Abstract

The internship took place in the WAM team of Inria Grenoble. As written on their website : "Project WAM aims at making it easier to develop and use rich multimedia contents and applications on the web. Particular attention is paid to documents and applications that tightly integrate different types of media objects, be they discrete (text, images, equations) or continuous (video, audio, animations). Multimedia resources are distributed, linked together, and use platform-neutral formats, which make them usable by anyone through different kinds of terminals and networks."

One of the particular topic that WAM is studying, and the background of the internship, is XML processing. More precisely, it consisted in translating formulas over trees into a tree automata defining the same set.

This report is divided in two main parts. The first two sections settle the background and cover several potential applications of the work done. The other sections present the main results produced in a article-like form. While not present in this report, implementing the translation, and implementing it in an efficient way, was one of the goal of the internship. The code was written in Java and is an extension to the code from the XML Reasoning Solver Project.

1 Motivation

2 Background

2.1 XML

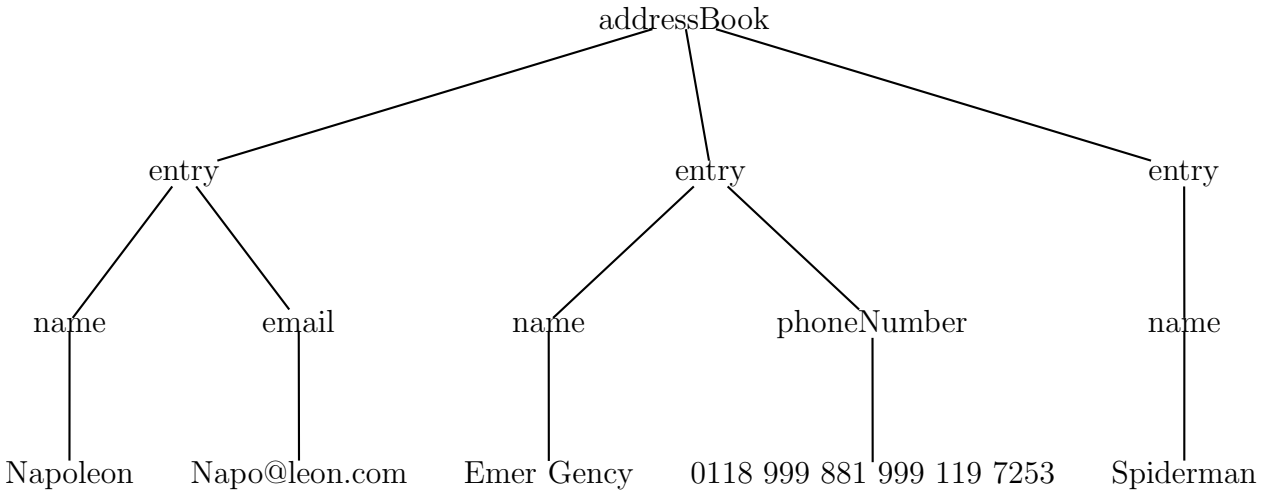
XML[1] is a language designed to manipulate organized data in order to have generic tools to manipulate them. In XML, data is stored in the form of an unranked tree. For example, one can represent in XML a small addressBook like this :

```
<addressBook>
     <entry>
         <name>Napoleon</name>
         <email>Napo@leon.com</email>
     </entry>
     <entry>
```

[1]XML is not described here but a precise description of XML can be found on *http://www.w3.org/TR/xml/*

```
        <name>Emer Gency</name>
        <phoneNumber>0118 999 881 999 119 7253</phoneNumber>
    </entry>
    <entry>
        <name>Spiderman</name>
    </entry>
</addressBook>
```

And the corresponding tree would be :



In order to deal with unranked trees, we will use the left child-right sibling notation. This notation is equivalent to unranked trees as long as we don't forget if a node is a right child or a left child.

2.2 Schemas and validations

An XML file is said well-formed if it complies with the XML specifications. But, because XML covers a lot of different application needs, we often need to check that an XML file complies with some additional constraints, depending on the need we have. For a correct address book, we can state that the root should always be a `<addressBook>`, that it has any number of children but all are `<entry>`. That an `<entry>` has always one `<name>` as a child and beside this `<name>`, it has only `<phoneNumber>` and `<email>`. There are many ways -DTD, XML schemas, RelaxNG, Schematron- to describe these additional constraints.

2.3 XPath

One of the frequent tool used with XML is XPath. XPath is a tool selecting nodes satisfying certain properties in a document. For instance, with the address book example we might want to know what are the entries in the address book that are just with a name. XPath allows us to do so by the query "self::entry[child::name[not (following-sibling::* or preceding-sibling::* or child::*)]]". Literally: the nodes named `<entry>` with a child `<name>` that has no preceding, following or child nodes. XPath can also be use to accept or reject some documents; if they have a node satisfying some property we need; like refusing address books with an `<entry>` that has only a `<name>`.

XPath is also used as a tool by other languages. For example, XSLT and XQuery use it to transform documents, Schematron uses it to describe constraints.

3 Applications

3.1 Typing

Typing a XML document is describing the document with a schema. Programs manipulating data often perform dynamic type checking : they receive a file and they check it over a schema before to use it. This is done in order to avoid run-time errors due to a false assertion over the form of the tree. Some data manipulation languages like CDuce or XDuce have already successfully introduced some static type checking. In XDuce, for instance, each function has annotation of its type and the type checker checks that each operation is valid with regard to its input and output types. There are type checkers for these languages and it is a variation arround automata containment. Nonetheless, their functional programming approach does not cover all the needs and each function has to be annotated.

In the general case of XPath queries with a schema, verifying or inferring an output type for a program is an undecidable task. In order to avoid undecidability we consider only a fragment -we take out arithmetic and strings manipulation- of XPath and we add to this fragment some operator making it FO-complete. This extension is called CXPath[2] introduced in [4].

