only the `and` boolean operator. In this paper we claim that we started off extending χαος to support a larger subset of XPath. The only additional ingredient we use is a static rewriting

algorithm [Olteanu02], which we use to deal with reverse axes in XPath expressions, simplifying the internal machinery of χαος.

Notes

1. unless otherwise specified, in this paper we will use "XPath" to refer to XPath 2.0, i.e. version 2.0 of the XPath specification [XPath2]
2. For those who have followed the evolution of the XML Schema specifications this probably does not come as a surprise: such specifications have an history of being overly restrictive. Just to mention a paradigmatic example, [Bex05] discusses the effect of the UPA [Unique Particle Attribution] mechanism and arguments how the desired benefits can be obtained in a cleaner manner, broadening the set of valid Schema documents.
3. If we do not consider reverse axes, predicates on attributes are useless.
4. The simple_let construct is used to bind a variable to a *single* node of the input document.
5. As XPath 1.0 can be seen as a subset of XPath 2.0, we can state that χαος works on a limited fragment of XPath 2.0.
6. If l is $*$, the node matching v is expected at any depth level.
7. In this case, (v, e) is said the *parent-matching* and (v', e') the *sub-matching*.
8. We are not discussing about an evaluator-structure, thus v is assumed to be an x-node representing a node-test.
9. The conversion rules are those defined by XPath.
10. The numbers used within the start and end tags of the elements are respectively the node identifier and the depth level.
11. Given a condition c our streaming evaluator finds every node n of the input document such that it satisfies c. However, as homonymous local (conditional) element declaration are allowed, it is possible that an element n satisfying c has a non-conditional declaration.

Acknowledgements

The authors would like to thank Noah Mendelsohn, Michael Sperberg-McQueen, and David Ezell for comments, criticisms, and encouragements on this work and the fruitful discussions within the XML Schema Working Group. We would also like to thank Claudio Sacerdoti Coen and Jacopo Zingoni for their help with SchemaPath and co-constraint specification and management.

Bibliography

[Altinel00] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *proceedings of the 26th international conference on Very Large Data Bases (VLDB)*, pages 53--64, 2000.

[Barton03] Charles M. Barton, Philippe G. Charles, Deepak Goyal, Mukund Raghavachari, Vanja Josifovski, and Marcus F. Fontoura. Streaming XPath Processing with Forward and Backward Axes. In *proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 455--466, Bangalore, India, March 2003.

[BarYossef04] Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. On the Memory Requirements of XPath Evaluation over XML Streams. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, June 2004, Paris, France, ACM Press, pages 177-188, 2004.

[Bex05] Geert Jan Bex, Wim Martens, Frank Neven, Thomas Schwentick. Expressiveness of XSDs: from practice to theory, there and back again. In *Proceedings of WWW'05: the 14th international conference on World Wide Web*, pages 712--721, Chiba, Japan. ACP Press.

[Chan02] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal, The International Journal on Very Large Data Bases*, 11(4):354--379, 2002.

[CoConstrWiki] Co-occurrence constraints. Page on the ESW Wiki, official W3C Wiki for discussing standard specifications, **http://esw.w3.org/topic/Co-occurrence_constraints**.

[Diao03] Yanlei Diao and Michael J. Franklin. High-performance XML filtering: An overview of YFilter. *IEEE Data Engineering Bullettin*, 26(1):41--48, 2003.

[Dom] Arnaud Le Hors, Philippe Le Hégaret, et al. Document Object Model (DOM) Level 2 Core Specification. Version 1.0. **http://www.w3.org/TR/DOM-Level-2-Core/**, 13 November 2000. *W3C Recommendation*.

[FpML] Financial products Markup Language. **http://www.fpml.org/**

[Josifovski05] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML streams. *The VLDB Journal, The International Journal on Very Large Data Bases*, 14(2):197--210, 2005.

[Kostoulas06] Margaret G. Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets, and Martha Mercaldi. XML screamer: an integrated approach to high performance xml parsing, validation and deserialization. In *proceedings of WWW'06: the 15th international conference on World Wide Web*, pages 93--102, New York, NY, USA, 2006. ACM Press.

