INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

XML Schema, Tree Logic and Sheaves Automata

Silvano Dal Zilio — Denis Lugiez

N° 4631

Novembre 2002

THÈME 1

Rapport de recherche

ISSN 0249-6399

XML Schema, Tree Logic and Sheaves Automata

Silvano Dal Zilio , Denis Lugiez*

Thème 1 — Réseaux et systèmes
Projet Mimosa

Rapport de recherche n 4631 — Novembre 2002 — 31 pages

Abstract: We describe a new class of tree automata, and a related logic on trees, with applications to the processing of XML documents and XML schemas.

XML documents, and other forms of semi-structured data, may be roughly described as edge labeled trees. Therefore it is natural to use tree automata to reason on them and try to apply the classical connection between automata, logic and query languages. This approach has been followed by various researchers and has given some notable results, especially when dealing with *Document Type Definition* (DTD), the simplest standard for defining XML documents validity. But additional work is needed to take into account XML schema, a more advanced standard, for which regular tree automata are not satisfactory. A major reason for this inadequacy is the presence of an associative-commutative operator in the schema language, inherited from the &-operator of SGML, and the inherent limitations of regular tree automata in dealing with associative-commutative algebras.

The class of automata considered here, called *sheaves automata*, is a tailored version of automata for unranked trees with both associative and associative-commutative symbols already proposed by the authors. In order to handle both ordered and unordered operators, we combine the transition relation of regular tree automaton with regular word expression and counting constraints. This extension appears quite natural since, when no counting constraints occurs, we obtain *hedge automata*, a simple model for XML schemata, and when no constraints occur, we obtain regular tree automata.

Building on the classical connection between logic and automata, we also present a decidable tree logic that embeds XML Schema as a plain subset.

Key-words: XML, XML Schema, tree automata, modal logic

* The authors work at the *Laboratoire d'Informatique Fondamentale de Marseille*, UMR 6166, CNRS et Université de Provence, and are partly supported by ATIP "Fondements de l'Interrogation des Données Semi-Structurées" and IST Profundis.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Schéma XML, logique d'arbres et Sheaves Automata

Résumé : Nous proposons une nouvelle classe d'automates d'arbres, ainsi qu'une logique modale d'arbres associée, spécialement dédiée à la manipulation de documents et de schémas XML.

Les documents XML, comme de nombreuses autres formes de données semi-structurées, peuvent être grossièrement décrits par des arbres étiquetés. Il apparaît alors naturel d'utiliser des automates d'arbres pour raisonner sur ces documents et d'essayer d'appliquer la relation classique qui lie automates, logiques et langage de requêtes. Cette approche a déjà été suivie et a donnée plusieurs résultats intéressants, en particulier dans l'étude des *Document Type Definition*, ou DTD, le standard le plus simple permettant de définir des critères de validité pour les documents XML.

Des recherches supplémentaires sont nécessaires pour prendre en compte les schémas XML, un standard plus avancé, pour lequel les automates d'arbres réguliers ne sont pas satisfaisants. Une des raisons principale de cette inadéquation est la présence d'un opérateur associatif-commutatif dans la définition des schémas, hérité de l'opérateur & de SGML, et les limitations inhérentes des automates d'arbres réguliers à traiter des algèbres abélienne.

La classe d'automates considérée dans ce rapport, appelée *sheaves automata*, est une restriction spéciale d'une classe d'automates d'arbres déjà proposée par les auteurs et qui manipule des arbres de degré non borné muni de symboles associatif-commutatif. De manière à pouvoir prendre en compte, à la fois, des opérateurs manipulant des données ordonnées et des opérateurs manipulant des données non ordonnées, nous avons ajouté aux règles de transitions des automates d'arbres, des contraintes de séquencement, sous forme d'expressions régulières, et des contraintes de comptage, sous forme de formule arithmétique. Cette extension apparaît naturelle puisque, lorsque l'on retire les contraintes de comptage on obtient les *hedge automata*, un modèle simple pour les schémas XML, et lorsque l'on retire toutes les contraintes on obtient les automates d'arbres réguliers.

En se basant sur cette nouvelle classe d'automates, nous présentons une logique modale pour les arbres qui est décidable et qui étend de manière naturelle les schémas XML.

Mots-clés : XML, Schéma XML, automates d'arbres, logique modale

1 Introduction

We describe a new class of tree automata, and a related logic on trees, with applications to the processing of XML documents and XML schemas [1]. This work is interesting from a programming language point of view since it could provide a structured and uniform approach to the implementation of tools targeting XML documents, in the same way that, for example, pushdown automata is the established "assembly language" for parsing applications.

XML documents, and other forms of semi-structured data [2], may be roughly described as edge labeled trees. It is therefore natural to use tree automata to reason on them and try to apply the classical connection between automata, logic and query languages. This approach has already been followed by various researchers, both from a practical and a theoretical point of view, and has given some notable results, especially when dealing with *Document Type Definition* (DTD), the simplest standard for defining XML documents validity.

A good example is the XDuce system of Pierce, Hosoya *et al.* [3], a typed functional language with extended pattern-matching operators for XML documents manipulation. In this tool, the types of XML documents (or *regular expression types*) are modeled by regular tree automata and the typing of pattern matching expressions is based on closure operations on automaton. Another example is given by the *hedge automaton* theory [4], an extension of regular tree automaton for unranked trees (that is, tree with nodes of unfixed and unbounded degrees.) Hedge automata are at the basis of the implementation of RELAX-NG [5], an alternative proposal to XML Schema. Various extension of tree automata [6] and monadic tree logic [7] have also been used to study the complexity of manipulating tree structured data but, contrary to our approach, these work are not directly concerned with schemas and are based on ordered content models. More crucially, several mentions to automata theory appear in the XML specifications, principally to express restrictions on DTD and Schemas in order to obtain (close to) linear complexity for simple operations.