[2]There is no need to describe it (nor to describe XPath) here precisely; there exists a linear translation of the CXPath in the syntax of the formulas manipulated, described in [2].

3.2 XPaths with a type

The logic solver of the WAM team is capable of translating a large number of formulas over trees into their logic : CXPath, DTD, RelaxNG, XML Schema. So, they could already answer all questions in the form of a boolean combination of these formulas. And, more than only solving the formula, they already could produce a tree for satisfiable formulas. But, because the solver solves in exponential time (exactly in $2^{O(n)}$ where n is the size of the formula), and because the translation of schemas is also linear, the time for the type checking of a schema s and an CXPath x takes a time $2^{O(|s|+|x|)}$. If we can produce an automaton $\mathcal{A}$ for x then the time would only be $O(|d| \times |A|)$ because schemas can be translated linearly in finite tree automata. This would be a real advance because schemas are usually big whereas XPaths are usually short.

Because CXPath can be translated linearly in WS2S cycle-free[1] formulas. And because the translation of WS2S cycle-free into automata is known, we also solve the problem of the translation of CXPath into automata. It has been known for long[6] that WS2S definable sets are equivalent with regular languages and there exists a tool, MONA[1], that uses this translation. It is also shown that the translation is generally non elementary. Here, we consider a different syntax where exists are replaced by a least fix point; and we where only cycle-free formulas are allowed. Here, as we will prove it, the translation is single-exponential.

Translating CXPath into automata translation was already attempted in [3] and improved in [1] but this translation has several majors drawbacks :

- the translation uses unranked tree automata : those automata are complex to handle.
- the translation was only for a fragment of CXPath where we translate all WS2S cycle-free formulas (strictly more expressive than CXPath).
- the translation was considering a lot of different cases making the proof more difficult to read and to verify. Some cases where even missing.

[1]These formulas and the term cycle-free are defined in the second part of the report.
1

4 Definition

4.1 Syntax

The syntax of the formulas over an alphabet of labels $\mathcal{A}$ and a disjointed alphabet of attributes $\mathcal{B}$ used is:

- $\varphi = \varphi_1 \vee \varphi_2$
- $\varphi = \varphi_1 \wedge \varphi_2$
- $\varphi = \neg\varphi'$
- $\varphi = \langle a \rangle \varphi'$ where $a \in \{1, 2, \bar{1}, \bar{2}\}$. We have $\bar{\bar{a}} = a$.
- $\varphi = \mu\overline{X_i = \varphi'_i}\ in\ \psi$
- X where X is a variable.
- $\varphi = \top$
- $\varphi = \sigma, \sigma \in \mathcal{A} \cup \mathcal{B}$

  The last two kind of formulas are called context-formulas.

4.2 Semantic

The trees we consider are over the alphabet $\mathcal{A}$, each node n has a label $\mathcal{L}(n) \in \mathcal{A}$, it also has a set $\mathcal{S}(n) \subset \mathcal{B}$ of attributes. We use the classic semantic:

- $[\![\varphi_1 \wedge \varphi_2]\!]_V = [\![\varphi_1]\!]_V \cap [\![\varphi_2]\!]_V$.
- $[\![\varphi_1 \vee \varphi_2]\!]_V = [\![\varphi_1]\!]_V \cup [\![\varphi_2]\!]_V$.
- $[\![\neg\varphi]\!]_V = \mathcal{F} \setminus [\![\varphi]\!]_V$.
- $[\![\langle a \rangle \varphi]\!]_V = \{\mathcal{T} \langle \bar{a} \rangle \mid \mathcal{T} \in [\![\varphi]\!]_V \wedge \mathcal{T} \langle \bar{a} \rangle \text{ defined } \}$
- $[\![\mu\overline{X_i = \varphi_i}\ in\ \psi]\!]_V = \text{let } T_i = \{ \bigcap_{T_i \subset \mathcal{F}} T_i \mid [\![\varphi_i]\!]_{V\{\overline{T_i/X_i}\}} \subset T_i\} \text{ in } [\![\psi]\!]_{V\{T_i/X_i\}}$
- If $\varphi = \sigma$ where $\sigma \in \mathcal{A}$, $[\![\varphi]\!]_V = \{\mathcal{T} \in \mathcal{F}, \mathcal{L}(\mathcal{T}) = \sigma\}$.
- If $\varphi = \sigma$ where $\sigma \in \mathcal{B}$, $[\![\varphi]\!]_V = \{\mathcal{T} \in \mathcal{F}, \sigma \in \mathcal{S}(\mathcal{T})\}$.

4.3 Expanding fix points

Definition 1. *A fix point can be expanded by* $exp(\mu\overline{X_i = \varphi_i}\ in\ \psi) = \psi_{\mu\overline{X_i=\varphi_i}\ in\ \varphi_i/X_i}$.
The unfolding of a formula is the smallest set verifying:

- $unf(\varphi_1 \wedge \varphi_2) = \{u_1 \wedge u_2 \mid u_1 \in unf(\varphi_1), u_2 \in unf(\varphi_2)\}$
- $unf(\varphi_1 \vee \varphi_2) = \{u_1 \vee u_2 \mid u_1 \in unf(\varphi_1), u_2 \in unf(\varphi_2)\}$
- $unf(\neg\varphi) = \{\neg u \mid u \in unf(\varphi)\}$
- $unf(\langle a\rangle\, \varphi) = \{\langle a\rangle\, u \mid u \in unf(\varphi)\}$
- $unf(\sigma) = \{\sigma\}$ *for* σ *context-formula*
- $unf(\mu\overline{X_i = \varphi_i}\ in\ \psi) = unf(exp(\mu\overline{X_i = \varphi_i}\ in\ \psi)) \cup \{\mu\overline{X_i = \varphi_i}\ in\ \psi\}$

Remark 1. *It has been proved in [2] that exp does not change the semantic of a formula.*

