INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

XML Reasoning Solver User Manual

Pierre Genevès — Nabil Layaïda — Vojtěch Knyttl

N° 6726

August 22, 2011

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6726--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

XML Reasoning Solver User Manual

Pierre Genevès* , Nabil Layaïda , Vojtěch Knyttl

Thème SYM — Systèmes symboliques
Équipes-Projets Wam

Rapport de recherche n° 6726 — August 22, 2011 — 21 pages

Abstract: This manual provides documentation for using the logical solver introduced in [Genevès, 2006; Genevès *et al.*, 2007; Genevès *et al.*, 2008].

Key-words: Static Analysis, Logic, Satisfiability Solver, XML, Schema, XPath, Queries

* CNRS

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Analyseur Statique pour XML et XPath: Manuel Utilisateur

Résumé : Ce manuel documente l'utilisation du solveur logique décrit dans[Genevès, 2006; Genevès *et al.*, 2007; Genevès *et al.*, 2008].

Mots-clés : Analyse Statique, Logique, Solveur, Satisfaisabilité, XML, Schema, Requêtes, XPath

1 Introduction

This document describes the logical solver introduced in [Genevès, 2006; Genevès *et al.*, 2007] and provides informal documentation for using its implementation.

The solver allows automated verification of properties that are expressed as logical formulas over trees. A logical formula may for instance express structural constraints or navigation properties (like e.g. path existence and node selection) in finite trees.

A decision procedure for a logic traditionally defines a partition of the set of logical formulas: formulas which are *satisfiable* (there is a tree which satisfies the formula) and remaining formulas which are *unsatisfiable* (no tree satisfies the given formula). Alternatively (and equivalently), formulas can be divided into *valid* formulas (formulas which are satisfied by all trees) and *invalid* formulas (formulas that are not satisfied by at least one tree). The solver is a satisfiability-testing solver: it allows checking satisfiability (or unsatisfiability) of a given logical formula. Note that validity of a formula φ can be checked by testing $\neg\varphi$ for unsatisfiability.

The solver can be used for reasoning over finite ordered trees whatever these trees do actually represent. In particular, the logic and the solver are specifically adapted for formulating and solving problems over XML tree structures [Bray *et al.*, 2004]. The logic can express navigational properties like those expressed with the XPath standard language [Clark and DeRose, 1999] for navigating and selecting sets of nodes from XML trees. Additionally, the logic is expressive enough to encode any regular tree language property (it subsumes finite tree automata). It can encode constraints definable with common XML tree type definition languages (such as DTD [Bray *et al.*, 2004], XML Schema [Fallside and Walmsley, 2004], and Relax NG [Clark and Murata, 2001]). The logic provides high-level constructs specifically designed for reasoning directly with such XML concepts: the user can directly write an expression using XPath notation in the logic, or even refer to an XML type in the logic. These characteristics make the system especially useful for solving problems like those encountered in the static analysis of XML code, static verification of XML access control policies, XML data security checking, XML query optimization, and the construction of static type-checkers, and optimizing compilers for a wide variety of tree-manipulating programs and XML processors.

Outline This user manual is organized as follows: Section 2 describes the basics for using the solver without requiring any logical knowledge; Section 3 gives some insights on the logic, especially on the simple yet general data tree model used by the logic (Section 3.1) and on the syntax of logical formulas (Section 3.2) including high-level constructs for embedding XPath expressions and XML tree types directly in the logic. Finally, Section 4 provides an overview of the background theory underlying the logic and its solver, with related references.

2 Getting Started

2.1 Accessing the Solver

Online version. A web interface to the logical solver is available from:

http://wam.inrialpes.fr/websolver/

Offline version. The logical solver is shipped as a compressed file which, once extracted, provides binaries along with all required libraries. The "`solver.jar`" executable file takes a filename as a parameter[1]. The filename refers to a text file containing the logical formula to solve. For example, provided a recent[2] Java runtime engine is installed, the following command line:

```
java -jar solver.jar formula.txt
```

runs the solver on the logical formula contained in "`formula.txt`". The full syntax of logical formulas is given in Section 3.2. The following examples introduce the logical formulation of some simple yet fundamental XML problems, and how the solver output should be interpreted.

2.2 XML Applications

Example 1: 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 is self contradictory and always yields an empty result. This test is important for error-detection and optimization of host languages implementations, *i.e.* implementations that process 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. For checking emptiness of the XPath expression `a/b[following-sibling::c/parent::d]`, the contents of the "`example1.txt`" file simply consists of the following line:

example1.txt

```
select("a/b[following-sibling::c/parent::d]")
```

Running the solver with " `example1.txt`" as parameter yields the following trace:

Output for example1.txt

```
Reading example1.txt

Satisfiability Tested Formula:
(mu X5.(((b & (mu X2.(<-1>(a & (mu X1.(<-1>T | <-2>X1))) | <-2>X2)))
& (mu X4.(<2>((mu X3.(<-1>d | <-2>X3)) & c) | <2>X4)))|(<1>X5|<2>X5)))

Computing Relevant Closure
Computed Relevant Closure [1 ms].
Computed Lean [1 ms].
Lean size is 20. It contains 14 eventualities and 6 symbols.
Computing Fixpoint.....[4 ms].
Formula is unsatisfiable [14 ms].
```

The input XPath expression is first parsed and compiled into the logic. The corresponding logical translation whose satisfiability is going to be tested is printed. The

[1] Running the command "`java -jar solver.jar`" prints the list of required and optional arguments.