Document type definitions are expressed in a language akin to regular expressions and specify the set of elements that may be present in a valid document, as well as constraining their occurrences order. Nonetheless, the "document types" expressible by means of DTD are sometimes too rigid and, for example, a document may become invalid after permutation of some of its elements. A new standard, XML Schema, has been proposed to overcome some of the limitations of the DTD model. In particular, we can interpret XML schemata as terms built using both associative and associative-commutative (AC) operators with unbounded arity, a situation for which regular tree automata are not satisfactory. Indeed, while regular tree automata constitute a useful framework, it has sometimes proved inadequate for practical purposes and many applications require the use of an extended model. To the best of our knowledge, no work so far has considered unranked trees with both associative and associative-commutative symbols, a situation found when dealing with XML Schemata.

We propose a new class of tree automata, named *sheaves automata*, for dealing with XML documents and schema. We believe it is the first work on automata theory applied to XML that consider the "all-group" composition (a simplified version of the SGML &-connector.)

By restricting our study to deterministic automaton, we obtain a class of recognizable languages that enjoys good closure properties and we define a related modal logic for documents that is decidable and exactly matches the recognizable languages. A leading goal in the design of our logic is to include a simplified version of XML Schema as a plain subset.

The content of this paper is as follows. Section 2 introduces the syntax of XML documents and XML schema [1]. XML documents are the basic objects considered in this work. A *document* is modeled as an ordered sequences of unranked, labeled trees. In this framework, an *element* refers to a single rooted tree, and an element's *tag* refers to the label of its root. We also introduce a notion of *document schema*, that can be viewed as a type system for documents. A distinctive aspect of the simplified schema language studied in this paper is to include the $\&$-operator, that is used to state properties of a document regardless of the order of its elements.

In Section 4, we present a decidable tree logic intended for querying XML documents, the Sheaves Logic (SL). This logic can be interpreted as a direct extension of the schema "type system" with logical operators. The sheaves logic deliberately resembles (and extends on some points) TQL, a query language for semi-structured data based on the *ambient logic* [8, 9]. We present here a similar logic, with the difference that we deal both with ordered and unordered data structures, while TQL only deals with multiset of elements. Another difference with TQL lies in the addition of arithmetical constraints. In this extended logic, it becomes for instance possible to express cardinality constraints on the occurrences of an element in a document, such as "there are more fields labeled a than labeled b" or "there is an even number of fields labeled a."

While the addition of counting constraints was purely motivated by the presence of commutative operators, it may incidentally provide a model for *cardinality constraint on repetitions*. These constraints, generally denoted $e\{m, n\}$ in regular word expressions, match k repetition of the expression e, where k, m and n are integers such that $m \leqslant k \leqslant n$, and are also found in the XML Schema specification. Although this is an enjoyable coincidence, the study of cardinality constraints on repetition is not one of our goals.

In Section 5, we introduce a new class of automaton for unranked trees with both associative and associative-commutative symbols, called *Sheaves Automata* (SA). These automata are a tailored version of automata already proposed by the authors [10]. In the transition relation of sheaves automata, we combine the general rules for regular tree automata with regular word expression and counting constraints. In this framework, regular word expressions allow to express constraints on sequences of elements and are used when dealing with associative operators, as in the *hedge automata* approach. Correspondingly, the counting constraints are used with associative-commutative operators.

The counting constraints are *Presburger's arithmetic* constraints over the number of occurrences of each different type of elements. Intuitively, counting constraints appear as the counterpart of regular expressions in the presence of a commutative composition operator. Indeed, when the order of the elements becomes irrelevant, that is, when we deal with bags instead of sequences, the only pertinent constraints are arithmetical. We choose Presburger's constraints because they naturally appear when we extend schema with a fixpoint operator and also because they represent a large and decidable class of constraints over positive natural numbers. Moreover, while the complexity of many operations on Presburger's arithmetic is hyper-exponential (in the worst case), the constraints observed in practice are very simple and it seems possible to neglect the complexity of constraints solving in realistic circumstances. Indeed, some simple limitations on the acceptable schemas, such as those found in the W3C recommendations, are likely to yield algorithms with polynomial or linear complexity.

Our extension of regular tree automaton appears quite natural since, when no counting constraints occurs, we obtain *hedge automata*, a simple yet effective model for XML schemata, and when no constraints occur, we obtain regular tree automata. Moreover, the class of languages recognized by sheaves automata enjoys many of the typical properties of regular languages: closure under union and intersection, decidability of the test for emptiness, ... as well as some new ones, like closure properties under composition by associative and associative-commutative operators. Nonetheless, due to the combination of both regular word expressions and counting constraints, we have no determinisation procedure in the general case, and therefore it is not always possible to compute the complement of accepted languages. However, deterministic SA are closed under all the boolean operations, including complementation.

