Co-Constraint Validation in a Streaming Context

Paolo Marinelli
pmarinel@cs.unibo.it
Department of Computer Science, University of Bologna

Stefano Zacchiroli
zacchiro@cs.unibo.it
Department of Computer Science, University of Bologna

Abstract

In many use cases applications are bound to be run consuming only a limited amount of memory. When they need to validate large XML documents, they have to adopt streaming validation, which does not rely on an in-memory representation of the whole input document. In order to validate an XML document, different kinds of constraints need to be verified. *Co-constraints*—which relate the content of elements to the presence and values of other attributes or elements—are one such kind of constraints.

In this paper we propose an approach to the problem of validating in a streaming fashion an XML document against a schema also specifying co-constraints. We describe how the streaming evaluation of co-constraints influences the output of the validation process. Our proposal makes use of the validation language SchemaPath [11], a light extension to XML Schema [14], adding conditional type assignment for the support of co-constraints. The paper is based on the description of our streaming SchemaPath validator [10].

1 Introduction

XML is frequently adopted as a format for data-exchange among applications. Such applications process the received data for their own purposes, e.g. applying queries or filters. In general, an application can profit by the knowledge that the processed document has been validated against a schema.

There are contexts where an application needs to process an XML document in a streaming fashion. This occurs, for instance, when the application receives a continuous flow of XML-formatted information. A concrete example is represented by a monitoring sensor producing data for some interested applications [13]. When the input XML data is very huge, such an application cannot locally store the incoming flow of data, because it would require large amounts

of memory. Rather, the application should use a streaming parser, e.g, a SAX parser.[1]

In order to exploit the knowledge of the validity of a document against a schema in a streaming context, also the validation process has to be performed in a streaming fashion. This means that the validator has to validate the input document based on a sequence of input events describing the document (e.g. SAX events), and without relying on an in-memory representation of the entire document.

Generally speaking, an XML document is said to be *valid* if it complies with a set of various kinds of constraints. At the very minimum this usually means that the logical structure of the document has to conform to a grammar (i.e. a set of *structural constraints*), such as those that can be defined using *grammar-based* languages (e.g. DTD [5], XML Schema [14, 4] and RELAX NG [8]). Additionally the elements and attributes of the document may be subject to complex rules constraining the set of their legal datavalues (*datavalues constraints*). Furthermore, the document may be subject to the called *co-occurrence constraints* (commonly referred to as *co-constraints*). A co-constraint is a rule governing the content of an element or attribute based on the presence or values of other attributes and elements. An example of co-constraint is the following:

> if element `def` has an attribute `href` then its content must be empty, otherwise its content must satisfy the content model `def.type`

In this paper we address the problem of streaming validation of XML documents. We describe how the problem of verifying in streaming that an XML document conforms to structural, datavalues, and co-occurrence constraints can be approached.

As the applicative case for our solution we have chosen SchemaPath [11] as validation language. SchemaPath is a validation language that extends XML Schema in a minimal and conservative way, adding support for co-constraints. The extension boils down to the addition of *conditional declarations*. They conditionally assign a type to elements and attribute, based on the evaluation of XPath predicates [7] on the instance document.

As shown in [11], the conditional type assignment represents a good approach to address a large set of co-constraints use cases. Moreover, SchemaPath inherits from XML Schema the constructs for the definition of a grammar constraining the logical structure of valid documents, as well as those for the definition of datatypes. This means that SchemaPath allows to express structural, datavalues, and co-occurrence constraints.

In this paper we discuss how an XML document can be validated against a SchemaPath specification in a streaming fashion. Our discussion is based on the description of our prototype streaming SchemaPath validator [10].

