XML Reasoning Made Practical

Pierre Genevès [1], Nabil Layaïda [2]

[1] *CNRS*
Grenoble, France
`pierre.geneves@inria.fr`

[2] *INRIA*
Grenoble, France
`nabil.layaida@inria.fr`

***Abstract*— We present a tool for the static analysis of XPath queries and XML Schemas. The tool introduces techniques used in the field of verification (such as binary decision diagrams) in order to efficiently solve XPath query satisfiability, containment, and equivalence, in the presence of real-world XML Schemas. The tool can be used in query optimizers, in order to prove soundness of query rewriting. It can also be used in type-checkers and optimizing compilers that need to perform all kinds of compile-time analyses involving XPath queries and XML tree constraints.**

I. Introduction

One of the biggest challenges in XML research today is to develop automated and tractable techniques for ensuring static type safety and optimization of programs that manipulate XML data [1]. To this end, there is a need to solve some basic reasoning tasks involving complex constructions such as XML schemas and powerful navigational primitives (XPath queries). Every future compiler of XML manipulating programs will have to routinely solve problems such as:

- XPath query emptiness in the presence of a schema [2]: if one can decide at compile time that a query is not satisfiable then subsequent bound computations can be avoided,
- query equivalence, which is important for verifying soundness of query reformulation and optimization,
- path type-checking, which is a basic property needed for ensuring at compile-time that invalid documents can never arise as output of XML processing code.

All of these problems are known to be computationally heavy (when decidable) [2], [3], and the related algorithms are often tricky. We present a tool for XML/XPath static analysis based on our earlier theoretical work [3], [4]. The tool has been implemented in Java and uses symbolic techniques (binary decision diagrams) in order to enhance its performance. It is capable of comparing path expressions in the presence of real-world schemas (such as the W3C SMIL and XHTML language recommendations). The tool supports XML schemas seen as regular tree grammars, and the navigational features of XPath[1].

The novelty of this tool is that it is the first to be capable of proving exact properties over such a large class of schemas and XPath features. Furthermore, the tool is fairly efficient. The cost ranges from several milliseconds for comparison of XPath queries without tree constraints, to several seconds for queries under very large, heavily recursive, tree constraints, such as the XHTML DTD. In addition, the analyzer generates XML counter-examples which allow observing and reproducing the program defects independently from the analyzer.

[1]The supported XPath fragment is given in appendix.

II. System Architecture

The tool is based on our theoretical results presented in [4] about a tree logic adapted for XML. The system architecture is illustrated on Figure 1. The tool consists of a combination of several software components:

- a parser for reading the text problem description. The syntax combines queries expressed with the standard XPath syntax, references to XML Schemas (expressed using DTD, XML Schema or Relax NG [5]). The problem is then formulated with logical connectives;
- compilers for translating schemas and queries into their logical representations [4];
- an optimized solver for checking satisfiability of logical formulas in time $2^{O(n)}$ where n is the formula size [4];
- and a counter-example XML tree generator.

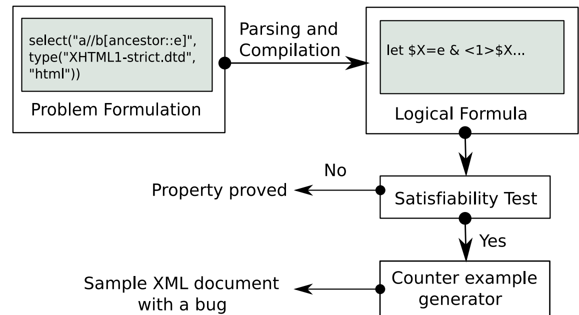

Fig. 1. System Architecture.

III. Demonstration Overview

The demonstration scenario consists of illustrating how to detect defects and prove properties on programs that manipulate both schemas and queries. The system has been

implemented as a Java/JSP web application and interaction with the system is offered through a web user interface in a web browser. The tool is available online from:

```
http://wam.inrialpes.fr/xml
```

Our demonstration aims to showcase the functionality of the analyser across a variety of use cases. First, we will analyze the relationship (forward and backward compatibility) between XHTML basic 1.0 and XHTML basic 1.1 schemas. Then, we will demonstrate how to check for XPath expressions emptiness under type constraints, as well as path containment and equivalence. Finally, we will explore the analysis of XPath expressions contained in a MathML XSLT transformation subject to an evolution of the input schema from MathML 1.0 to 2.0. The later is introduced progressively in order to allow the demo visitors to first notice that the transformation yields invalid results when fed with MathML 2.0 inputs. Then, we extract path expressions from the transformation and we proceed to their analysis. The analysis consists of testing and refining predicates to better understand the impact of a type evolution on a query.