Before concluding, we give some results on the complexity of basic problems for schemas. By design, every formula of our extended tree logic directly relates to a deterministic sheaves automaton. As a consequence, we obtain the decidability of the *model-checking problem* for SL, that is finding the answers to a query, and of the *satisfiability problems*, that is finding if a query is trivially empty. Moreover, since schemas are directly embedded in the models of SL, we can relate a XML schema to an accepting sheaves automaton obtaining the decidability of all basic problems concerning schemas: checking that a document conforms to a schema, computing the set of documents typed by a schema, computing the set of documents typed by the difference of two schemas ...

2 Documents and Schemata

XML documents are a simple textual representation for unranked, edge labeled trees, that is trees with an unfixed (and an unbounded) number of children at each nodes. In this report, we follow the notations found in the XDuce system [3] and choose a simplified version of XML documents by leaving aside attributes among other details. Most of the simplifications and notation conventions taken here are also found in the presentation of MSL [11], an attempt to formalize some of the core idea found in XML Schema.

2.1 Documents

A document, d, is an ordered sequence of elements, $a_1[d_1] \cdot \ldots \cdot a_n[d_n]$, where a_i is a tag name and d_i is a sub-document. A document may also be empty, denoted ϵ, or be a constant.

We consider given sets of atomic data constant partitioned into primitive data types, like `String` or `Integer` for instance. Documents may be concatenated, denoted $d_1 \cdot d_2$, and this composition operation is associative with identity element ϵ. In order to simplify our presentation, we only consider that the set of tag names and the set of data constants are both finite, but all our results can be extended to the case of infinite sets.

Elements and Documents

$e ::=$	element
$a[d]$	element labeled a, containing d
$d ::=$	document
ϵ	empty document
cst	constant (any type)
$a[d]$	element
$d_1 \cdot d_2$	document composition

Example 1 *A typical entry of a bibliographical database could be the document:*

$$book[title["Art\ of\ Computer\ Programming"] \cdot author["Knuth"] \cdot year[1970]]$$

■

2.2 XML Schemas

The definition of XML Schema mostly follows the presentation made in MSL [11]. Nonetheless, we bring some simplifications and modifications to better fit our objective. In particular, we consider three separate syntactical categories: E for element schema definitions, S for (regular) schemata, and T for schemata that may only appear at top level of an element definition.

Schemas

$E ::=$	Element schema
$a[T]$	element with tag a and interior matching T
$a[T]?$	optional element
`Datatype`	datatype constant
$S ::=$	Regular schema
ϵ	empty schema
E	element
S_1, S_2	sequential composition
$S \mid S$	choice
$S*$	indefinite repetition (Kleene star)
$T ::=$	Top-level schema
$AnyT$	any type (match everything)

S	regular schema
E_1 & $\ldots$ & E_n	interleaving composition

A schema is basically a regular expression that constrains the order and number of occurrences of elements in a document. An element, $a[T]$, describes documents that contains a single top element tagged with a and enclosing a sub-document satisfying the schema T. An optional element, $a[T]?$, matches one or zero occurrence of $a[T]$.

The constructors for (regular) schemata include the standard operators found in regular expression languages, where S, S' stands for concatenation and $S \mid S'$ for choice. For simplicity reasons, we have chosen both iteration, $S*$, and option, $a[T]?$, instead of the repetition operator $S\{m, n\}$ found in the Schema recommendation.

The most original operator is the *interleaving operator*, E_1 & $\ldots$ & E_n, which describes documents containing (exactly) elements matching E_1 to E_n regardless of their order. Our simplified Schema definition also contains a constant, $AnyT$, which stands for the most general type, or *anyType* in the XML Schema terminology [1], which does not constrain its content in any way. For example, the following schema states that a valid book entry has an author name, a title and possibly a publication year (in any order).

Example 2 *Assuming that* `String` *and* `Year` *are the datatype associated to string and date constants, the following schema matches the book entry given in Example 1:*

$$book[\ author[\texttt{String}]\ \&\ title[\texttt{String}]\ \&\ year[\texttt{Year}]?\ ]$$

■

In this report, like in the presentation of MSL, we prefer to stick to an algebraic definition of schemas instead of using the concrete syntax given in the XML Schema specification [1]. In particular, in the XML Schema specification, composite schemas are formed using *groups of elements* instead of the basic operators introduced here. For instance, using XML schema terminology, the sequential composition $(E_1, \ldots, E_n)$ corresponds to a *sequence group* and the tensor product (E_1 & $\ldots$ & E_n) corresponds to an *all group*. We will retain this vocabulary in the remainder of this text.

The distinction of a top-level schema allows expressing some of the constraints associated to the interleaving operator, like for example that & must appear as the sole child at the top of an element schema. For instance, under this restriction, the terms $E_1, (E_2\ \&\ E_3)$ and $(E_1\ \&\ E_2)*$ are ill-formed. Another restriction found in the XML Schema specification is obtained by limiting the option operator to element schemas, precluding terms of the form $(E_1\ \&\ E_2)?\ \&\ E_3$. An advantage of our approach is that, like in the XML Schema specification, optional elements may still be used inside an interleaving operator. For instance, the term $(E_1\ \&\ E_2?)$ corresponds to a valid schema.