[2] A Java virtual machine version 1.5.0 (or further) and a Java compiler compliance level version 5.0 (or further).

solver 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 (see [Genevès *et al.*, 2007] for details). 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.

In this example, no satisfying tree is found: the formula is unsatisfiable (in other terms, no matter on which XML document this XPath expression is evaluated, it will always yield an empty result). Intuitively, that is because this XPath expression contains a contradiction: according to the query, the same node is required to be named both "`a`" and "`d`", which is not allowed for an XML tree.

Empty queries often come from the use of an XPath expression in a constrained setting. The combination of navigational information of the query and structural constraints imposed by a DTD (or XML Schema) may rapidly yield contradictions. Such contradictions can also be detected by checking a logical formula for satisfiability.

Example 2: checking XPath emptiness in the presence of tree constraints. Suppose we want to check emptiness of the XPath expression

```
descendant::switch[ancestor::head]/descendant::seq/
   descendant::audio[preceding-sibling::video]
```

over the set of documents defined by the DTD of the SMIL language [Hoschka, 1998]. The following formula is used:

example2.txt

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

The first argument for the predicate `type()` is a path to the DTD file (here the DTD is assumed to be located in a subdirectory called "sampleDTDs"), and the second argument is the name of the element to be considered as top-level start symbol. Running the solver with this "`example2.txt`" file as parameter yields the following trace:

Output for example2.txt

```
Reading example2.txt
Converted tree grammar into BTT [169 ms].
Translated BTT into Tree Logic [60 ms].

Satisfiability Tested Formula:
(mu X22.(((audio & (mu X20.(<-1>((seq & (mu X19.(<-1>(((switch
 & (mu X17.(<-1>(
(let_mu
  X1=(((meta & ~(<1>T)) & ~(<2>T)) | ((meta & ~(<1>T)) & <2>X1)),
  ...
  X16=((smil & (~(<1>T) | <1>X15)) & ~(<2>T))
in
  X16) | X17) | <-2>X17))) & (mu X18.(<-1>(head | X18) | <-2>X18)))
  | X19) | <-2>X19)))  | X20) | <-2>X20))) &
```

```
  (mu X21.(<-2>video | <-2>X21)))  | (<1>X22 | <2>X22)))

Computing Relevant Closure
Computed Relevant Closure [39 ms].
Computed Lean [1 ms].
Lean size is 50. It contains 31 eventualities and 19 symbols.
Computing Fixpoint......[37 ms].
Formula is satisfiable [99 ms].
A satisfying finite binary tree model was found [52 ms]:
smil(head(switch(seq(video(#, audio), layout), meta), #), #)
In XML syntax:
<smil xmlns:solver="http://wam.inrialpes.fr/xml" solver:context="true">
  <head>
    <switch>
      <seq>
        <video/>
        <audio solver:target="true"/>
      </seq>
      <layout/>
    </switch>
    <meta/>
  </head>
</smil>
```

The referred external DTD (tree grammar) is first parsed, converted into an internal representation on binary trees (called "BTT" and that corresponds to the mapping described in 3.1), and then compiled into the logic. The XPath expression is also parsed and compiled into the logic so that the global formula can be composed. In that case, the formula is satisfiable (the XPath expression is non-empty in the presence of this DTD). The solver outputs a sample tree for which the formulas is satisfied. This sample tree is enriched with specific attributes: the "solver:target" attribute marks a sample node selected by the XPath expression when evaluated from a node marked with "solver:context".

Example 3: checking containment and equivalence between XPath expressions. One of the most 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 the static type-checking of XPath queries, for the control-flow analysis of XSLT [Clark, 1999], for checking integrity constraints in XML databases, for XML data security...
Suppose for instance that we want to check containment between the following XPath expressions:

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

and:

```
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 formulas is valid. Since we use a satisfiability-testing algorithm, we verify this validity by checking for the unsatisfiability of the negated implication, as follows:

example3.txt

```
~( select("descendant::d[parent::b]/following-sibling::a",#)
  => select("ancestor-or-self::*/descendant-or-self::b
                                    /a[preceding-sibling::d]",#))
```

Note that XPath expressions must be compared from the same evaluation context, which can be any set of nodes, but should be the same set of nodes for both expressions. This is denoted by "#". Running the solver with this "`example3.txt`" file results in the following trace:

Output for example3.txt