[Marinelli04] Paolo Marinelli, Claudio Sacerdoti Coen, and Fabio Vitali. SchemaPath, a Minimal Extension to XML Schema for Conditional Constraints. In *proceedings of WWW'04: the 13th international conference on World Wide Web*, New York, May 2004.

[Marinelli06] Paolo Marinelli and Stefano Zacchiroli. Co-constraint validation in a streaming context. In *proceedings of XML 2006*, Boston, MA, December 2006.

[Nguyen01] Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, and Mihai Preda. Monitoring XML data on the web. In *proceeding of the 2001 ACM SIGMOD international conference on Management of data*, pages 437--448, 2001.

[Olteanu01] Dan Olteanu, Holger Meuss, Tim Furche, and Francois Bry. Symmetry in XPath. Technical Report PMS-FB-2001-16, Institute of Computer Science, University of Munich, Munich, Germany, 2001.

[Olteanu02] Dan Olteanu, Holger Meuss, Tim Furche, and Francois Bry. XPath: Looking forward. In *proceedings of the EDBT Workshop on XML Data Management (XMLDM)* Prague, Czech Republic, March 2002, volume 2490 of Lecture Notes in Computer Science, pages 109--127. Springer-Verlag.

[Sax] The SAX project **http://www.saxproject.org**, 2007.

[Snoeren01] Alex C. Snoeren, Kenneth Conley, and David K. Gifford. Mesh based content routing using XML. In *proceedings of SOSP: the 18th ACM Symposium on Operating Systems Principles*, October 2001, pages 160--173, 2001.

[UBL] Jon Bosak, Tim McGrat, and G. Ken Holman. Universal Business Language v2.0, December 2006. *Oasis Standard*.

[Xerces2] Xerces2 Java parser. **http://xerces.apache.org/xerces2-j/**, 2007. The Apache XML project.

[XHTML] Steven Pemberton *et al*. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). **http://www.w3.org/TR/xhtml1/**, January 2000. *W3C Recommendation*.

[XMLSchema] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema 1.1 part 1: Structures. **http://www.w3.org/TR/2006/WD-xmlschema11-1-20060831/**, August 2006. *W3C Working Draft.*

[XPath1] James Clark and Steve DeRose. XML Path Language (XPath) version 1.0. **http://www.w3.org/TR/xpath**, November 1999. *W3C Recommendation*.

[XPath2] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. **http://www.w3.org/TR/2006/CR-xpath20-20060608/**, June 2006. *W3C Candidate Recommendation*.

[XPath2Op] Ashok Malhotra, Jim Melton, Norman Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. **http://www.w3.org/TR/xpath-functions/**, January 2007. *W3C Recommendation*.

[XSLT] James Clark. XSL Transformations (XSLT). **http://www.w3.org/TR/xslt**, November 1999. *W3C Recommendation*.

The Authors

Paolo Marinelli
University of Bologna, Department of Computer Science
Mura Anteo Zamboni 7
40127
Bologna
Italy
pmarinel@cs.unibo.it

Paolo Marinelli holds a Master Degree in Computer Science at the University of Bologna. The topic of his Master Thesis regards SchemaPath, the conservative extension of XML Schema for expressing conditional content models and co-constraints, partially described in this paper.

Fabio Vitali
University of Bologna, Department of Computer Science
Mura Anteo Zamboni 7
40127
Bologna
Italy
fabio@cs.unibo.it

Fabio Vitali is an associate professor at the Department of Computer Science at the University of Bologna. He holds a Laurea degree in Mathematics and a Ph.D. in Computer and Law, both from the University of Bologna. His research interests include markup languages; distributed, coordinated systems; and the World Wide Web. He is the author of several papers on hypertext functionalities, the World Wide Web, and XML.

Stefano Zacchiroli
University of Bologna, Department of Computer Science
Mura Anteo Zamboni 7
40127
Bologna
Italy
zacchiro@cs.unibo.it

Stefano Zacchiroli is a PhD candidate in Computer Science at the University of Bologna. His thesis, titled *User Interaction Widgets for Interactive Theorem Proving*, sits at the intersection of the type theory and human computer interaction fields. His research interests also encompass markup languages and in particular co-constraints and overlapping markup for XML-based languages.

Extreme Markup Languages 2007®
Montréal, Québec, August 7-10, 2007
This paper was formatted from XML source via XSL
by Mulberry Technologies, Inc.