To capture some situations arising in practice, we enrich schemata by recursive definitions presented by a system of equations. This can be simply obtained by enriching the syntax of schemas with a set of (schema) variables, ranged over by $\mathcal{X}, \mathcal{Y}, \ldots$. Then, a recursive schema is an expression of the following form, where $\mathcal{X}_1, \ldots, \mathcal{X}_n$ are free schema variables occurring in $S, S_1, \ldots, S_n$.

$$S \textbf{ where } \mathcal{X}_1 = S_1, \ldots, \mathcal{X}_n = S_n$$

The **where** construct is a binder for the variables $\mathcal{X}_1, .., \mathcal{X}_n$ in $S, S_1, \ldots, S_n$ and we consider this operator up-to α-renaming of bound names[1]

Example 3 *We may want to extend book entries with a ref element, that lists the entries cited in the book. This is possible using the following recursive schema, where $\mathcal{B}ook$ is a schema variable, meaning that a book entry may also contain a sequence of book in its ref element:*

$$\begin{array}{lll} \mathcal{B}ook \textbf{ where } & \mathcal{B}ook & = book[\ author[\texttt{String}] \ \& \ title[\texttt{String}] \ \& \ \mathcal{R}ef\ ], \\ & \mathcal{R}ef & = ref[\ \mathcal{B}ook* \ ]? \end{array}$$