The remainder of the paper is structured as follows. In Section 1 we give a description of the SchemaPath syntax and semantics. We also shortly describe

[1] `http://www.saxproject.org/`

```
<xs:element name="def">
  <xs:alt cond="@href"    priority="2.0">
    <xs:complexType />
  </xs:alt>
  <xs:alt type="def.type" priority="1.0"/>
</xs:element>
```

Figure 1: Sample element conditional declaration in SchemaPath

there the streaming environment where a streaming validator works. In Section 3 we describe how our validator communicates the result of the validation process to the interested applications while in Section 4 we describe how it evaluates the XPath predicates associated to conditional declarations. In Section 5 we give some technical details of our implementation. Then, in Section 6 we describes some related works, mainly concerning the streaming evaluation of XPath expressions. We conclude the paper in Section 7, where we also sketch our future development lines.

2 Background

2.1 An Introduction to SchemaPath

SchemaPath [11] is a validation language minimally and conservatively extending XML Schema, the schema language directly backed by the W3C. SchemaPath extends XML Schema adding the concept of *conditional declarations* and the new `xs:error` built-in type.

Conditional declarations are used to conditionally assign a type to elements and attributes of the instance document. Indeed, a conditional declaration specifies several type definitions, each associated with a condition and a priority. The conditions are expressed as XPath predicates and they are evaluated on the instance document. A conditionally declared item is assigned the type associated with the holding condition with the highest priority.

In Figure 1 an example of conditional declaration (implementing the co-constraint example of Section 1) is shown. The declared element `def` is assigned an anonymous empty type if it has an attribute `href`; otherwise (no attribute `href` is present) it is assigned the type `def.type`.

Each element `xs:alt` of a conditional declaration represents a so called *alternative*, and it specifies:

1. an XPath condition through the attribute `cond` (if not present, it defaults to the always-true condition `true()`);

2. a type definition: it is either referenced by name through the attribute `type`; or it is anonymously defined through the constructs provided by XML Schema.

```
<xs:element name="a">
  <xs:alt cond=".//a" type="xs:error" />
  <xs:alt             type="a.type"   />
</xs:element>
```

Figure 2: Sample usage of `xs:error` within a conditional declaration

3. a priority, i.e. a real number specified through the attribute `priority`. It is used to disambiguate those situations where multiple alternatives of the declaration are satisfied. If an explicit priority is not provided, an implicit one is added by SchemaPath: the more the specificity of the XPath condition, the higher the priority.

Within each conditional declaration there is a *default alternative*, chosen when all the other alternatives are excluded. Such an alternative specifies the always-true condition `true()`, a priority lower than the priorities of all the other alternatives, and the type `xs:error`.

`xs:error` is a simple built-in type added by SchemaPath, whose lexical and value spaces [4] are empty, i.e. each item of the instance document assigned such a type has to be considered invalid. Within a conditional declaration `xs:error` is used to express a condition we do not want to be satisfied by the instance document. For example, consider one of the most famous element prohibitions stated in the specifications of XHTML: an element `a` "must not contain other `a` elements".

In SchemaPath this prohibition can be enforced using the conditional declaration of Figure 2 in which the declared element `a` is assigned type `xs:error` if it contains another element `a`. Otherwise, it is assigned type `a.type`, which is meant to define the logical structure of `a`.

2.2 Validation Environment

In this section we describe a generic environment where a streaming validator works. Our validator works in a very similar environment, which can be generalized to the one described here.

A streaming validator usually works within a pipeline composed by three components: a parser, a validator and an application interested in the result of the validation (a downstream application). Each component of the pipeline receives a sequence of input events and sends a sequence of output events to the next component. The parser has the purpose of reading the input XML document. It generates events such as those generated by a SAX parser. We assume the events generated by the parser are:

`startDocument()` to communicate the beginning of the document;

`endDocument()` to communicate the end of the document;

`startElement(name, id, level)` to communicate the start-tag of an element and its name, node-id and depth-level;

`endElement(name, id, level)` to communicate the end-tag of an element;

`characters(text, id)` to communicate the presence of a text node, and the actual text characters;

`attribute(name, value, id)` to communicate the presence of an attribute node, and its name, value and node-id.