Remark 2. *As we always unfold fix points we encounter, we don't have to consider variables as a case for induction on formulas.*

4.4 Modality-free variables

Definition 2. *We defined the set* $\mathcal{U}(\varphi)$ *of modality-free variables in* φ *as the smallest set verifying:*

- $\mathcal{U}(\varphi_1 \wedge \varphi_2) = \mathcal{U}(\varphi_1) \cup \mathcal{U}(\varphi_2)$
- $\mathcal{U}(\varphi_1 \vee \varphi_2) = \mathcal{U}(\varphi_1) \cup \mathcal{U}(\varphi_2)$
- $\mathcal{U}(\neg\varphi) = \mathcal{U}(\varphi)$
- $\mathcal{U}(\langle a\rangle\, \varphi) = \emptyset$
- $\mathcal{U}(\mu\overline{X_i = \varphi_i}\ in\ \psi) = \{X_i\} \sqcup \mathcal{U}(\psi)$
- $\mathcal{U}(\varphi) = \emptyset$ *if* φ *is a context-formula*

In the formulas we consider, for any formula $\mu\overline{X_i = \varphi_i}\ in\ \psi$, *we have every occurrence of* X_i *in* ψ *guarded by a* $\langle a\rangle$. *That's why we wrote* $\{X_i\} \sqcup \mathcal{U}(\psi)$.

Such a set is well defined because the size of the formulas considered always decrease in induction calls.

4.5 Modality paths

Definition 3. *A modality path is a sequence of programs from* $\{1, 2, \bar{1}, \bar{2}\}$. *We can easily extend navigational operations to modality path. If all* $(\mathcal{T}\langle a_1\rangle \dots \langle a_n\rangle)_{1\leq i\leq n}$ *are defined then* $\mathcal{T}\langle a_1\rangle \dots \langle a_n\rangle$ *is defined and is equal to* $(\mathcal{T}\langle a_1\rangle \dots \langle a_{n-1}\rangle)\langle a_n\rangle$ *(left-associative).*

A modality path $\langle a_1\rangle \dots \langle a_n\rangle$ *is called valid on* $\mathcal{T}$ *if* $\mathcal{T}\langle a_1\rangle \dots \langle a_n\rangle$ *is defined.*

Definition 4. *A cycle in a modality path is a sub-sequence of the form* $\langle a\rangle\langle\bar{a}\rangle$.

Remark 3. *It has been proved in [2] that for cycle-free formulas the largest and smallest fix points collapse. Because we only consider cycle-free formulas, there is no need of a largest fix point.*

Definition 5. *We define the set* $\mathcal{P}(\varphi)$ *of modality paths of a formula* φ *as the smallest set verifying:*

- $\mathcal{P}(\varphi_1 \wedge \varphi_2) = \mathcal{P}(\varphi_1) \cup \mathcal{P}(\varphi_2)$
- $\mathcal{P}(\varphi_1 \vee \varphi_2) = \mathcal{P}(\varphi_1) \cup \mathcal{P}(\varphi_2)$
- $\mathcal{P}(\neg\varphi) = \mathcal{P}(\varphi)$
- $\mathcal{P}(\langle a\rangle \varphi) = \{S\langle a\rangle \,|S \in \mathcal{P}(\varphi)\} \cup \{\langle a\rangle\}$
- $\mathcal{P}(\mu\overline{X_i = \varphi_i}\ in\ \psi) = \mathcal{P}(exp(\mu\overline{X_i = \varphi_i}\ in\ \psi))$
- $\mathcal{P}(\sigma) = \{\epsilon\}$ *for* σ *a context-formula;* ϵ *represents the path of size 0.*

Definition 6. *A formula* φ *is called cycle-free if it exists* n *such that* $\forall u \in unf(\varphi), \forall p \in \mathcal{P}(u), p$ *contains less than* n *cycles.*

Lemma 1. *For every focused tree* $\mathcal{T}$ *we cannot have a valid cycle-free modality path of length greater than* $size(\mathcal{T})$.

Proof. Let $a_1, \dots, a_n$ be a valid cycle-free modality path, with $n \geq size(\mathcal{T})$. Because $(\mathcal{T}\langle a_1\rangle \dots \langle a_i\rangle)_i$ is valid, we can think of it as a way in the tree of $\mathcal{T}$. A way longer than the number of nodes and therefore a way that contains a cycle (in the sense of a cycle in a graph). A cycle in a tree always contains a turn back. So, $a_1, \dots, a_n$ necessarily contains a cycle (ie a pattern $\langle a\rangle\langle\bar{a}\rangle$). □

Corollary 1. *For every focused tree* $\mathcal{T}$ *and every formula* φ, *it exists a* $n_{\varphi,\mathcal{T}}$ *such that* $\forall u \in unf(\varphi), \forall p \in \mathcal{P}(u), |p| > n_{\varphi,\mathcal{T}} \Rightarrow p$ *is not valid on* $\mathcal{T}$.

Proof. Let φ be a cycle-free formula and $\mathcal{T}$ a focused tree. We already know there is a n such that any valid path of any unfolding of φ contains less than n cycles.

Let $\langle a_1 \rangle \dots \langle a_m \rangle$ be a valid modality path.

Let i, j be with $1 \leq i \leq j \leq n$ and $\langle a_i \rangle \dots \langle a_j \rangle$ cycle-free. Because $\langle a_1 \rangle \dots \langle a_n \rangle$ is valid we can state $\mathcal{T}' = \mathcal{T} \langle a_1 \rangle \dots \langle a_{i-1} \rangle$ and then 1 proves that $j - i \leq size(\mathcal{T})$. Any cycle-free subsequence is smaller in size than $size(\mathcal{T})$.