This example may be expressed in the following way using the concrete syntax found in the XML Schema specification:

```
<xsd:element name="book" type="BookType"/>
<xsd:element name="ref"  type="RefType"/>

<xsd:complexType name="BookType">
 <xsd:all>
   <xsd:element name="author" type="xsd:string"/>
   <xsd:element name="title" type="xsd:string"/>
   <xsd:element ref="ref" minOccurs="0"/>
 </xsd:all>
</xsd:complexType>

<xsd:complexType name="RefType">
  <xsd:sequence>
    <xsd:element ref="book" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

■

Next, we explicit the role of Schema as a type system for documents.

2.3 Schema as types for documents

We define the relation $\Gamma \vdash d : S$, meaning that the document d satisfies the schema S under the hypothesis Γ, where Γ is a mapping from variables to schema. We simply write this relation $d : S$ when the set of hypothesis is empty.

[1] The XML Schema specification provides the ability to name each group definition. We do not consider this extension here, but recursive variables may provide a nice way to model naming.

The empty typing environment is denoted $\emptyset$ and $\Gamma, \mathcal{X} : S$ denotes the mapping Γ extended with an association between $\mathcal{X}$ and S. The domain of Γ is denoted $dom(\Gamma)$ and $\Gamma(\mathcal{X}) = S$ means that the variable $\mathcal{X}$ is mapped to S in Γ. In the following, we only consider well-formed hypothesis, denoted $\Gamma \vdash \diamond$, such that there is only one schema associated to each variable in $dom(\Gamma)$.

Good Environments

$$\frac{}{\emptyset \vdash \diamond} \qquad \frac{\Gamma \vdash \diamond \quad \mathcal{X} \notin dom(\Gamma)}{\Gamma, \mathcal{X} : S \vdash \diamond}$$

The relation $\Gamma \vdash d : S$ is based on an auxiliary function, $inter(d)$, which computes the *interleaving* of the elements in d, that is all the documents obtainable from d after permutation of its elements. More formally, $inter(\epsilon) = \{\epsilon\}$ and, if d is the document $e_1 \cdot \ldots \cdot e_n$, then $inter(d)$ is the set containing the documents $e_{\sigma(1)} \cdot \ldots \cdot e_{\sigma(n)}$ for all permutation σ of the interval $1..n$.

Good Documents

$$\frac{\Gamma \vdash d : T}{\Gamma \vdash a[d] : a[T]} \qquad \frac{\Gamma \vdash d : T}{\Gamma \vdash d : a[T]?} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \epsilon : a[T]?} \qquad \frac{\Gamma \vdash \diamond \quad \mathrm{cst} \in \mathtt{Datatype}}{\Gamma \vdash \mathrm{cst} : \mathtt{Datatype}}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \epsilon : \epsilon} \qquad \frac{\Gamma \vdash d_1 : S_1 \quad \Gamma \vdash d_2 : S_2}{\Gamma \vdash d_1 \cdot d_2 : S_1, S_2} \qquad \frac{\Gamma(\mathcal{X}) = S \quad \Gamma \vdash d : S}{\Gamma \vdash d : \mathcal{X}}$$

$$\frac{\Gamma \vdash d : S}{\Gamma \vdash d : S \mid S'} \qquad \frac{\Gamma \vdash d : S'}{\Gamma \vdash d : S \mid S'} \qquad \frac{\Gamma \vdash d_1 : S, \ldots, \Gamma \vdash d_n : S}{\Gamma \vdash d_1 \cdot \ldots \cdot d_n : S*}$$

$$\frac{\Gamma, \mathcal{X}_1 : S_1, \ldots, \mathcal{X}_n : S_n \vdash \diamond \quad \Gamma, \mathcal{X}_1 : S_1, \ldots, \mathcal{X}_n : S_n \vdash d : S}{\Gamma \vdash d : S \ \textbf{where}\ \mathcal{X}_1 = S_1, \ldots, \mathcal{X}_n = S_n}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash d : AnyT} \qquad \frac{d \in inter(e_1 \cdot \ldots \cdot e_n) \quad \Gamma \vdash e_1 : E_1, \ldots, \Gamma \vdash e_n : E_n}{\Gamma \vdash d : E_1 \ \& \ \ldots \ \& \ E_n}$$

In the next section, we introduce some basic mathematical tools that will be useful in the definition of both our tree logic and our new class of tree automata.

3 Basic Results on Presburger's Arithmetic and Words

Some computational aspects of our tree automaton rely on arithmetical properties over the group $(\mathbb{N}, +)$ of positive natural numbers with addition. The first-order theory of equality on this structure is also known as Presburger's arithmetic. In this section we review some basic results on Presburger's arithmetic and the relation with semilinear sets.

A possible presentation for Presburger's arithmetic is given by the following grammar, where *Exp* is an integer expression and ψ is a Presburger's formula, also called *Presburger's constraint*. We use $N, M, \ldots$ to range over integer variables and $n, m, \ldots$ to range over natural numbers.

Presburger's Constraint

$Exp ::=$	Integer expression
n	positive integer constant
N	positive integer variable
$Exp_1 + Exp_2$	addition
$\phi, \psi, \ldots ::=$	Presburger's constraint
$(Exp_1 = Exp_2)$	test for equality
$\neg\phi$	negation
$\phi \vee \psi$	disjunction
$\exists N.\phi$	existential quantification

Presburger's constraints allow the definition of flexible, yet decidable, properties over positive integer like for example: the value of X is strictly greater than the value of Y, using the formula $\exists Z.(X = Y + Z + 1)$; or X is an odd number, $\exists Z.(X = Z + Z + 1)$.

We denote $\phi(X_1, \ldots, X_p)$ a Presburger's formula with free integer variables $X_1, \ldots, X_p$ and we shall simply write $\models \phi(n_1, \ldots, n_p)$ when $\phi(n_1, \ldots, n_p)$ is satisfied. We will also use the constant *True* for tautologies, like for example the formulas $(N = N)$ or $(N = M) \vee \neg(N = M)$.

3.1 Presburger Arithmetic and Semilinear Sets

Presburger's arithmetic has a decidable theory. This result can be proved using a connection with *semilinear sets* of natural numbers. More precisely, the models of Presburger's arithmetic formulas in free variables $N_1, \ldots, N_n$ are the semilinear sets of $\mathbb{N}^n$, the set of integer vectors of size n. Addition on $\mathbb{N}^n$ is defined as the pointwise addition on the vectors coordinates.