Such events are sent to the validator, which processes them in order to perform validation. For each received event, the validator sends it to the interested application. The validator can also output events other than those received in input in order to inform the downstream application about the outcome of the validation process. For instance, the validator can inform it about the validity of an element, the presence of a validation error, and so on.

The exact set of output events generated by a validator depends on the relevant validation language and its semantics. For instance, an XML Schema validator could send a different set of events with respect to a DTD validator. In the next sections we describe the output events generated by our streaming SchemaPath validator.

3 Streaming SchemaPath Validation Outcome

As in XML Schema, the reuslt of a SchemaPath validation is the production of a *Post Schema Validation Infoset* (PSVI). Each element and attribute of the instance document is associated with a set of properties (the so called PSVI properties) describing, e.g. the type definition used to validate the item, the validity state of the item, and the list of validation errors encountered during the validation of the item.

With respect to XML Schema, SchemaPath does not change the set of PSVI properties associated to elements and attributes of the instance document. In fact, the alternatives of a conditional declaration do not survive the validation process: they are used only to decide the proper type to assign to an element. In a way, they are similar to a type attribution via the `xsi:type` mechanism.

Therefore the output events generated by our streaming SchemaPath validator concerns the PSVI properties associated to elements and attributes of the document. Due to space constraints in this paper we do not consider the full set of PSVI properties, but rather a minimal one: `[type definition]` and `[validity]`. For a given element or attribute node, `[type definition]` indicates the type definition used to validate it; `[validity]` indicates the validity state of the item, i.e. `valid`, `invalid`, or `notKnown`: the third value is used when it is not possible to decide whether the item is valid or not.

When our validator receives a `startElement()` event for an element e it sends the event to the downstream application. Furthermore, it searches for a

```
<xs:element name="quantity">
  <xs:alt cond="following-sibling::unit='meters'" type="xs:decimal"
          priority="2.0" />
  <xs:alt cond="following-sibling::unit='items'"  type="xs:integer"
          priority="1.0" />
</xs:element>
```

Figure 3: Sample conditional declaration with conditions on following elements

type definition to use to validate e. If such a type is found, the validator also outputs the event `typeDefinition(type)`, informing the downstream application that the value of the property `[type definition]` of e is `type`.

Similarly when the validator receives an `endElement()` for e, it sends the event downstream and checks whether the content of e satisfies the associated type definition. If the check succeeds, the validator outputs the event `validity(valid)`, otherwise it outputs the event `validity(invalid)`.

However, the presence of conditional declarations affects the validation process meaning that our validator is not always able to decide the PSVI property `[type definition]` for an element node at the corresponding `startElement()` event. Similarly, our SchemaPath validator cannot be always able to decide the `[validity]` property of an element at the corresponding `endElement()` event. This is is not due to a limitation of the implementation, but is rather intrinsic in certain co-constraints, as we are going to see in Section 3.1.

3.1 Other Output Events

Consider the following co-constraint:

> if the element `quantity` is followed by a sibling `unit` with value `"meters"`, it has to be considered a decimal number; otherwise, if `unit` has value `"items"`, `quantity` has to be considered an integer.

It is properly defined by the element declaration of Figure 3. Let's leave aside SchemaPath for the moment, and suppose that the co-constraint has to be checked on the following `quantity` element:

```
<invoiceLine>
  <quantity>10.2</quantity>
  <unit>meters</unit>
</invoiceLine>
```

According to its definition `quantity` has to be treated as a decimal.

Within a streaming environment, when the start-tag for `quantity` is read, it is not possible to know if the element has to be considered a decimal or an integer, because the element `unit` has not been read yet. Note that this is

not a limitation of the SchemaPath language (that we are still ignoring for the moment), but rather an intrinsic property imposed by the co-constraint itself.

In this case, the most reasonable behaviour for any streaming validator is to inform the downstream application that the type of `quantity` is either a decimal or an integer but that it is not possible to decide among the two.