The user can either enter an analysis problem using a simple predicate language through area (1) of Figure 1 or select from pre-loaded analysis tasks offered in area (4) of Figure 1. The level of details displayed by the solver can be adjusted in area (2) of Figure 1 and allows to inspect logical translations and statistics on problem size and the different operation costs. The results of the analysis are displayed in area (3) of Figure 1 together with XML counter-examples.

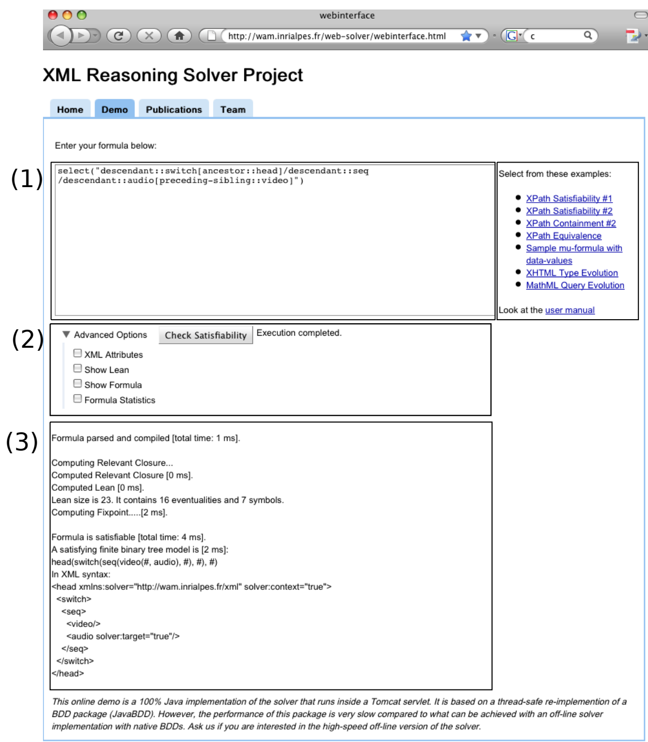

Fig. 2. Screenshot of the Solver Interface.

Users can formulate a wide range of problems with a user-friendly syntax based on predicates. Predicates facilitate the formulation of decision problems. They are logical macros allowing tool usage while focusing (only) on the XML-side properties, and keeping the underlying logical formulation transparent for the user. We show that the tool works fairly well even with complex queries and fairly large real-world schemas such as XHTML, MathML, SMIL. The tool is a web-based application which allows one to formulate an analysis problem using the predicate language, test the system with pre-defined set of sample tests, set the level of details for the analysis phase and produce analysis results. Our demonstration system is pre-loaded with several instances of real-world XML schemas together with their various dialects or profiles.

Evolution of XHTML Basic

The first test consists in analyzing the relationship (forward and backward compatibility) between XHTML basic 1.0 and XHTML basic 1.1 schemas. In particular, backward compatibility can be checked by the following command:

```
backward_incompatible("xhtml-basic10.dtd",
                      "xhtml-basic11.dtd", "html")
```

The test immediately yields a counter example as the new schema contains new element names. The counter example (shown below) contains a `style` element occurring as a child of `head`, which is not permitted in XHTML basic 1.0:

```
<html>
  <head>
    <title/>
    <style type="_otherV"/>
  </head>
  <body/>
</html>
```

The next step consists in focusing on the relationship between both schemas excluding these new elements. This can be formulated by the following command:

```
backward_incompatible("xhtml-basic10.dtd",
                      "xhtml-basic11.dtd", "html")
& exclude(added_element(
      type("xhtml-basic10.dtd","html"),
      type("xhtml-basic11.dtd", "html")))
```

The result of the test shows a counter example document that proves that XHTML basic 1.1 is not backward compatible with XHTML basic 1.0 even if new elements are not considered. In particular, the content model of the `label` element cannot have an `a` element in XHTML basic 1.0 while it can in XHTML basic 1.1. The counter example produced by the solver is shown below:

```
<html>
  <head>
    <object>
      <label>
        <a>
          <img/>
        </a>
        <img/>
      </label>
      <param/>
    </object>
    <meta/>
    <title/>
    <base/>
```

```
  </head>
  <body/>
</html>

XTML basic 1.0 validity error: element "a" is not
declared in "label" list of possible children
```

Notice that we observed similar forward and backward compatibility issues with several other W3C normative schemas (in particular for the different versions of SMIL and SVG). Such backward incompatibilities suggest that applications cannot simply ignore new elements from newer schemas, as the combination of older elements may evolve significantly from one version to another.

Emptiness test for an XPath expression

The most basic decision problem for a query language is the emptiness test of an expression: whether or not a query always yields an empty result. This test is important for error-detection and optimizing implementations of languages in which XPath expressions are used. For instance, if one can decide at compile time that a query result is empty then subsequent bound computations can be ignored (in other terms, dead code can be eliminated).