```
Reading example3.txt

Satisfiability Tested Formula:
(mu X26.(((a & (mu X15.((<-2>T & (~(<-2>T) | <-2>((d & (mu X13.((<-1>T
& (~(<-1>T) | <-1>(_context | X13))) | (<-2>T & (~(<-2>T) | <-2>X13)))))
& (mu X14.((<-1>T & (~(<-1>T) | <-1>b)) | (<-2>T & (~(<-2>T)
 | <-2>X14)))))) | (<-2>T & (~(<-2>T) | <-2>X15))))) & ((~(a) |
 (mu X22.((~(<-1>T) | <-1>(~(b) | ((~(_context) & (~(<1>T) |
 <1>(mu X18.((~(_context) & (~(<1>T) | <1>X18)) & (~(<2>T) |
 <2>X18))))) & (mu X20.((~(<-1>T) | <-1>((~(_context) & (~(<1>T) |
 <1>(mu X19.((~(_context) & (~(<1>T) | <1>X19)) & (~(<2>T) |
 <2>X19))))) & X20)) & (~(<-2>T) | <-2>X20)))))) &(~(<-2>T) |
 <-2>X22)))) | (mu X25.((~(<-2>T) | <-2>~(d)) & (~(<-2>T) |
 <-2>X25))))) | (<1>X26 | <2>X26)))

Computing Relevant Closure
Computed Relevant Closure [4 ms].
Computed Lean [1 ms].
Lean size is 29. It contains 23 eventualities and 6 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 XPath expression is contained in the second XPath expression.

A related decision problem is the equivalence problem: whether or not two queries always return the same result. It is important for reformulation and optimization of an expression, which aims at enforcing operational properties while preserving semantic equivalence. Equivalence is reducible to containment (bi-implication) and is noted `<=>` in the logic. Note that the previous XPath expressions are not equivalent. The reader may check this by using the solver, that will generate the following counter-example tree:

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

3 Logical Insights

3.1 Data Model for the Logic

An XML document is considered as a finite tree of unbounded depth and arity, with two kinds of nodes respectively named elements and attributes. In such a tree, an element may have any number of children elements, and may carry zero, one or more attributes. Attributes are leaves. Elements are ordered whereas attributes are not, as illustrated on Figure 1. The logic allows reasoning on such trees. Notice that from an XML perspective, data values are ignored.

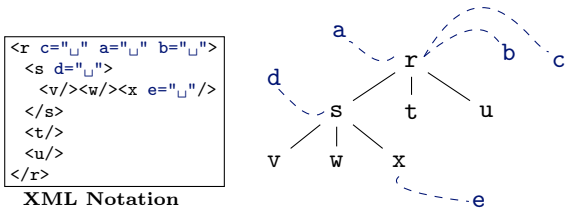


Figure 1: Sample XML Tree with Attributes.

Unranked and Binary Trees There are bijective encodings between unranked trees (trees of unbounded arity) and binary trees. Owing to these encodings binary trees may be used instead of unranked trees without loss of generality. The logic operates on binary trees. The logic relies on the "first-child & next-sibling" encoding of unranked trees. In this encoding, the first child of a node is preserved in the binary tree representation, whereas siblings of this node are appended as right successors in the binary representation. The intuition of this encoding is illustrated on Figure 2 for a sample tree. Trees can be seen as terms or function calls. More formally, a binary tree t can be defined by the recursive syntax $t ::= \sigma(t, t') \mid \epsilon$ where σ is a node label and ϵ denotes the empty tree. Similarly unranked trees can be defined as $t ::= \sigma(h)$ where h is a hedge (a sequence of unranked trees) defined as $h ::= \sigma(h), h' \mid \epsilon$. The function f that translates unranked trees into binary trees is then defined by $f(\sigma(h), h') = \sigma(f(h), f(h'))$ and $f(\epsilon) = \epsilon$. The reverse mapping used for reconstructing unranked trees from binary trees can be expressed as: $f^{-1}(\sigma(t, t')) = \sigma(f^{-1}(t)), f^{-1}(t')$ and $f^{-1}(\epsilon) = \epsilon$.

In the remaining part of this manual, the binary representation of a tree is implicitly considered, unless stated otherwise. From an XML point of view, notice that only the nested structure of XML elements (which are ordered) is encoded into binary form like this. XML attributes (which are unordered) are left unchanged by this encoding. For instance, Figure 3 presents how the sample tree of Figure 1 is mapped.

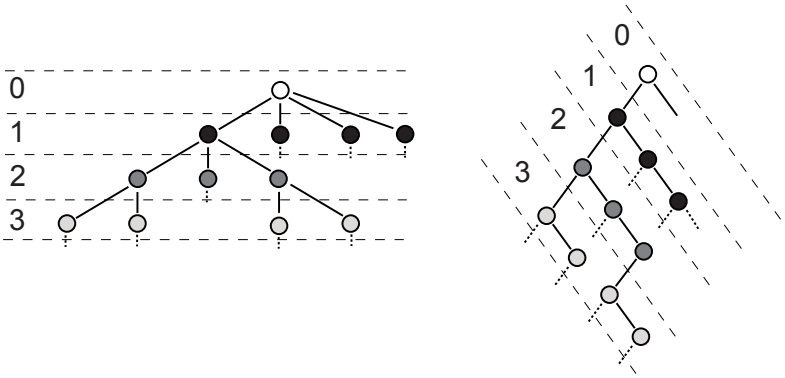


Figure 2: Binary Encoding Principle.