A *linear set* of $\mathbb{N}^n$, $L(\mathbf{b}, P)$, is a set of vectors generated by a basis, $\mathbf{b} \in \mathbb{N}^n$, and a set of periods, $P = \{\mathbf{p}_1, \ldots, \mathbf{p}_k\}$, with $\mathbf{p}_i \in \mathbb{N}^n$ for all $i \in 1..k$. A linear set $L(\mathbf{b}, P)$ consists of all the possible elements obtained by linear combination of the periods with the base, and a *semilinear set* is a finite union of linear sets.

$$L(\mathbf{b}, P) = \{\mathbf{b} + \sum_{i \in 1..k} \lambda_i \mathbf{p}_i \mid \lambda_1, \ldots, \lambda_k \in \mathbb{N}\}$$

An important result is that semilinear sets are closed under *union*, *sum* and *iteration* [10], where: $L+M = \{x+y \mid x \in L, y \in M\}$, and $L^n = L+\ldots+L$ (n times) and $L* = \bigcup_{n \geq 0} L^n$. In the case of iteration, the semilinear set $L*$ may be a union of exponentially many linear sets (in the number of linear sets in L.)

Another useful mathematical tool needed in the presentation of our new class of automaton is the notion of *Parikh mapping*.

3.2 Parikh mapping

Given some finite alphabet $\Sigma = \{a_1, \ldots, a_n\}$, that we consider totally ordered, the Parikh mapping of a word w of Σ^* is a n-uple of natural numbers, $\#(w) = (m_1, \ldots, m_n)$, where m_i is the number of occurrences of the letter a_i in w. We shall also use the notation $\#_a(w)$ to denote the number of occurrences of the letter a in w, or simply $\# a$ when there is no ambiguity on the word w.

The Parikh mapping of a set of words is the set of Parikh mappings of its elements. This mapping can be easily computed when the set of words is a regular language. Indeed, if *reg* is a regular expression, then $\#(reg)$, the Parikh mapping of *reg*, is the semilinear set obtained as follows.

If a_i is the i^{th} letter in the alphabet, then $\#(a_i)$ is the i^{th} unit vector of $\mathbb{N}^n$, the vector with all elements equal to 0 except the i^{th} that is equal to 1: $(0, \ldots, 0, 1, 0, \ldots, 0)$.

The Parikh mapping for choice is the union of the Parikh mappings:

$\#(reg_1 + reg_2) = \#(reg_1) \cup \#(reg_2)$.

The Parikh mapping for sequential composition is the pointwise addition:

$\#(reg_1.reg_2) = \#(reg_1) + \#(reg_2)$.

The Parikh mapping for iteration, $\#(reg*)$ is the iterate of $\#(reg)$:

$\#(reg*) = \#(reg)*$.

A regular expression associated to a regular word automaton $\mathcal{A}$ can be computed in $O(|\mathcal{A}|^3)$. Therefore, using regular expressions of semilinear sets, we can compute the Parikh mapping of a regular word language in time $O(|\mathcal{A}|^3)$.

3.3 Relation Between Presburger's Arithmetic and XML Schema

We clarify the relation between Presburger's constraint, Parikh's mapping and the semantics of the interleaving operator and try to give an intuition on how the &-operator may add "counting capabilities" to Schema.

The Parikh mapping introduced in the previous section provides a straightforward mapping between documents and tuples of natural numbers. Let $a_1, \ldots, a_p$ be distinct element tags. We can easily obtain an injective mapping, $[\![.]\!]$, from $\mathbb{N}_p$ to the set of documents with the following definition:

$$[\![(n_1,...,n_p)]\!] \;=_{\text{def}}\; \underbrace{a_1[\epsilon]\cdot ...\cdot a_1[\epsilon]}_{n_1 \text{ times}} \cdot ... \cdot \underbrace{a_p[\epsilon]\cdot ...\cdot a_p[\epsilon]}_{n_p \text{ times}}$$

Conversely, if d is a document accepted by the regular word expression $(a_1|\ldots|a_p)*$, then $\#(d)$ is an injective function from documents to $\mathbb{N}^p$.

Suppose that we extend the Schema language with a fixpoint operator, $\mu\mathcal{X}.S$, in the style of recursive type for the λ-calculus [12], and that we slightly relax the syntactic constraints on schemas in order to accept expressions of the form $((E_1 \;\&\; \cdots \;\&\; E_n) \mid E)$ and $(E_1 \;\&\; \cdots \;\&\; E_n \;\&\; \mathcal{X})$. Then, for any Presburger's constraint, ϕ, it is possible to define an (extended recursive) schema that matches the vectors of integers satisfying ϕ.

For example, the schema $\mu\mathcal{X}.((a_1 \;\&\; a_2 \;\&\; \mathcal{X})|\epsilon)$ is associated to the formula $\# a_1 = \# a_2$ (there are as many a_1's than a_2's) and $\mu\mathcal{X}.((a_1 \;\&\; a_1 \;\&\; \mathcal{X})|a_1)$ to the formula $\exists N. \# a_1 = N + N + 1$ (there is an odd number of a_1's.)

Proposition 3.1 *For every Presburger's formula ϕ, there is an extended recursive schema, S, such that $d : S$ if and only if $\models \phi\,\#(d)$.*

Proof Any Presburger's formula, ϕ, is associated to a finite union of linear sets that correspond to the model of ϕ.

Assume the models of ϕ is a single linear set, $L(\mathbf{b}, P)$, with $P = \{\mathbf{p}_1, ..., \mathbf{p}_k\}$. Let $[\![\mathbf{p}]\!]$ and $![\![\mathbf{p}]\!]$ be the following extended recursive schema, where p is the vector $(n_1, ..., n_p)$:

$$[\![\mathbf{p}]\!] \;=_{\text{def}}\; (\underbrace{a_1[\epsilon] \;\&\; ... \;\&\; a_1[\epsilon]}_{n_1 \text{ times}} \;\&\; ... \;\&\; \underbrace{a_p[\epsilon] \;\&\; ... \;\&\; a_p[\epsilon]}_{n_p \text{ times}})$$

$$![\![\mathbf{p}]\!] \;=_{\text{def}}\; \mu\mathcal{X}.(\epsilon \mid \underbrace{a_1[\epsilon] \;\&\; ... \;\&\; a_1[\epsilon]}_{n_1 \text{ times}} \;\&\; ... \;\&\; \underbrace{a_p[\epsilon] \;\&\; ... \;\&\; a_p[\epsilon]}_{n_p \text{ times}} \;\&\; \mathcal{X})$$

Therefore a document d satisfies $[\![\mathbf{p}]\!]$ if and only $\#(d) = \mathbf{p}$. Likewise, a document d satisfies $![\![\mathbf{p}]\!]$ if and only if it is the combination of several documents, $d_1, ..., d_\lambda$, such that $\#(d_i) = \mathbf{p}$ for all $i \in 1..\lambda$, that is, there exists $\lambda \in \mathbb{N}$ such that $\#(d) = \lambda.\mathbf{p}$.