Now, we can cut the path at each cycle. For example $\langle 1 \rangle \langle 2 \rangle \langle \bar{2} \rangle \langle 1 \rangle \langle 1 \rangle \langle \bar{1} \rangle \langle \bar{1} \rangle \langle \bar{1} \rangle \langle \bar{1} \rangle$ become $\langle 1 \rangle \langle 2 \rangle \mid \langle \bar{2} \rangle \langle 1 \rangle \langle 1 \rangle \mid \langle \bar{1} \rangle \langle \bar{1} \rangle \langle \bar{1} \rangle \langle \bar{1} \rangle$. There are, at most, $n + 1$ parts because there was at most n cycles. Each part is smaller than $size(\mathcal{T})$ so m was, at most, of size $size(\mathcal{T}) \times (n + 1)$. $n_{\varphi,\mathcal{T}} = size(\mathcal{T}) \times (n + 1)$ works. □

Remark 4. *The formulas we consider are all cycle-free.*

4.6 Focused trees

Definition 7. *The lean is the set* $\mathcal{A} \cup \mathcal{B} \cup \{\langle a \rangle \varphi | \langle a \rangle \varphi \in sub(\xi) \cup \{\top\}\}$. *cf [2].*

Definition 8. *We define the set* $\mathcal{S}$ *of tree over* $\mathcal{A}$ *as all finite binary trees whose labels are in* $\mathcal{A}$ *and attributes in* $\mathcal{B}$.

Definition 9. *A focused tree over* $\mathcal{A}$ *is a tree from* $\mathcal{S}$ *with the additional information of which node we are looking at. We write* $\mathcal{F}$ *for the set of all focused trees.*

For a focused tree $\mathcal{T}$ *we can define the following navigational operations:*

- *if* $\mathcal{T}$ *is centered in* n *and* n *has a father* p *and is a left child then* $\mathcal{T} \langle \bar{1} \rangle$ *is the tree of* $\mathcal{T}$ *but centered in* p.
- *if* $\mathcal{T}$ *is centered in* n *and* n *has a father* p *and is a right child then* $\mathcal{T} \langle \bar{2} \rangle$ *is the tree of* $\mathcal{T}$ *but centered in* p.
- *if* $\mathcal{T}$ *is centered in* n *and* n *has a right child* p *then* $\mathcal{T} \langle 2 \rangle$ *is the tree of* $\mathcal{T}$ *but centered in* p.
- *if* $\mathcal{T}$ *is centered in* n *and* n *has a left child* p *then* $\mathcal{T} \langle 1 \rangle$ *is the tree of* $\mathcal{T}$ *but centered in* p.

4.7 Types

Definition 10. *A formula is called a $\mathcal{L}ean$-formula if it can be rewritten as a formula depending on formulas in the $\mathcal{L}ean$ (in CNF form for instance?)*

Definition 11. *A set $t \subset \mathcal{L}ean$ is called a type if t does not contain both $\langle\bar{1}\rangle \top$ and $\langle\bar{2}\rangle \top$ and if it contains exactly one element from $\mathcal{A}$. A type t also needs to respect the following condition: $\langle a\rangle \varphi \in t \Rightarrow \langle a\rangle \top \in t$.*

Definition 12. *The constrained formula for a type t is $\Phi_c(t) = \bigwedge_{\varphi\in t} \varphi \wedge \bigwedge_{\varphi\in\mathcal{L}ean\setminus t} \neg\varphi$.*

Remark 5. *Given a type t and a $\mathcal{L}ean$-formula φ we have either $\Phi_c(t) \Rightarrow \varphi$ or $\Phi_c(t) \Rightarrow \neg\varphi$.*

Definition 13. *Two types x and y are compatible for $a \in \{1, 2, \bar{1}, \bar{2}\}$ if, for every $\langle a\rangle \varphi \in \mathcal{L}ean$, we have $\langle a\rangle \varphi \in x \Leftrightarrow (\Phi_c(y) \Rightarrow \varphi)$. For $a \in \{1, 2\}$, we have $\Delta_a(x, y)$ iff x is compatible with y for a and y with x for $\bar{a}$*

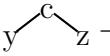
5 Automaton

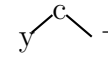
5.1 Automata over trees with attributes

Because trees with labels from $\mathcal{A}$ and attributes from $\mathcal{B}$, can be seen like trees with labels in $\mathcal{A} \times 2^{\mathcal{B}}$; we see automata for trees with attributes like automata over the alphabet $\mathcal{A} \times 2^{\mathcal{B}}$. It might also be useful to erase some attributes by transforming all rules labels from $a, x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{|\mathcal{B}|}$ to $a, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{|\mathcal{B}|}$ and see the automaton for an alphabet $\mathcal{A} \times 2^{\mathcal{B}\setminus\{x_i\}}$.

5.2 Automaton for a formula

We build the automaton $(Q, \mathcal{A}, \delta, \mathcal{S})$ for the formula ξ with:

- Q the set of types
- For any tuple of types x, y, z, any label $c \in \mathcal{A}$ and any set of attributes $\mathcal{O} \subset \mathcal{B}$ we have:
  - If $\Phi_c(x) \Rightarrow \langle 1\rangle \top \wedge \langle 2\rangle \top$ then δ contains 

$\rightarrow x$ iff $\Delta_1(x, y) \wedge \Delta_2(x, z) \wedge \mathcal{L}(x) = c \wedge \mathcal{O} = \mathcal{S}(x)$
  - If $\Phi_c(x) \Rightarrow \neg\langle 1\rangle \top \wedge \langle 2\rangle \top$ then δ contains 

$\rightarrow x$ iff $\Delta_1(x, y) \wedge \mathcal{L}(x) = c \wedge \mathcal{O} = \mathcal{S}(x)$

– If $\Phi_c(x) \Rightarrow \langle 1\rangle \top \wedge \neg \langle 2\rangle \top$ then δ contains $\rightarrow x$ iff $\Delta_2(x,z) \wedge \mathcal{L}(x) = c \wedge \mathcal{O} = \mathcal{S}(x)$

– If $\Phi_c(x) \Rightarrow \neg \langle 1\rangle \top \wedge \neg \langle 2\rangle \top$ then δ contains $\rightarrow x$ iff $\mathcal{L}(x) = c \wedge \mathcal{O} = \mathcal{S}(x)$

- $Q_f = \{q \in Q \ / \ \Phi_c(q) \Rightarrow \mu X = \xi \vee \langle 1\rangle X \vee \langle 2\rangle X \text{ in } X \wedge \neg \langle \bar{1}\rangle \top \wedge \neg \langle \bar{2}\rangle \top\}$.