Similarly when the end-tag of `quantity` is read, it is not yet possible to know if the element has to be considered a decimal or an integer. Consequently, it is not possible to know if `quantity` is valid or invalid. Indeed, `"10.2"` is considered a decimal but not an integer. Again, this is due to the nature of the co-constraint itself.

The most reasonable behaviour for a validator in this case is to inform the downstream application that `quantity` may be valid or invalid, depending on the value of its sibling `unit`.

We now come back to SchemaPath and our streaming validator. According to the declaration of Figure 3, at the `startElement()` event for `quantity`, our validator does not know if the type to assign to the element is either `xs:decimal`, `xs:integer`, or `xs:error` (the type assigned by the default alternative, see Section 2.1). Our validator informs the downstream application about all the types that may be assigned to `quantity`, sending the following output events:[2]

```
startElement(quantity, 5, 3)
possibleTypeDefinition(xs:decimal, alt1)
possibleTypeDefinition(xs:integer, alt2)
possibleTypeDefinition(xs:error,   alt3)
```

where `alt1`, `alt2`, and `alt3` represent respectively the first, the second and the default alternatives of the declaration.

When the validator receives the `endElement` event for `quantity`, it is still unable to decide which condition `quantity` satisfies. Consequently, it cannot know if the value `"10.2"` has to be validated against `xs:decimal`, `xs:integer` or `xs:error`, and thus it cannot know if `quantity` is valid. Our streaming processor performs the validation against all the three types and outputs the following events:

```
endElement(quantity, 5, 3)
possibleValidity(valid, alt1)
possibleValidity(invalid, alt2)
possibleValidity(invalid, alt3)
```

informing the downstream application about the fact that `quantity` has to be considered valid if the first alternative will be chosen, invalid otherwise.

Finally, when the validator receives the end element event for `unit` it processes the event and as result obtains that `quantity` satisfies the XPath condition of the first alternative (it has a following sibling `unit` with value `"meters"`). Since the first alternative has the greatest priority among the alternatives of the

[2] We assume `quantity` is the fifth node of the document and is located at depth level 3.

```
<xs:element name="quantity">
  <xs:alt cond="preceding-sibling::unit='meters'" type="xs:decimal"
          priority="2.0" />
  <xs:alt cond="preceding-sibling::unit='items'"  type="xs:integer"
          priority="1.0" />
</xs:element>
```

Figure 4: A conditional declaration with conditions using backward axes.

declaration, the validator can assert that the second and the default alternatives can be discarded and thus that the first alternative has to be chosen. The validator communicates these results to the downstream application through the events:

```
removeAlternative(5, alt2)
removeAlternative(5, alt3)
assignAlternative(5, alt1)
```

Note that the downstream application is informed that the node identified by 5 (i.e. `quantity`) is assigned the first alternative, and thus the application can infer that the type definition of `quantity` is `xs:decimal` and the element has to be considered valid.

4 Evaluating the XPath Conditions

In this section we describe how our validator evaluates the XPath conditions used by the SchemaPath specification.

An element conditionally declared is assigned the proper type definition on the base of the evaluation of the XPath conditions specified within its conditional declaration. Consequently, the result of the validation of the element depends on the evaluation of such XPath predicates. Thus, within a streaming validation environment it is crucial to evaluate the XPath conditions of a conditional declaration in a streaming fashion.

Unfortunately streaming evaluation of XPath is not so easy to implement. Indeed an XPath predicate can be used to specify a condition on nodes preceding (in document order) the item for which it has to be evaluated. This occurs when the XPath expression makes use of location-steps with *backward axes* (i.e. `ancestor::`, `parent::`, `preceding::`, etc).

In Figure 4 a conditional declaration making use of "backward conditions" is shown. Through that declaration, an element `quantity` is assigned one among two alternative types on the base of the value of a sibling element `unit` *preceding* it.