Let S be the extended recursive schema $[\![\mathbf{b}]\!] \;\&![\![\mathbf{p}_1]\!] \;\&\; ... \;\&![\![\mathbf{p}_k]\!]$. Then a document d satisfies the schema S if and only if d is a combination of a document satisfying $[\![\mathbf{b}]\!]$ with (any number) of documents satisfying one the schema $[\![\mathbf{p}_i]\!]$, with $i \in 1..k$. Therefore, d satisfies the schema S if and only if $\#(d)$ is in $L(\mathbf{b}, P)$, that is $\models \phi\,\#(d)$.

Assume the models of ϕ is a combination of the linear sets $L_1, ..., L_m$. From the previous case, we can construct the schemas $(S_i)_{i \in 1..m}$ such that for all $i \in 1..m$, $d : S_i$ if and only if $\#(d) \in L_i$. Let S be the schema $S_1 \mid ... \mid S_m$. Therefore, d satisfies S if and only if there exists an indices $i \in 1..m$ such that $\#(d) \in L_i$, that is $\models \phi\,\#(d)$. ■

We conjecture that this ability to count is exactly circumscribed to Presburger's arithmetic, that is, for every schema denoting a set of natural numbers, there is a Presburger's formula denoting the same set.

In the next section, we introduce a modal logic for documents that directly embeds counting constraint. Indeed, Proposition 3.1 indicates that it is necessary to take into account Presburger's constraints when dealing with the interleaving operator. Moreover, aside from the fact that counting constraints add expressiveness to our logic, a valuable result of adding Presburger's formulas is to obtain a logic with good (and decidable) closure properties.

4 The Sheaves Logic

We extend the schema language with a set of logical operators and relax some of its syntactical constraints in order to define a modal logic for documents, the *Sheaves Logic* (SL). The sheaves logic is a logic in the spirit of the Tree Query Logic (TQL) of Cardelli and Gordon [9], a modal logic for unranked, edge-labeled trees that has recently been proposed as the basis of a query language for semi-structured data. A main difference between TQL and SL is that the latter may express properties on both ordered and unordered sets of trees. In contrast, our logic lacks some of the operators found in TQL like recursion or quantification over tag names, which could be added at the cost of some extra complexity.

The formulas of SL ranged over by $A, B, \ldots$ are given by the following grammar. The formulas are partitioned into three syntactical categories: (1) *elements formula*, E, to express properties of a single element in a document; (2) *regular formulas*, S, corresponding to regular expressions on sequences of elements; (3) *counting formulas*, T, to express counting constraints on bags of elements, that is in the situation where the order of the elements is irrelevant.

Logical Formulas

$E ::=$	Element
$a[S]$	element with tag a and regular formula S
$a[T]$	element with tag a and counting formula T
$AnyE$	any type (match any element)
`Datatype`	datatype constant
cst	constant
$S ::=$	Regular formula
ϵ	empty
E	element
S, S'	sequential composition
$S*$	indefinite repetition (Kleene star)
$S \vee S$	choice
$\neg S$	negation
$T ::=$	Counting formula
$\exists \mathbf{N} : \phi(\mathbf{N}) : N_1 E_1 \ \& \ \ldots \ \& \ N_p E_p$	generalized interleaving ($\mathbf{N} = (N_1, \ldots, N_p)$)
$T \vee T$	choice
$\neg T$	negation

$A, B, \ldots ::=$		Formula
	S	regular formula
	T	counting formula
	$A \vee A$	choice
	$\neg A$	negation

Aside from the usual propositional logic operators, our main addition to the logic is the "Any Element" modality, $AnyE$, and a constrained form of existential quantification, $\exists \mathbf{N} : \phi(\mathbf{N}) : N_1 E_1 \ \& \ \ldots \ \& \ N_p E_p$, that matches documents made of $n_1 + ... + n_p$ elements, with n_i elements matching E_i, regardless of their order, and such that $(n_1, ..., n_p)$ satisfies the formula ϕ.

The improved interleaving operator, inspired by the relation between schema and counting constraint given in Section 3.1, is useful to express more liberal properties on documents than with Schemas. For example, it is now possible to define the type $(E_1 * \ \& \ E_2)$, of documents made only of elements matching E_1 but one matching E_2, using the formula: $\exists N_1, N_2 : (N_1 \geqslant 0) \wedge (N_2 = 1) : N_1 E_1 \ \& \ N_2 E_2$.

The $AnyE$ modality matches documents made of a single element. It has been chosen instead of the less general $AnyType$ schema since it could be used in a constrained existential quantification. In particular, it is possible to model $AnyT$ using the formula $\exists N : (N \geqslant 0) : N\,AnyE$.

4.1 Satisfaction relation

We define the relation $d \models A$, meaning that the document d satisfies the formula A. This relation is defined inductively on the definition of A and the evaluation rules are the same for regular and counting formulas. in the following, we use the symbol Ψ to stand for formulas of sort S, T or A.

Satisfaction

$d \models a[\Psi]$	iff	$(d = a[d']) \ \wedge \ (d' \models \Psi)$
$d \models AnyE$	iff	$(d = a[d'])$
$d \models \texttt{Datatype}$	iff	$(d = \mathrm{cst}) \ \wedge \ (\mathrm{cst} \in \texttt{Datatype})$
$d \models \mathrm{cst}$	iff	$d = \mathrm{cst}$
$d \models \epsilon$	iff	$d = \epsilon$
$d \models S, S'$	iff	$(d = d_1 \cdot d_2) \ \wedge \ (d_1 \models S) \ \wedge \ (d_2 \models S')$
$d \models S*$	iff	$(d = \epsilon) \vee (d = d_1 \cdot \ldots \cdot d_p \ \wedge \ \forall i \in 1..p, d_i \models S)$
$d \models \exists \mathbf{N} : \phi(\mathbf{N}) : N_1 E_1 \ \& \ \ldots \ \& \ N_p E_p$	iff	$\exists n_1, \ldots, n_p, \exists (e_1^j)_{j \in 1..n_1}, \ldots, (e_p^j)_{j \in 1..n_p}$ $e_i^j \models E_i \ \wedge \ \models \phi(n_1, \ldots, n_p) \ \wedge \ d \in inter(e_1^1 \cdot \ldots \cdot e_p^{n_p})$