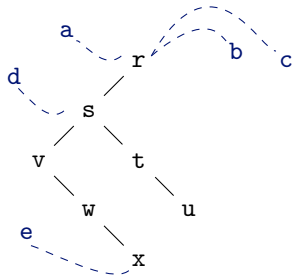


Figure 3: Binary Encoding of Tree of Figure 1.

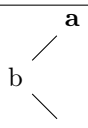
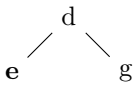
Sample Formula	Satisfying Binary Tree	XML syntax
`a & <1>b`		`<a><b/></a>`
`a & <2>b`		`<a/><b/>`
`a & <1>(b & <2>c)`		`<a><b/><c/></a>`
`e & <-1>(d & <2>g)`		`<d><e/></d><g/>`
`f & <-2>(g & ~<2>T)`	none	none

Table 1: Sample Formulas using Modalities.

3.2 Syntax of Logical Formulas

Modal Formulas for Navigating in Trees The logic uses two *programs* for navigating in binary trees: the program `1` allows to navigate from a node down to its first successor and the program `2` for navigating from a node down to its second successor. The logic also features *converse programs* `-1` and `-2` for navigating upward in binary trees, respectively from the first and second successors to the parent node. Some basic logical formulas together with corresponding satisfying binary trees are shown on Table 1. When using XPath expressions, like e.g. `select("a[b]")`, the XPath expression is automatically compiled into a logical formula over the binary tree representation (see Section 3.2).

The set of logical formulas is defined by the syntax given on Figure 4, where the meta-syntax $\langle X \rangle^{\oplus}$ means one or more occurences of X separated by commas. Models of a formula are finite binary trees for which the formula is satisfied at some node. The semantics of logical formulas is formally defined in [Genevès, 2006; Genevès *et al.*, 2007]. Table 1 gives basic formulas that use modalities for navigating in binary trees and node names.

Recursive Formulas The logic allows expressing recursion in trees through the use of a fixpoint operator. For example the recursive formula:

```
let $X = b | <2>$X in $X
```

means that either the current node is named `b` or there is a sibling of the current node which is named `b`. For this purpose, the variable `$X` is bound to the subformula `b | <2>$X` which contains an occurence of `$X` (therefore defining the recursion). The scope of this binding is the subformula that follows the "`in`" symbol of the formula, that is `$X`. The entire formula can thus be seen as a compact recursive notation for a infinitely nested formula of the form:

φ	::=		formula
		`T`	true
	|	`F`	false
	|	l	element name
	|	p	atomic proposition
	|	`#`	start context
	|	φ `|` φ	disjunction
	|	φ `&` φ	conjunction
	|	φ `=>` φ	implication
	|	φ `<=>` φ	equivalence
	|	`(`φ`)`	parenthesized formula
	|	`~`φ	negation
	|	`<`p`>`φ	existential modality
	|	`<`l`>T`	attribute named l
	|	`$X`	variable
	|	`let` $\langle$`$X` $= \varphi\rangle^{\oplus}$ `in` φ	binder for recursion
	|	*predicate*	predicate (See Section 3.3)
p	::=		program inside modalities
		`1`	first child
	|	`2`	next sibling
	|	`-1`	parent
	|	`-2`	previous sibling

Figure 4: Syntax of Logical Formulas.

```
b | <2>(b | <2>(b | <2>(...)))
```

Recursion allows expressing global properties. For instance, the recursive formula:

```
~ let $X = a | <1>$X | <2>$X in $X
```

expresses the absence of nodes named `a` in the whole subtree of the current node (including the current node). Furthermore, the fixpoint operator makes possible to bind several variables at a time, which is specifically useful for expressing mutual recursion. For example, the mutually recursive formula:

```
let $X = (a & <2>$Y) | <1>$X | <2>$X, $Y = b | <2>$Y in $X
```

asserts that there is a node somewhere in the subtree such that this node is named `a` and it has at least one sibling which is named `b`. Binding several variables at a time provides a very expressive yet succinct notation for expressing mutually recursive structural patterns (that may occur in DTDs for instance).