It's easy to build an algorithm that, given a formula ξ, builds the above automaton in single exponential time.

6 Consequences

We can translate ... to NDFTA in single exponential time.

7 Proof

7.1 Existence of run

Lemma 2. *For any tree T from $[\![\xi]\!]$, it exists a labeling of the nodes of T with states of Q that is consistent with the transitions built for ξ.*

Proof. Let T be a tree. We introduce a labeling and then we show that it is consistent.

The labeling $t \rightarrow q_t$ is defined like this : for t a node and for $\varphi \in \mathcal{L}ean$, $\varphi \in q_t$ iff $t \in [\![\varphi]\!]$.

There are four cases to consider but they are redundant. So, we will not consider the cases of trees with one child.

In the case of a node d with no children. The focused tree $\mathcal{T}$ focused on d is in $[\![\Phi_c(q_d)]\!]$. So, d respects $\mathcal{S}(q_d)$ and $\mathcal{L}(q_d)$ and the automaton can jump to q_d.

In the case of a node d, where d has a left child f_1 and a right f_2. We will show that if the automaton can jump to q_{f_a} when it reads f_a for $a \in \{1,2\}$ then it can jump to q_d when it reads d. As before, d respects $\mathcal{S}(q_d)$ and $\mathcal{L}(q_d)$. We need also to show that $\Delta_a(q_d, q_{f_a})$ for $a \in \{1,2\}$. By definition:

$$\Delta_a(q_d, q_{f_a}) \ \Leftrightarrow \ \begin{cases} \langle a\rangle \varphi \in q_d & \Leftrightarrow \quad \Phi_c(q_{f_a}) \Rightarrow \varphi \quad \text{for } \langle a\rangle \varphi \in \mathcal{L}ean \\ \langle \bar{a}\rangle \varphi \in q_{f_a} & \Leftrightarrow \quad \Phi_c(q_d) \Rightarrow \varphi \quad \text{for } \langle \bar{a}\rangle \varphi \in \mathcal{L}ean \end{cases}$$

and for $\langle a \rangle \varphi \in \mathcal{L}ean$ as we already have $f_a \in [\![\Phi_c(q_{f_a})]\!]$ so

$$\langle a \rangle \varphi \in q_d \Leftrightarrow d \in [\![\langle a \rangle \varphi]\!] \Leftrightarrow f_a \in [\![\varphi]\!] \Leftrightarrow f_a \in [\![\varphi \wedge \Phi_c(q_{f_a})]\!]$$

$$\langle a \rangle \varphi \in q_d \Leftrightarrow [\![\varphi \wedge \Phi_c(q_{f_a})]\!] \neq \emptyset \Leftrightarrow \neg(\Phi_c(q_{f_a}) \Rightarrow \neg\varphi) \Leftrightarrow \Phi_c(q_{f_a}) \Rightarrow \varphi$$

At the same time for $\langle \bar{a} \rangle \varphi \in \mathcal{L}ean$ we already have $d \in [\![\Phi_c(q_d)]\!]$ so

$$\langle \bar{a} \rangle \varphi \in q_{f_a} \Leftrightarrow f_a \in [\![\langle \bar{a} \rangle \varphi]\!] \Leftrightarrow d \in [\![\varphi]\!] \Leftrightarrow d \in [\![\varphi \wedge \Phi_c(q_d)]\!]$$

$$\langle \bar{a} \rangle \varphi \in q_{f_a} \Leftrightarrow [\![\varphi \wedge \Phi_c(q_d)]\!] \neq \emptyset \Leftrightarrow \neg(\Phi_c(q_d) \Rightarrow \neg\varphi) \Leftrightarrow \Phi_c(q_d) \Rightarrow \varphi$$

As expected, we got:

$$\Delta_1(q_d, q_{f_1}) \wedge \Delta_2(q_d, q_{f_2}) = \top$$

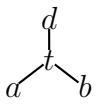
□

7.2 The verification function $\mathcal{V}$

Now, we consider φ, a $\mathcal{L}ean$-formula, and $t \to q_t$ a run.

As we want to prove that states define the correct truth assignment, we build a function $\mathcal{V}$ to check, truth assignments by passing through, at most, k modalities.

Definition 14. *We define $\mathcal{V}$ like this :*

- *If $k = 0$ then* $\mathcal{V}(\varphi,$  $, 0) = (\Phi_c(q_t) \Rightarrow \varphi)$