Suppose the declaration of Figure 4 has to be used by the streaming SchemaPath processor to validate the following `quantity` element:

```
<invoiceLine>
  <unit>meters</unit>
  <quantity>10.2</quantity>
</invoiceLine>
```

The validator cannot start to evaluate the XPath predicates of the declaration *after* the start-tag of the element `quantity` has been read: it would be too late. Indeed, the nodes selected by the location-step `preceding-sibling::unit` (used within both the conditions) would have been already read by the parser. Since the validator does not rely on an in-memory representation of the entire document and the parser cannot read multiple times a given portion of the XML document, the validator would not be able to recognize the nodes selected by the backward location-step, and thus it would not be able to decide which condition the element `quantity` satisfies.

In order to overcome this kind of difficulties, the XPath conditions of a SchemaPath specification pass through a pre-processing phase performed before the validation process starts, and able to produce an XPath expression with no reverse axes. The evaluation is then performed on the obtained expression.

4.1 XPath Pre-Processing

Each XPath predicate of a conditional declaration is transformed into an absolute location-path, which is in turn transformed into a semantically equivalent location-path with no reverse-axes.

For example, consider the first alternative of the element declaration of Figure 4. The obtained absolute location-path is the following:

```
/descendant::quantity[preceding-sibling::unit="meters"]
```

In order to remove the reverse-axes, such an expression is then transformed through some rewriting rules into the following location-path:[3]

```
/descendant::quantity[/descendant::unit[self::node()="meters"]/
                      following-sibling::node() == self::node()]
```

Note that the obtained expression selects all the elements of the instance document named `quantity` and preceded by a sibling `unit` with value `"meters"`.

The rewriting algorithm actually used in our implementation is that presented in [12], it is called *rare* and it is mainly based on the symmetry relating reverse and forward axes.

4.2 Streaming Evaluation

Our validator does not evaluate the XPath predicates as expressed within the conditional declarations, but for each of them it evaluates an absolute location-path with no reverse axes.

[3] The binary operator `==` expresses the node equality based on identity. An expression $p1$ `==` $p2$ is true iff the location-path $p1$ selects a node also selected by the location-path $p2$.

To be more precise, the validator delegates the evaluation of a location-path to a streaming XPath evaluator. The validator sends to each evaluator (there is an evaluator for each XPath condition present in the SchemaPath specification) the input events it receives from the parser.

During the evaluation process of a location-path, an evaluator informs the validator (through events) about which nodes are selected and which nodes are not selected by the location-path. For example, consider the absolute location-path shown in Section 4.1. For each `quantity` element of the instance document, the streaming evaluator informs the validator whether the element belongs to the node-set selected by the location-path or not. Based on this information, the validator decides which alternatives have to be discarded and which alternative has to be assigned to the element.

The algorithm used to evaluate an absolute location-path is an extended version of $\chi\alpha o\varsigma$ [2]. In our algorithm, each time a (forward) location-step has to be evaluated with respect to a context node, a Deterministic Finite State Automata (DFA) recognizing all the nodes selected by the location-step is created. The DFA accepts as input the events produced by the parser of the XML document. It has an initial state, one or more final states, some intermediate states, and a sink state. The transitions from a state to another depends on the axis of the corresponding location-step and the depth-level of the context node (i.e. a DFA is parameterized on the depth-level of the context node for which it has to be evaluated). For every DFA there is a sequence of input events making it transiting into the sink state. Once such a state is reached, the DFA can be discarded.

If with a start element event for a node n, a DFA transits into a final state, the appropriate actions are taken:

- if the location-step corresponding to the DFA is followed by another location-step a DFA, parametric on the depth-level of n, for such a location-step is created;
- if the location-step corresponding to the DFA has a predicate containing a location-path, a DFA for the first location-step of such a location-path is created and it is parameterized on the depth-level of n;
- if the location-step corresponding to the DFA has no predicate and it is the last location-step of a location-path, then n is a node selected by such location-path.