The combination of modalities and recursion makes the logic one of the most expressive (yet decidable) logic known. For instance, regular tree grammars can be expressed with the logic using recursion and (forward) modalities. The combination of converse programs and recursion allows expressing properties about ancestors of a node for instance. The possibility of nesting recursive formulas allow XPath expressions to be translated into the logic. Note that use of variables in bound formulas must be guarded by a program $<\alpha>$; expressions of the following form are not allowed:

```
let $X = $Y in $X
```

Cycle-Freeness Restriction There is a restriction on the use of recursive formulas. Only formulas that are *cycle-free* are allowed. Intuitively a formula is cycle-free if it does not contain both a program and its converse inside the *same* recursion. For instance, the formula

```
let $X = a | <-1>$X | <1>$X in $X
```

is not cycle-free since `1` and `-1` occur in front of the same variable bound by the same binder. A formula is cycle-free if one cannot find both a program and its converse by starting from a variable and going up in the formula tree to the binder of this variable. For instance, the following formula is cycle-free:

```
let $X = a & (let $X = b | <1>$X in $X) | <-1>$X in $X
```

since variable binders are properly nested and a program and its converse never appear in front of the same variable bound by the same binder.

Translations of XPath expressions and XML tree types into the logic *always* generate cycle-free formulas, whatever the translated XPath or XML type is. The cycle-freeness restriction only matters when one directly writes recursive logical formulas. From a theoretical perspective the cycle-freeness restriction comes from the fact that converse programs may interact with recursion in a subtle manner such that the finite model property is lost, so the cycle-freeness restriction ensures that the negation of every formula can also be expressed in the logic, or in other terms, that the logic is closed under negation and all other boolean operations (a detailed discussion on this topic can be found in [Genevès *et al.*, 2007]).

spec	::=		
		φ	formula (see Fig. 4)
	|	$def; \varphi$	
def	::=		
		$predicate\text{-}name(\langle l \rangle^{\oplus}) = \varphi'$	custom definition
	|	$def; def$	list of definitions

Figure 5: Global Syntax for Specifying Problems.

Supported XPath Expressions The logic provides high-level constructions for facilitating the formulation of problems involving XPath expressions. The construct `select("`e`",`φ`)` where e is an XPath expression provides a way of embedding XPath expression directly into the logic (e is automatically compiled into a logical formula, see [Genevès *et al.*, 2007] for details on the compilation technique). The second parameter φ denotes the context from which the XPath is applied; it can be any formula. The other construct `select("e")` is simply a shorthand for `select("`e`",#)`, where `#` is the initial context node mark. The syntax of supported XPath expressions is given on Figure 6. We observed that, in practice, many XPath expressions contain syntactic sugars that can also fit into this fragment. Figure 7 presents how our XPath parser rewrites some commonly found XPath patterns into the fragment of Figure 6, where the notation $(a::nt)^k$ stands for the composition of k successive path steps of the same form: $\underbrace{a::nt/.../a::nt}_{k \text{ steps}}$.

Supported XML Types The logic is expressive enough to allow for the encoding of any regular tree grammar. The logical construction `type("`*filename*`",`*start*`)` provides a convenient way of referring to tree grammars written in usual notations like DTD, XML Schema, or Relax NG. The referred tree type is automatically parsed and compiled into the logic, starting from the given *start* symbol (which can be the root symbol or any other symbol defined by the tree type).

3.3 Predicates

We build on the aforementioned query and schema compilers, and define additional predicates that facilitate the formulation of decision problems at a higher level of abstraction. Specifically, these predicates are introduced as logical macros with the goal of allowing system usage while focusing (only) on the XML-side properties, and keeping underlying logical issues transparent for the user. Ultimately, we regard the set of basic logical formulas (such as modalities and recursive binders) as an assembly language, to which predicates are translated. Default predicates are defined as follows:

3.3.1 `nempty|non_empty(`*query*`,` φ`)`

Test for non-emptiness of xpath query posed on type formula φ.

query	::=		
		$/path$	absolute path
	|	$path$	relative path
	|	$query \mid query$	union
	|	$query \cap query$	intersection
path	::=		
		$path/path$	path composition
	|	$path[qualifier]$	qualified path
	|	$a::nt$	step
qualifier	::=		
		$qualifier$ and $qualifier$	conjunction
	|	$qualifier$ or $qualifier$	disjunction
	|	not($qualifier$)	negation
	|	$path$	path
	|	$path/@nt$	attribute path
	|	$@nt$	attribute step
nt	::=		node test
		σ	node label
	|	$*$	any node label
a	::=		tree navigation axis
		self | child | parent	
	|	descendant | ancestor	
	|	descendant-or-self	
	|	ancestor-or-self	
	|	following-sibling	
	|	preceding-sibling	
	|	following | preceding	

Figure 6: XPath Expressions.

$$
\begin{aligned}
nt[\text{position}() = 1] &\rightsquigarrow nt[\text{not}(\text{preceding-sibling}::nt)] \\
nt[\text{position}() = \text{last}()] &\rightsquigarrow nt[\text{not}(\text{following-sibling}::nt)] \\
nt[\text{position}() = \underbrace{k}_{k>1}] &\rightsquigarrow nt[(\text{preceding-sibling}::nt)^{k-1}] \\
\text{count}(path) = 0 &\rightsquigarrow \text{not}(path) \\
\text{count}(path) > 0 &\rightsquigarrow path \\
\text{count}(nt) > \underbrace{k}_{k>0} &\rightsquigarrow nt/(\text{following-sibling}::nt)^{k}
\end{aligned}
$$

$$
\begin{aligned}
&\text{preceding-sibling}::*[\text{position}() = \text{last}() \text{ and } qualifier] \\
&\qquad \rightsquigarrow \text{preceding-sibling}::*[\text{not}(\text{preceding-sibling}::*) \text{ and } qualifier]
\end{aligned}
$$

Figure 7: Syntactic Sugars and their Rewritings.