- *If $\varphi = \top$ then* $\mathcal{V}(\varphi, \bullet, \bullet) = \varphi = \top$
- *If $\varphi \in \mathcal{A} \cup \mathcal{B}$ formula then* $\mathcal{V}(\varphi, \mathcal{T}, \bullet) = \varphi \in q_t$
- $\mathcal{V}(\varphi_1 \vee \varphi_2, \mathcal{T}, k+1) = \mathcal{V}(\varphi_1, \mathcal{T}, k+1) \vee \mathcal{V}(\varphi_2, \mathcal{T}, k+1)$
- $\mathcal{V}(\varphi_1 \wedge \varphi_2, \mathcal{T}, k+1) = \mathcal{V}(\varphi_1, \mathcal{T}, k+1) \wedge \mathcal{V}(\varphi_2, \mathcal{T}, k+1)$
- $\mathcal{V}(\neg\varphi, \mathcal{T}, k+1) = \neg\mathcal{V}(\varphi, \mathcal{T}, k+1)$
- $\mathcal{V}(\langle a \rangle \varphi, \mathcal{T}, k+1) = \begin{cases} \mathcal{V}(\varphi, \mathcal{T}\langle a \rangle, k) & \text{if } \mathcal{T}\langle a \rangle \in q_t \\ \bot & \text{if } \neg\mathcal{T}\langle a \rangle \in q_t \end{cases}$
- $$\begin{array}{rcl} \mathcal{V}(\mu\overline{X_i = \varphi_i} \text{ in } \psi, \mathcal{T}, k+1) & = & \mathcal{V}(exp(\mu\overline{X_i = \varphi_i} \text{ in } \varphi_i), \mathcal{T}, k+1) \\ & = & \mathcal{V}(\varphi_{[X_i/\mu\overline{X_i=\varphi_i} \text{ in } \varphi_i]}, \mathcal{T}, k+1) \end{array}.$$

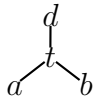
Remark 6. *$\mathcal{V}$ is defined by induction on k, next on the set of modality-free variables, next on the size of the formula.*

Because $\mathcal{V}$ is defined recursively we can make proofs by induction on it.

Lemma 3. *At φ and $\mathcal{T}$ constant, the function $k \to \mathcal{V}(\varphi, \mathcal{T}, k)$ is constant.*

Proof. We prove that $\mathcal{V}(\varphi, \mathcal{T}, k) = \mathcal{V}(\varphi, \mathcal{T}, 0)$ by induction on k, next on the set of modality-free variables and finally on the size of the formulas.

The base case $k = 0$ is immediate.

Consider k, φ, $\mathcal{T} =$ 

- if φ is a context formula then $\mathcal{V}$ does not depend on k.
- if $\varphi = \varphi_1 \wedge \varphi_2$, $\varphi = \varphi_1 \vee \varphi_2$ or $\varphi = \neg\varphi'$. Only the size of formulas changes (and decreases), the result holds by induction.
- if $\varphi = \langle a \rangle \psi$, we have

  $$\mathcal{V}(\langle a \rangle \psi, \mathcal{T}, k+1) = \mathcal{V}(\psi, \mathcal{T} \langle a \rangle, k) = \mathcal{V}(\psi, \mathcal{T} \langle a \rangle, 0) = \mathcal{V}(\langle a \rangle \psi, \mathcal{T}, 1)$$

  but by definition of Δ_a we have

  $$\mathcal{V}(\psi, \mathcal{T} \langle a \rangle, 0) \Leftrightarrow \langle a \rangle \psi \in q_t$$

  and by definition of $\mathcal{V}$ we have

  $$\langle a \rangle \psi \in q_t \Leftrightarrow \mathcal{V}(\langle a \rangle \psi, \mathcal{T}, 0)$$

  Finally we have

  $$\mathcal{V}(\langle a \rangle \psi, \mathcal{T}, k+1) = \mathcal{V}(\langle a \rangle \psi, \mathcal{T}, 0)$$

- if $\varphi = \mu\overline{X_i = \varphi_i}$ *in* ξ then X_i cannot appear with modality in ξ so the set of modality-free variables decrease and we have the result by induction.

□

Remark 7. *As k does not play a role in the definition of $\mathcal{V}$ we can refer to $\mathcal{V}(\varphi, \mathcal{T}, k)$ as $\mathcal{V}(\varphi, \mathcal{T})$.*

7.3 Equivalence between $\mathcal{V}$ and $[\![\varphi]\!]$

Lemma 4. *For every focused tree and every $\mathcal{L}$ean-formula φ we have $\mathcal{V}(\varphi, \mathcal{T}) \Leftrightarrow \mathcal{T} \in [\![\varphi]\!]$.*

Proof. Let k be such that $k > n(\varphi, \mathcal{T})$. That means it does not exist any valid modality paths p from $\mathcal{P}(u)$ where u is an unfolding of φ and p is of size k.

We now show that $\mathcal{V}(\varphi, \mathcal{T}) = \mathcal{V}(\varphi, \mathcal{T}, k) = (\mathcal{T} \in [\![\varphi]\!])$ for $k \geq n(\varphi, \mathcal{T})$ by induction on the order used to define $\mathcal{V}$.

- if φ is a context formula then clearly $\mathcal{V}(\varphi, \mathcal{T}, k) = \mathcal{T} \in [\![\varphi]\!]$.
- if $\varphi = \varphi_1 \wedge \varphi_2$, because $\mathcal{V}(\varphi, \mathcal{T}, k) = \mathcal{V}(\varphi_1, \mathcal{T}, k) \wedge \mathcal{V}(\varphi_2, \mathcal{T}, k)$ and $[\![\varphi_1 \wedge \varphi_2]\!] = [\![\varphi_1]\!] \cap [\![\varphi_2]\!]$ we have by induction $\mathcal{V}(\varphi, \mathcal{T}, k) = (\mathcal{T} \in [\![\varphi]\!])$. We have the induction property for k, because any path of any unfolding of φ_1 or φ_2 is a path for an unfolding of φ.
- $\varphi = \varphi_1 \vee \varphi_2$ or $\varphi = \neg\varphi'$ it is the same.
- if $\varphi = \langle a \rangle \psi$. If $\mathcal{T} \langle a \rangle$ then we have $\mathcal{V}(\varphi, \mathcal{T}, k+1) = \mathcal{V}(\psi, \mathcal{T} \langle a \rangle, k)$ and $\mathcal{V}(\psi, \langle a \rangle \mathcal{T}, k) = \mathcal{T} \langle a \rangle \langle \bar{a} \rangle \in [\![\psi]\!] = \mathcal{T} \langle a \rangle \langle \bar{a} \rangle \in [\![\varphi]\!] = \mathcal{T} \in [\![\varphi]\!]$. If $\neg \mathcal{T} \langle a \rangle$ then $\mathcal{V}(\varphi, \mathcal{T}, k+1) = \bot$ and $\mathcal{T} \notin [\![\langle a \rangle \varphi]\!] = \{\mathcal{T} \langle \bar{a} \rangle / \mathcal{T} \in [\![\varphi]\!]\}$ so we always have $\mathcal{V}(\varphi, \mathcal{T}, k+1) = \mathcal{T} \in [\![\varphi]\!]$. We have the induction property for k, because any path of any unfolding of ψ of size k is a path for an unfolding of φ of size $k+1$.
- if $\varphi = \mu\overline{X_i = \varphi_i}\ in\ \psi$ then $\mathcal{V}(\varphi, \mathcal{T}, k) = \mathcal{V}(exp(\mu\overline{X_i = \varphi_i}\ in\ \psi), \mathcal{T}, k) = \mathcal{T} \in [\![exp(\mu\overline{X_i = \varphi_i}\ in\ \psi)]\!] = \mathcal{T} \in [\![\varphi]\!]$. We have the induction property for k, because any path of any unfolding of $exp(\mu\overline{X_i = \varphi_i}\ in\ \psi)$ is a path for an unfolding of φ.