The algorithm is more involved than described here. In fact, it handles the presence of absolute location-paths within a predicate of a location-step, and it also handles the functions and operators defined by XPath 1.0. However, because of space limitations we do not discuss these aspects.

5 Implementation

Our streaming SchemaPath validator has been realized patching the source code of Xerces2 [15], a streaming XML Schema validating parser. We also implemented the streaming evaluator for absolute location-paths, used by the validator.

The implementation is a prototype, whose purpose is to show that SchemaPath allows a streaming validation. It has some limitations concerning both the SchemaPath validator and the XPath evaluator. For instance, our validator does not correctly handle the default attributes conditionally declared. The XPath evaluator does not implement the algorithm *rare* used to transform a location-path into a reverse-axes-free one. 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 validator can be downloaded from: `http://tesi.fabio.web.cs.unibo.it/Tesi/TesiMarinelliSpecialistica`.

6 Related Work

The problem of XML streaming validation has already been faced. In [6, 13] the authors investigate the problem of validating an XML document against a DTD in a streaming fashion, using finite-state machines. However, the authors consider only structural constraints, leaving aside co-constraints. Similarly, Xerces2 [15] provides a streaming XML Schema validator, but XML Schema does not support co-constraints.

For what concerns the streaming evaluation of XPath expressions, several research groups have already faced the problem. In [2] the authors propose $\chi\alpha o\varsigma$, an algorithm for the streaming evaluation of absolute location-paths. $\chi\alpha o\varsigma$ is the algorithm we have modified and extended in order to evaluate our XPath expressions in our validator. Its main contribution is represented by the ability of handling backward and forward axes. However, the subset of XPath considered is limited with respect to the purposes of SchemaPath. Indeed, the handled axes are `ancestor::`, `parent::`, `child::`, and `descendant::`; $\chi\alpha o\varsigma$ handles just one boolean operator (`and`); no XPath function is considered; and the predicates of a location-step cannot contain an absolute location-path. Furthermore, the purpose of $\chi\alpha o\varsigma$ is to create a compact memory representation of the node-set selected by the evaluated XPath expression. On the other hand, our XPath evaluator has the purpose of signaling during the evaluation process which nodes are and which nodes are not selected by the location-path.

Other contributions come from the research on the problem of *Selective Dissemination of Information* (SDI), where a system receives a streamed XML document and has to send it to a set of registered users, on the base of their profiles. In [1] the authors propose XFilter, an algorithm for the streaming evaluation of a large number (thousands) of XPath expressions (each expression represents a user profile) on a XML document. XFilter transforms each XPath

into a nondeterministic finite automata, accepting SAX events as input. The XPath subset considered in [1] accepts only forward axes, and thus ignore the problem of backward axes.

In [9] the authors propose another technique for the streaming evaluation of a large set of XPath queries on a XML document. The idea is to collect all the XPath expressions and to transform them into a unique *Deterministic Pushdown Automata* (PDA) accepting SAX events as input, and whose states are computed lazily during the evaluation process. The main contribution concerns the time-saving in the evaluation of those predicates common to many XPath queries. Again, the subset of XPath considered in [9] excludes backward axes.

In [12] the authors do not propose an algorithm for the streaming evaluation of XPath expressions, but rather a rewriting algorithm called *rare* able to transform an XPath expression with reverse axes into an equivalent expression with no reverse axes. Each rule is mainly based on the symmetry relating forward and backward axes. As already discussed, *rare* is used by our validator in order to pre-process the XPath conditions used within the SchemaPath specification.

7 Conclusions

In this paper we proposed an approach for validating an XML document in a streaming fashion against a schema specifying co-constraints. Our approach relies on SchemaPath as the validation language, since it allows to define co-constraints and structural and datavalues constraints.

We described our streaming SchemaPath validator, obtained modifying Xerces2. In particular, we discussed the approach to the streaming verification of co-constraints, and the streaming evaluation of the XPath conditions.

We think our work represents a contribution to the problem of the streaming validation of XML documents, because it expressly faces the issue of evaluating co-constraints in a streaming fashion.