Example.

```
non_empty("//body", type("xhtml11.dtd","html")
```

3.3.2 added_elements(φ,ψ)

Nodes which appears in type formula ψ, but not in φ.

Example.

```
added_elements(b|c|d, a|b|c|d|e)
```

3.3.3 added_elements($schema_1$, $schema_2$, $root$)

Nodes which appears in $schema_2$, but not in $schema_1$ (with common $root$).

Example.

```
added_elements("mathml.dtd", "mathml2.dtd", "math")
```

3.3.4 new_elements($query$, $schema_1$, $schema_2$, $root$)

Nodes which are selected by query posed on $schema_2$, but not on $schema_1$.

Example.

```
new_elements("//*", "mathml.dtd", "mathml2.dtd", "math")
```

3.3.5 new_elements($query_1$, $query_2$, $schema_1$, $schema_2$, $root$)

Nodes which are selected by $query_2$ posed on $schema_2$, but not by $query_1$ on $schema_1$.

Example.

```
new_elements("//*", "//*", "mathml.dtd",
                              "mathml2.dtd", "math")
```

3.3.6 new_regions($query$, $schema_1$, $schema_2$, $root$)

Nodes selected by the query on $schema_2$ which appear in new context with respect to $schema_1$.

Example.

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

Example.

```
new_region("//sin[preceding-sibling::*[position()=last()
                    and (self::compose or self::inverse)]]",
     "mathml.dtd","mathml2.dtd","math")
  & exclude(added_elements(
        type("mathml.dtd", "math"),
        type("mathml2.dtd", "math")))
  & exclude(declare)
```

3.3.7 new_parents($query_1$, $query_2$, $schema$, $root$)

Predicate select parents of target nodes for $query_2$ on given $schema$, which were not parent nodes for target nodes of $query_1$.

3.3.8 new_parents($query_1$, $query_2$, $schema_1$, $schema_2$, $root$)

Predicate select parents of target nodes for $query_2$ on given $schema_2$, which were not parent nodes for target nodes of $query_1$ posed on $schema_1$. This predicate can either be used for query evolution on new schemas, or even simply for testing structural properties with respect to one single schema:

Example.

```
new_parents("//img","//div", "xhtml-basic10.dtd", "html")
```

Predicate is equivalent to:

$$\texttt{child(select(}query_2\texttt{, type(}schema_2\texttt{, }root\texttt{)) \&}$$
$$\texttt{(target\_elements(child(select(}query_1\texttt{, type(}schema_1\texttt{, }root\texttt{)))}$$

3.3.9 new_siblings($query$, $schema_1$, $schema_2$, $root$)

Potentional siblings of nodes selected by query on $schema_2$ which were not siblings of nodes selected on $schema_1$.

3.3.10 new_siblings($query_1$, $query_2$, $schema_1$, $schema_2$, $root$)

Potentional siblings of nodes selected by $query_2$ on $schema_2$ which were not siblings of nodes selected by $query_1$ on $schema_1$.

3.3.11 new_contents($query$, $schema_1$, $schema_2$, $root$)

Satisfiable if the content model of the selected nodes by query have changed.

Example.

```
new_contents("//apply[*[1][self::apply]/inverse]","mathml.dtd",
                        "mathml2.dtd","math")
        & exclude(added_elements(type("mathml.dtd", "math"),
        type("mathml2.dtd", "math")))
```

3.3.12 `new_children(`$query_1$`, `$query_2$`, `$schema$`, `$root$`)`

Predicate selects nodes, which are children of $query_2$ posed on $schema$, that are not children of $query_1$.

3.3.13 `new_children(`$query_1$`, `$query_2$`, `$schema_1$`, `$schema_2$`, `$root$`)`

Predicate selects nodes, which are children of $query_2$ posed on $schema_2$, that are not children of $query_1$ posed on $schema_1$. This predicate can either be used for query evolution on new schemas, or even simply for testing structural properties with respect to one single schema:

Example.

```
new_children("//tbody", "//table", "xhtml1-strict.dtd", "html")
```

Predicate is equivalent to:

```
        parent(select(query2, type(schema2, root)) &
(target_elements(parent(select(query1, type(schema1, root)))
```

3.3.14 `backward_incompatible(`$schema_1$`, `$schema_2$`, `$root$`)`

Predicate is satisfiable if $schema_2$ is bacward-incompatible with $schema_1$.

Example.

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

User might consider following conjuction to consider only elements, which appeared in both schemas:

Example.