Empty queries often come from the use of an XPath expression in a setting where structural constraints are enforced by a schema. The combination of navigational information of the query and structural constraints imposed by the schema may rapidly yield contradictions. Such contradictions can be detected by checking a logical formula for satisfiability. For example, we may be interested in checking emptiness of an XPath expression over the set of documents which are valid against the DTD of the SMIL language. The following example shows the text file description of such a test:

```
select("descendant::switch[ancestor::head]/
  descendant::seq/descendant::audio
    [preceding-sibling::video]",
      type("SMIL/SMIL10.dtd", "smil"))
```

The first argument for the predicate `select()` is the XPath expression. The second argument describes the structural constraint. The predicate `type()` takes two parameters: a path to the DTD file (here the DTD is assumed to be located in a subdirectory called "SMIL"), and the name of the element to be considered as root symbol. Running the tool with this example formula yields the following:

```
Reading tree grammar 'SMIL/SMIL10.dtd'...
CFT: 30 variables, 30 rules, 19 element names.
BTT: 22 variables, 18 rules, 19 element names.
Converted tree grammar into BTT [153 ms].
Translated BTT into Tree Logic [34 ms].

Formula parsed and compiled [total time: 313 ms].

Computing Relevant Closure
Computed Relevant Closure [25 ms].
Computed Lean [1 ms].
Lean size is 52 with 32 eventualities and 20 symbols.
Computing Fixpoint......[55 ms].
Formula is satisfiable [total time: 300 ms].
A satisfying finite binary tree model was found [50 ms].
In XML syntax:
<smil xmlns:solver="http://wam.inrialpes.fr/xml"
      solver:context="true">
  <head>
    <switch>
      <seq>
        <video/>
        <audio solver:target="true"/>
      </seq>
      <layout>
        <root-layout/>
      </layout>
    </switch>
    <meta/>
  </head>
</smil>
```

The tool proceeds as follows. First, the input predicate `select()`, the XPath expression and the DTD are parsed and compiled into the logic. The referred DTD is converted into an intermediate representation on binary trees (called "BTT") before being compiled into the logic.

The logical translation of the problem whose satisfiability is going to be tested is built into memory. The tool then computes the Fisher-Ladner closure and the Lean of the formula: the set of all basic subformulas that notably defines the search space that is going to be explored by the solver. The solver attempts to build a satisfying tree in a bottom-up way, in the manner of a fixpoint computation that iteratively updates a set of tree nodes. This computation is performed in at most $2^{O(n)}$ steps with respect to size n of the Lean (see [4] for details). In this example, the formula is satisfiable: the XPath expression is not-empty in the presence of this DTD. A sample XML document satisfying the problem formulation is printed. Notice that the tool automatically annotates a pair of nodes related by the query: the query selects the node marked with `solver:target` when evaluated from a node marked with the attribute `solver:context`.

XPath Containment and Equivalence

Another essential problem for a query language is the containment problem: whether or not the result of one query is always included into the result of another one. Containment for XPath expressions is for instance needed for checking integrity constraints in XML databases, for the control-flow analysis of XSLT, for the static type-checking of XPath queries, and for checking XML access control policies.
Suppose for instance that we want to check containment between the following XPath expressions:

```
e1 = descendant::d[parent::b]/following-sibling::a

e2 = ancestor-or-self::*/descendant-or-self::b/
  a[preceding-sibling::d]
```

Since containment corresponds to logical implication, we actually want to check whether the implication of the two corresponding logical formulas is valid. Since we use a satisfiability solver, we verify this validity by checking for the unsatisfiability of the negated implication, formulated as follows:

```
~(select("e1",#)  => select("e2",#))
```

Queries must be compared from the same evaluation context, which can be any set of nodes, but the same for both expressions. The evaluation context is denoted by "#". Running the tool with this formula returns the following:

```
Formula parsed and compiled [total time: 131 ms].

Computing Relevant Closure
Computed Relevant Closure [2 ms].
Computed Lean [0 ms].
Lean size is 27 with 22 eventualities and 5 symbols.
Computing Fixpoint.....[8 ms].
Formula is unsatisfiable [22 ms].
```

The tested formula is unsatisfiable (in other terms: the implication is valid), so one can conclude that the first expression is contained in the second expression, for any XML document.