□

Lemma 5. *A tree is accepted by the automaton made for the formula ψ iff this tree respect ψ.*

Proof. Given a tree $\mathcal{T}$, it exists a run on $\mathcal{T}$, by definition of Q_f the run is accepted iff we have $\mathcal{V}(\mu X = \xi \vee \langle 1 \rangle X \vee \langle 2 \rangle X \text{ in } X \wedge \neg \langle \bar{1} \rangle \top \wedge \neg \langle \bar{2} \rangle \top, \mathcal{T}, 0)$ iff $\mathcal{T} \in [\![\mu X = \xi \vee \langle 1 \rangle X \vee \langle 2 \rangle X \text{ in } X \wedge \neg \langle \bar{1} \rangle \top \wedge \neg \langle \bar{2} \rangle \top]\!]$. □

8 Path formulas containment

Attributes from $\mathcal{B}$ are needed for translating XPath formulas with XML attributes. But, in the context of XPath, they can also be useful to describe where the context of the evaluation started and stating intersection, containment, union of path formulas like `self::a/descendant::b` $\cap$ `self::b/descendant::b` $= \emptyset$.

8.1 Query automata

The idea of using query automata to recognize the sets of solution for CXpath formulas was already in [1]. It is easy to adapt the above construction to build a query automaton. For instance, we could state that the set of accepting states that would be $\{s \in Q | \Phi_c(s) \Rightarrow \xi\}$. Here we will introduce a more powerful tool: the binary relation automaton.

8.2 Binary relation over binary tree with automata

Definition 15. *Let $\mathcal{L}$ be a regular tree language over an alphabet $\mathcal{A} \times \{0,1\}^2$, T a binary tree, l a function labeling T with the alphabet $\mathcal{A}$.*

If a is a function labeling T with the alphabet $\{0,1\}^2$, we say that a is an annotation. For an annotation we define a function labeling T with $\mathcal{A} \times \{0,1\}^2$ by $l_a(n) = (l(n), a(n))$

For any annotation a, such that T labeled by l_a is in $\mathcal{L}$, we ensure that it exists x and y with either $y = x \wedge a(x) = (1,1)$ and $z \neq x \Rightarrow a(z) = (0,0)$ or with $a(x) = (1,0)$, $a(y) = (0,1)$ and $\forall z, z \neq y \wedge z \neq x \Rightarrow a(z) = (0,0)$. Given a x and a y there is only one annotation satisfying those requirements so we can write $a_{(x,y)}$.

Because checking that an annotation is of the form $a_{(x,y)}$ - ie the annotation that has on each coordinate one node with a 1 - can be done by a regular tree language, we can suppose there are only annotation a of this form that produces l_a in $\mathcal{L}$.

$\mathcal{R}_T$ is the binary relation over the nodes of T defined like this by $x\mathcal{R}_T y$ iff T labeled by $l_{a_{(x,y)}}$ is in $\mathcal{L}$.

Proposition 1. *Given a tree T,and a labeling l, suppose we have $\mathcal{L}_1$ defining a relation $\mathcal{R}_{T,1}$ and $\mathcal{L}_2$ defining $\mathcal{R}_{T,2}$ then $\mathcal{L}_1 \cap \mathcal{L}_2$ define $x\mathcal{R}_T y \Leftrightarrow x\mathcal{R}_{T,1} y \wedge x\mathcal{R}_{T,2} y$.*

Proof. Let x and y be two nodes. We have $x\mathcal{R}y$ iff T labeled with $l_{a_{(x,y)}}$ is in $\mathcal{L}_1 \cap \mathcal{L}_2$ iff T labeled with $l_{a_{(x,y)}}$ is in $\mathcal{L}_1$ and T labeled with $l_{a_{(x,y)}}$ is in $\mathcal{L}_2$ iff $x\mathcal{R}_1 y$ and $x\mathcal{R}_2 y$. □

Proposition 2. *Given a tree T,and a labeling l, suppose we have $\mathcal{L}_1$ defining a relation $\mathcal{R}_{T,1}$ and $\mathcal{L}_2$ defining $\mathcal{R}_{T,2}$ then $\mathcal{L}_1 \cup \mathcal{L}_2$ define $x\mathcal{R}_T y \Leftrightarrow x\mathcal{R}_{T,1} y \vee x\mathcal{R}_{T,2} y$.*

Proposition 3. *Given a tree T,and a labeling l, suppose we have $\mathcal{L}$ defining a relation $\mathcal{R}$ then $\bar{\mathcal{L}}$ define $\neg x\mathcal{R}_T y$.*

8.3 Nominals

To make use of binary relation with automata it might be useful to have attributes for which we are sure they are only present once in the tree. The most easiest way is to add to the formula. To ensure that the attribute c is present only one in the formula, we write :

$$\neg\mu X = \langle 1\rangle\,(X \vee c) \vee \langle 2\rangle\,(X \vee c)\ in\ X \wedge \neg\mu X = \langle \bar{1}\rangle\,(X \vee c) \vee \langle \bar{2}\rangle\,(X \vee c)\ in\ X \wedge c$$

An attributes for which we made sure it was unique can be called a nominal.