Our work is also an important contribution for the SchemaPath language, because it demonstrates how it can be used in those contexts where a streaming validation is required. However, some remarks are worth.

As discussed in Section 3, there are cases where a conditionally declared element cannot be assigned a type definition during the processing of its start element event. It means that the downstream application is informed about the presence of a new element, for which it is temporary impossible to know the type. Consequently, the application cannot decide neither the required amount of memory for the content of the element, nor how such memory has to be structured. However, our validator informs the application about all the possible types that can be assigned to the new element. As more events are processed, the validator informs the application about the type definitions becoming impossible. Thus, the downstream application, after allocating a memory area for each possible type, can then progressively get rid of the superfluous allocated memory areas.

There are also cases where during the processing of the relevant end element

event, it is not possible to assign a type to a conditionally declared element, and thus it is not possible to decide if the closed element is valid or not. In such cases, the downstream application does not know if the information represented by the element has to be considered valid or not. What has to be studied are the effects of such cases on a downstream application.

Another future work is to consider XPath 2.0 [3] as language to express the conditions of the conditional declarations. XPath 2.0 makes use of a type system based on XML Schema. Thus, it has to be studied whether the type system of the language is compatible with the conditional type assignment of elements and attributes. Furthermore, we will study if our approach of streaming evaluation of XPath 1.0 expressions works also for the streaming evaluation of XPath 2.0.

Finally, our prototype implementation has some limitations, implementative work has to be done in order to overcome them.

8 Acknowledgements

We thank Prof. Fabio Vitali, the supervisor of our work and the main inventor of SchemaPath.

References

[1] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 53–64, Cairo, Egypt, September 2000. Morgan Kaufmann Publishers Inc.

[2] C. M. Barton, P. G. Charles, D. Goyal, M. Raghavachari, V. Josifovski, and M. 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.

[3] A. Berglund, S. Boag, D. Chamberlin, M. F. F. M. Kay, J. Robie, and J. Siméon. *XML Path Language (XPath) 2.0*. `http://www.w3.org/TR/2006/CR-xpath20-20060608/`, June 2006. W3C Candidate Recommendation.

[4] P. V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes Second Edition*. `http://www.w3.org/TR/xmlschema-2`, October 2004. W3C Recommendation.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. cois Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. `http://www.w3.org/TR/REC-xml`, August 2006. W3C Recommendation.

[6] C. Chitic and D. Rosu. On validation of XML streams using finite state machines. In *Proceedings of the 7th International Workshop on the Web*

and Databases: colocated with ACM SIGMOD/PODS 2004, volume 67, pages 85–90, Paris, France, June 2004. ACM Press.

[7] J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0*. `http://www.w3.org/TR/xpath`, November 1999. W3C Recommendation.

[8] J. Clark and M. Murata. RELAX NG. `http://relaxng.org`, 2001.

[9] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *Proceeding of ACM SIGMOD Conference on Management of Data*, pages 419–430, San Diego, California, 2003. ACM Press.

[10] P. Marinelli. Validazione in streaming del linguaggio SchemaPath. Master's thesis, University of Bologna, 2006. Dissertation in English.

[11] P. Marinelli, C. Sacerdoti Coen, and F. Vitali. SchemaPath, a Minimal Extension to XML Schema for Conditional Constraints. In *Proceedings of the Thirteenth International World Wide Web Conference*, New York, May 2004.

[12] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*, volume 2490 of *LNCS*, pages 109–127, Prague, Czech Republic, March 2002. Springer-Verlag.

[13] L. Segoufin and V. Vianu. Validating streaming XML documents. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 53–64, Madison, Wisconsin, June 2002. ACM Press.

[14] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures Second Edition*. `http://www.w3.org/TR/xmlschema-1/`, October 2004. W3C Recommendation.

[15] *Xerces2 Java Parser*. `http://xerces.apache.org/xerces2-j/`, 2006. The Apache Project XML.