A related decision problem is the equivalence problem: whether or not two queries always return the same result. It is important for proving soundness of query rewriting rules used for query optimization, for instance. Equivalence corresponds to two containment tests. Note that the previous XPath expressions are not equivalent. This can be checked with the tool, that generates the following counter-example tree:

```
<b xmlns:solver="http://wam.inrialpes.fr/xml">
  <d/>
  <a solver:context="true" solver:target="true"/>
</b>
```

MathML Content to Presentation Analysis

In this test, we focus on the analysis of the queries contained in the XSLT transformation [6] from MathML [7] Content to Presentation, and evaluate the impact of the schema change from MathML 1.0 to MathML 2.0 on these queries. Most of the queries contained in the transformation represent only a few patterns very similar up to element names. The following test checks whether a frequently used XPath pattern may return new nodes when evaluated over MathML 2.0 documents compared to MathML 1.0:

```
new_region("//apply[*[1][self::eq]]","mathml.dtd",
  "mathml2.dtd","math")
```

The predicate new_region("*query*", $\mathcal{T}, \mathcal{T}'$) is satisfied iff the query *query* selects elements whose names already occurred in $\mathcal{T}$, but such that these nodes now occur in a new context in $\mathcal{T}'$. In this setting, the path from the root of the document to a node selected by the XPath expression *query* contains a node whose type is defined in $\mathcal{T}'$ but not in $\mathcal{T}$ (see [8] for more details).

The result of the test shows a counter example document that proves that the query selects new nodes in MathML 2.0 compared to MathML 1.0. In particular, the query selects `apply` elements whose ancestors can be `declare` elements, as indicated on the document produced by the tool:

```
<math xmlns:solver="http://wam.inrialpes.fr/xml"
            solver:context="true">
  <declare>
    <apply solver:target="true">
      <eq/>
    </apply>
    <condition/>
  </declare>
</math>
```

To evaluate the effect of this change, the counter example is filled with content and passed as an input parameter to the transformation. This shows immediately a bug in the transformation as the resulting document is not a MathML 2.0 presentation document. Based on this analysis, we know that the XSLT template associated with the match pattern `//apply[*[1][self::eq]]` must be updated to cope with MathML evolution from version 1.0 to version 2.0.

IV. Conclusion

We illustrated how to use the tool with XML schemas and queries on realistic examples. The tool can be very useful for standard schema writers, transformation writers, and maintainers in order to assist them for detecting bugs and enforcing some level of quality. In the longer term, we plan to equip XML programming languages with static type-checking and mechanically verified optimization features.

References

[1] E. Sedlar, “Managing structure in bits & pieces: the killer use case for XML,” in *SIGMOD ’05*. ACM, 2005, pp. 818–821.

[2] M. Benedikt, W. Fan, and F. Geerts, “XPath satisfiability in the presence of DTDs,” in *PODS ’05: Proceedings of the twenty-fourth ACM Symposium on Principles of Database Systems*. New York, NY, USA: ACM Press, 2005, pp. 25–36.

[3] P. Genevès, “Logics for XML,” Ph.D. dissertation, Institut National Polytechnique de Grenoble, December 2006, http://www.pierresoft.com/pierre.geneves/phd.htm.

[4] P. Genevès, N. Layaïda, and A. Schmitt, “Efficient static analysis of XML paths and types,” in *PLDI ’07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM Press, 2007, pp. 342–351.

[5] J. Clark and M. Murata, “RELAX NG specification, OASIS committee specification,” December 2001, http://relaxng.org/spec-20011203.html.

[6] E. Pietriga, “MathML content2presentation transformation,” May 2005, http://www.lri.fr/~pietriga/mathmlc2p/mathmlc2p.html.

[7] D. Carlisle, P. Ion, R. Miner, and N. Poppelier, “Mathematical markup language (MathML) version 2.0, W3C recommendation,” October 2003, http://www.w3.org/TR/MathML2/.

[8] P. Genevès, N. Layaïda, and V. Quint, “Identifying query incompatibilities with evolving XML schemas,” in *ICFP ’09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2009, pp. 221–230.

Appendix

The syntax of supported XPath queries is given below (see [4] for a detailed translation of XPath expressions into logical formulas):

$$
\begin{aligned}
\text{XPath} &::= \text{PathExpr} \mid /\text{PathExpr} \\
\text{PathExpr} &::= \text{PathExpr}/\text{PathExpr} \mid \text{PathExpr[Qualifier]} \mid \text{Step} \\
&\quad \mid \text{PathExpr union PathExpr} \mid \text{PathExpr intersect PathExpr} \\
&\quad \mid \text{PathExpr except PathExpr} \\
\text{Qualifier} &::= \text{PathExpr} \mid \text{not Qualifier} \mid \text{Qualifier and Qualifier} \\
&\quad \mid \text{Qualifier or Qualifier} \mid \text{PathExpr}/@\text{NameTest} \\
\text{Step} &::= \text{Axis::NameTest} \\
\text{Axis} &::= \text{self} \mid \text{child} \mid \text{parent} \mid \text{descendant} \mid \text{ancestor} \mid \text{following} \\
&\quad \mid \text{preceding} \mid \text{following-sibling} \mid \text{preceding-sibling} \\
&\quad \mid \text{descendant-or-self} \mid \text{ancestor-or-self} \\
\text{NameTest} &::= \text{QName} \mid *
\end{aligned}
$$