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

3.3.15 `forward_incompatible(`$schema_1$`, `$schema_2$`, `$root$`)`

Predicate is satisfiable if $schema_2$ is forward-incompatible with $schema_1$.

3.3.16 `exclude(`φ`)`

Predicate which guarantees, that φ will never be satisfied in the whole considered tree. Predicate is equivalent to:

```
(ancestor-or-self(descendant-or-self(φ))
```

Example.

```
exclude(added_elements(
          type("xhtml-basic10.dtd","html"),
          type("xhtml-basic11.dtd", "html")))
```

3.3.17 `ncontain|non_containment(`$query_1$`,` $query_2$`)`

Predicate is satisfiable if the xpath *expression* $query_2$ describes a path, which is not contained in *expression* $query_2$. Predicate is equivalent to:

`select(`$query_1$`) & select(`$query_2$`)`

Example.

```
non_containment("descendant::d[parent::b]/following-sibling::a",
         "ancestor-or-self::*/descendant-or-self::b/a[preceding-sibling::d]")
```

3.3.18 `ncontain|non_containment(`$query_1$`,` $query_2$`,` *schema*`,` *root*`)`

Predicate is satisfiable, if there exists an element, which is selected by $query_2$, but not by $query_1$. Predicate is equivalent to:

`select(`$query_1$`, type(`*schema*`,` *root*`)) & select(`$query_2$`, type(`*schema*`,` *root*`))`

3.3.19 `nequiv|non_equivalence(`$query_1$`,` $query_2$`)`

Predicate is satisfiable, if there exists an element selected by one of the queries, but not with the other one. Equivalent to:

`non_containment(`$query_1$`,` $query_2$`) & non_containment(`$query_2$`,` $query_1$`)`

3.3.20 `ncover|non_coverage(`*query*`,` *node_name*`,` *schema*`,` *root*`)`

Predicate is satisfiable, if the query does not select all the nodes *node_name* in the schema. Predicate is equivalent to:

`non_containment("//"+`*node_name*`,` *query*`,` *schema*`,` *root*`)`

Example.

```
non_coverage("//tbody/tr", "tr", "xhtml-basic10.dtd", "html")
```

3.3.21 `nsubtype|non_subtype(`φ`,` ψ`)`

Semantic subtyping test predicate. More information can be found in [Gesbert *et al.*, 2011].

3.3.22 `isd()`

Predicate defined for Parametric polymorphism and semantic subtyping [Gesbert *et al.*, 2011].

3.3.23 `target_elements(`φ`)`

Return union of all possible target node names. Note that this predicate will run the solver as many times as there are target nodes.

Example.

```
target_elements(<2>(<-2>a)|<1>(<-1>b))
```

Following use of target_elements will show all possible nodes, which can have <tr> as a child.

Example.

```
target_elements(ancestor(type("xhtml1-strict.dtd", "html")) & child(tr))
```

3.3.24 `satisfiable(`φ`)`

Predicate returns TRUE in case of satisfiability of φ. Otherwise it returns FALSE. Note that this predicate requires additional run of the solver.

Example.

```
satisfiable(a & b) => satisfiable(a | b)
```

3.3.25 `reg_exp(`*expression*`)`

Interprets given regular *expression*. Square bracket notation "[a-z]" is not supported, please use union as "(a|b|...|z)" instead. Complement is defined only for character union as "^(a|b|...)". Intersection of two regular *expression*s:

Example.

```
reg_exp("a+b{2,4}(a|b|c|d)*") & reg_exp("abb((ab)|(ad)){3}")
```

Non equivalence of two regular *expression*s:

Example.

```
~(reg_exp("(ab)+a") <=> reg_exp("a(ba)+"))
```

Example.

```
~(reg_exp("(a|b|c)+") <=> reg_exp("(a|b|c){1,3}"))
```

3.4 Custom Predicates

Following the spirit of predicates presented in the previous section, users may also define their own custom predicates. The full syntax of XML logical specifications to be used with the system is defined on Figure 5, where the meta-syntax $\langle X \rangle^{\oplus}$ means one or more occurrence of X separated by commas. A global problem specification can be any formula (as defined on Figure 4), or a list of custom predicate definitions separated by semicolons and followed by a formula. A custom predicate may have parameters that are instanciated with actual formulas when the custom predicate is called. A formula bound to a custom predicate may include calls to other predicates,

but not to the currently defined predicate (recursive definitions must be made through the let binder shown on Figure 4).