8.4 Path formulas

8.5 Motivation

We consider $\varphi = context \wedge \langle 1\rangle\,(\mu X = (a \wedge \langle 2\rangle\,(select \wedge b)) \vee \langle 2\rangle\,X\ in\ X)$ where a and b are labels and *context* and *select* are attributes. φ selects all trees where a node with the attribute *context* has a child a who has a brother b with an attribute *select*. If we ensure *context* and *select* are nominals then this correspond to the XPath query `child::a/following-sibling::b`. The test the containment of this query into $\psi = context \wedge \langle 1\rangle\,(\mu X = (a \wedge select) \vee \langle 2\rangle\,X\ in\ X) \vee \langle 1\rangle\,(\mu X = b \vee \langle 2\rangle\,X\ in\ X)$ corresponding to `self::*[child::b]/child::a` can be done by building the automata for φ and $\neg\psi$, make the intersection of the two and then test the emptiness. The automaton will also permit to give counter-example.

8.6 Translation of XPath to formula with context and select nominals

In order to define XPath into path automata we need to add the attributes context and select. Context was already introduced in [2] but it was not a nominal. Once the formula is translated with a context into φ then to add select we just use the formula $\varphi \wedge @select$. Then we make $@context$ and $@select$ nominals and plunge the formula.

9 Practical algorithm

9.1 algorithm

The practical algorithm does not build the exact automaton describe above but an equivalent one. A state is not a type but a set of type, this leads to memory efficiency and speed improvement because many types share the same kind of children. We often perform determinization of the automata we build because deterministic automata have a minimal form that is, in general very compact. (In the tests, all minimal deterministic representation of automata were smaller than the automata that would have been built with the above description, despite a theoretical exponential blow-up).

Algorithm 1 BUILD

Input: S a set of types
Output: The name of the state created

1: Memoize S
2: $name \leftarrow newname()$
3: **for all** $t \in S$ **do**
4: $\quad left \leftarrow BUILD(SUCCS(j, left))$
5: $\quad right \leftarrow BUILD(SUCCS(k, right))$
6: $\quad rules \leftarrow rules \cup \{ \overset{\mathcal{L}(i)}{\underset{left}{\swarrow}\quad\underset{right}{\searrow}} \rightarrow name\}$
7: **end for**
8: **return** $name$

10 Results

The implementation of tree automata was first a naive translation of their pseudo-code in Java. Because it was too slow to be useful some parts of the code were rewritten to improve speed. As explained in [5], a much better representation can be chosen for the automata, leading to great speed improvements and more memory efficiency. But, even without a complex representation of automata, rewriting the automata management with a precise control over the memory used and a simple profiling of the code can lead to, at least, a gain of a factor 5 in speed and in memory footprint. Because the other part of the code is very well optimized, in small test cases the old algorithm is faster, but, in large test cases the algorithm using automata out-powers the old one.

Algorithm 2 SUCCS

Input: t a type, $side \in \{left, right\}$
Output: S the set of types compatible with t on the side $side$

$repr = t \cap USEFULLFOR(side)$
Memoize $(repr, side)$
$res = Fixpoint$
for all $\varphi \in \mathcal{L}ean$ **do**
  if $\varphi = \langle side \rangle \psi \in t$ or $(\psi = \langle \bar{side} \rangle \varphi$ and $\varphi \in t)$ **then**
    $res \leftarrow \{e \in res, \psi \in e\}$
  else
    $res \leftarrow \{e \in res, \psi \notin e\}$
  end if
end for
return $BUILD(res)$

For these reasons a complete comparison of the two algorithm is not necessary nor useful. But, for example, on the DTD xhtml, applying all the following XPath expression is done by the algorithm in 30s whereas the old XML logic solver was suspended after several hours of calculation.

```
select("*[not (ancestor::*/descendant::b[ancestor::i])]")
select("*[not (ancestor::*/descendant::img[not (ancestor::body)])]")
select("*[not (ancestor::*/descendant::img[not (parent::p)])]")
select("*[not (ancestor::*/descendant::p[parent::a])]")
select("*[not (ancestor::*/descendant::b[ancestor::i])]")
select("*[not (ancestor::*/descendant::img[parent::p])]")
select("*[not (ancestor::*/descendant::img[*])]")
select("*[not (ancestor::*/descendant::a[ancestor::a])]")
select("*[not (ancestor::*/descendant::div[parent::b])]")
select("*[not (ancestor::*/descendant::h1[ancestor::h2])]")
```

References

[1] Nadime Francis, Claire David, and Leonid Libkin. A Direct Translation from XPath to Nondeterministic Automata. In *5th Alberto Mendelzon International Workshop on Foundations of Data Management*, 2011.

[2] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of xml paths and types. *SIGPLAN Not.*, 42:342–351, June 2007.

[3] Leonid Libkin and Cristina Sirangelo. Reasoning about xml with temporal logics and automata. In *In LPAR'08*, pages 97–112, 2008.

[4] Maarten Marx. Conditional xpath, the first order complete xpath dialect, 2004.

[5] Hendrik Maryns and Universität Tübingen. On the implementation of tree automata: Limitations of the naive approach. In *In Proc. 5th Int. Treebanks and Linguistic Theories Conference (TLT 2006*, pages 235–246, 2006.

[6] Klaus Reinhardt. *The complexity of translating logic to finite automata*, chapter 8, pages 231–238. Springer-Verlag New York, Inc., New York, NY, USA, 2002.