Example.
```
sibling($x) = preceding_sibling($x) | following_sibling($x);
sibling(table)
```

4 Overview of the Background Theory

The logic and its solver are formally described in [Genevès, 2006; Genevès *et al.*, 2007]. The logic is a modal logic of trees, more specifically an alternation-free μ-calculus with converse for finite trees. The logic is equipped with forward and backward modalities, which are notably useful for capturing all XPath (including reverse) axes. The logic is also equipped with a fixed-point operator for expressing recursion in finite trees. A n-ary fixed-point operator is also provided so that mutual recursion occurring in XML types can be succintly expressed in the logic. The logic is also able to express any propositional property, for instance about nodes labels (XML element and attribute names). Last but not least, the logic is closed under negation [Genevès, 2006; Genevès *et al.*, 2007], that is, the negation of any logical formula can be expressed in the logic too (this property is essential for checking XPath containment which corresponds to logical implication). All these features together: propositions, forward and backward modalities, recursion (fixed-points operators), and boolean connectives yield a logic of very high expressive power. Actually, this logic is one of the most expressive yet decidable known logic. It can express properties of regular tree languages. Specifically, it is as expressive as tree automata (which notably provide the foundation for the Relax NG language in the XML world) and monadic second-order logic of finite trees (often referred as WS2S or "MSO" in the literature) [Thatcher and Wright, 1968; Doner, 1970]. However, the logical solver is considerably (orders of magnitude) faster than solvers for monadic second-order logic, like e.g., the MONA solver [Klarlund *et al.*, 2001] (the MONA solver nevertheless remains useful when one wants to write logical formulas using MSO syntax). Technically, the truth status of a logical formula (satisfiable or unsatisfiable) is automatically determined in exponential time, and more specifically in time $2^{O(n)}$ where n is proportional to (and smaller than) the size of the logical formula [Genevès, 2006; Genevès *et al.*, 2007]. In comparison, the complexity of monadic second-order logic is much higher: it was proved in the late 1960s that the best decision procedure for monadic second order logic is at least hyper-exponential in the size of the formula [Thatcher and Wright, 1968; Doner, 1970] that is, not bounded by any stack of exponentials. The tree logic described in this document currently offers the best balance known between expressivity and complexity for decidability. The acute reader may notice that the complexity of the logic is optimal since it subsumes tree automata and less expressive logics such as CTL [Clarke and Emerson, 1981], for instance.

XPath expressions and regular tree types can be linearly translated into the logic. This observation allows to generalize the complexity of the algorithm for solving the logic to a wide range of problems in the XML world.

The decision procedure for the logic is based on an inverse tableau method that searches for a satisfying tree. The algorithm has been proved sound and complete in [Genevès, 2006; Genevès *et al.*, 2007]. The solver is implemented using symbolic techniques like binary decision diagrams (BDDs) [Bryant, 1986]. It also uses numerous optimization techniques such as on-the-fly formula normalization and simplification, conjunctive partitioning, early quantification.

Finally, another benefit of this method (illustrated in Section 2.2) is that the solver can be used to generate an example (or counter-example) XML tree for a given property, which allows for instance to reproduce a program's bug in the developer environment, independently from the logical solver.

References

[Bray *et al.*, 2004] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.0 (third edition), W3C recommendation, February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/. 3

[Bryant, 1986] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. 20

[Clark and DeRose, 1999] James Clark and Steve DeRose. XML path language (XPath) version 1.0, W3C recommendation, November 1999. http://www.w3.org/TR/ 1999/REC-xpath-19991116. 3

[Clark and Murata, 2001] James Clark and Makoto Murata. RELAX NG specification, OASIS committee specification, December 2001. http://relaxng.org/spec-20011203.html. 3

[Clark, 1999] James Clark. XSL transformations (XSLT) version 1.0, W3C recommendation, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116. 6

[Clarke and Emerson, 1981] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, volume 131 of *LNCS*, pages 52–71, London, UK, 1981. Springer-Verlag. 20

[Doner, 1970] John Doner. Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 4:406–451, 1970. 20

[Fallside and Walmsley, 2004] David C. Fallside and Priscilla Walmsley. XML Schema part 0: Primer second edition, W3C recommendation, October 2004. http://www.w3.org/TR/xmlschema-0/. 3

[Genevès *et al.*, 2007] Pierre Genevès, Nabil Layaïda, and Alan 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*, pages 342–351, New York, NY, USA, 2007. ACM Press. 1, 2, 3, 5, 10, 12, 13, 20

[Genevès *et al.*, 2008] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of XML paths and types. Research Report 6590, INRIA, July 2008. 1, 2

[Genevès, 2006] Pierre Genevès. *Logics for XML*. PhD thesis, Institut National Polytechnique de Grenoble, December 2006. http://www.pierresoft.com/pierre.geneves/phd.htm. 1, 2, 3, 10, 20

[Gesbert *et al.*, 2011] Nils Gesbert, Pierre Genevès, and Nabil Layaïda. Parametric polymorphism and semantic subtyping: the logical connection. In *ICFP '11: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, page (to appear), 2011. 18

[Hoschka, 1998] Philipp Hoschka. Synchronized multimedia integration language (SMIL) 1.0 specification, W3C recommendation, June 1998. http://www.w3.org/TR/REC-smil/. 5

[Klarlund *et al.*, 2001] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA 1.4, January 2001. http://www.brics.dk/mona/. 20

[Thatcher and Wright, 1968] James W. Thatcher and Jesse B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968. 20

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399