$d \models \Psi \vee \Psi'$	iff	$(d \models \Psi) \ \vee \ (d \models \Psi')$
$d \models \neg \Psi$	iff	not $(d \models \Psi)$

Given a formula A of SL, the models of A is the set $Mod(A) =_{\text{def}} \{d \mid d \models A\}$ of all documents matching A.

4.2 Example of Formulas

We start by defining some syntactic sugar. The modality *True* will be used for tautologies, that is formulas satisfied by all documents (like $T \vee \neg T$ for instance.) We also define the notation $E_1 \,\&\, \ldots \,\&\, E_p$, for the formula satisfied by documents made of a sequence of p elements matching $E_1, \ldots, E_p$, regardless of their order. Likewise, we define the notation $(a[S] \,\&\, \cdots)$ for the formula satisfied by documents containing at least one element matching $a[S]$:

$$\begin{array}{lll} (E_1 \,\&\, \ldots \,\&\, E_p) & =_{\text{def}} & \exists \mathbf{N} : (N_1 = 1) \wedge \ldots \wedge (N_p = 1) : N_1 E_1 \,\&\, \ldots \,\&\, N_p E_p \\ (a[S] \,\&\, \cdots) & =_{\text{def}} & \exists N_1, N_2 : (N_1 = 1) \wedge (N_2 \geqslant 0) : N_1 a[S] \,\&\, N_2 AnyE \end{array}$$

Let us assume that a book reference is given by the author field, the title and the year of publication. A collection is a sequence of such entries. The references may have been collected in several databases and we cannot be sure of the order of the fields. The following formula matches collections of books written by Knuth or Lamport and that contains as many occurrences of books written by Knuth than written by Lamport.

$$\begin{array}{l} \mathit{collection}[\ \mathit{book}[\ \mathit{author}[\texttt{String}] \,\&\, \mathit{title}[\texttt{String}] \,\&\, \mathit{year}[\texttt{String}]\ ]*\ ] \\ \wedge \\ \mathit{collection}[\ \exists N, M : (N = M) : \begin{array}{l} N\mathit{book}[(\mathit{author}[\text{"Knuth"}] \,\&\, \cdots)] \\ \&\ M\mathit{book}[(\mathit{author}[\text{"Lamport"}] \,\&\, \cdots)] \end{array}\ ] \end{array}$$

Next, we define a new class of tree automata that will be used to decide SL, in the sense that the set of documents matched by a formula will correspond to the set of terms accepted by an automaton.

5 A New Class of Tree Automata

We define a class of automata specifically designed to operate with XML schemata. A main distinction with other automata-based approach, like *hedge automata* [4] for example, is that we do not focus on regular expressions over paths but, instead, concentrate on the &-operator, that is one of the chief addition of XML Schema with respect to DTD.

The definitions presented here have been trimmed down for the sake of brevity, but the framework that we have proposed is far more rich and general. For example, in the complete version of our class of automaton, we consider rich sets of constraints between subtrees [10]. Moreover, the definition of SA can be extended to any signature involving free function symbols and an arbitrary number of associative and AC symbols, giving an elegant way to model XML attributes.

5.1 Sheaves Automata

A (bottom-up) sheaves automaton, $\mathcal{A}$, is a triple $\langle Q_{\mathcal{A}}, Q_{\text{fin}}, R\rangle$ where $Q_{\mathcal{A}}$ is a finite set of states, $\{q_1, \ldots, q_p\}$, Q_{fin} is a set of final states included in $Q_{\mathcal{A}}$, and R is a set of transition rules. Transition rules are of three kinds:

$$\begin{array}{ll}
\text{(Type 1)} & c \to q \\
\text{(Type 2)} & a[q'] \to q \\
\text{(Type 3)} & \phi(N_1, \ldots, N_p) \vdash \mathit{Reg}(Q_{\mathcal{A}}) \to q
\end{array}$$

Type 1 and type 2 rules correspond to the transition rules found in regular tree automata for constants (leave nodes) and unary function symbols. Type 3 rules, or *constrained rules*, are the only addition to the regular tree automata model and are used to compute on nodes built using the concatenation operator (the only nodes with an unbounded arity.) In type 3 rules, $\mathit{Reg}(Q_{\mathcal{A}})$ is a regular expression on the alphabet $Q_{\mathcal{A}} = \{q_1, \ldots, q_p\}$ (with concatenation $\cdot$, union $+$ and iteration $*$) and $\phi(N_1, \ldots, N_p)$ is a Presburger's arithmetic formula in the free variables $N_1, \ldots, N_p$. Intuitively, the variable N_i denotes the number of occurrences of the state q_i in a run of the automata. A type 3 rules may fire if we have a term of the form $d_1 \cdot \ldots \cdot d_n$ such that:

the term d_i leads to a state $q_{j_i} \in Q_{\mathcal{A}}$ for all $i \in 1..n$;

the word $q_{j_1} \cdot \ldots \cdot q_{j_n}$ is in the language defined by $\mathit{Reg}(Q_{\mathcal{A}})$;

the formula $\phi\,\#(q_{j_1} \cdot \ldots \cdot q_{j_n})$ is satisfied, *i.e.* $\models \phi(n_1, \ldots, n_p)$, where n_i is the number of occurrences of q_i in $(q_{j_1} \cdot \ldots \cdot q_{j_n})$.

In order to better stress the connection between the variable N_i, in the formula $\phi(N_1, \ldots, N_p)$, and the number of occurrences of q_i in the words accepted by $\mathit{Reg}(Q_{\mathcal{A}})$ (where $Q_{\mathcal{A}}$ is the set $\{q_1, \ldots, q_p\}$), we will often use $\#\,q_i$ instead of N_i as a variable name in a Presburger's formula, and write type 3 rules in the following way:

$$\text{(Type 3)} \qquad \phi(\#\,q_1, \ldots, \#\,q_p) \vdash \mathit{Reg}(Q_{\mathcal{A}}) \to q$$

Example 4 *Let Σ be the signature $\{c, a[\_], b[\_]\}$ (we always implicitly add sequential composition, $\cdot$, and its unit, ϵ, to the signature.) An example of automaton is given by the set of states $Q_{\mathcal{A}} = \{q_a, q_b, q_s\}$, the set of final states $Q_{fin} = \{q_s\}$ and the set of transition rules R,*

$$\begin{array}{lll}
\epsilon \to q_s & a[q_s] \to q_a & (\#\,q_a = \#\,q_b) \wedge (\#\,q_s \geqslant 0) \vdash (q_a + q_b)* \to q_s \\
c \to q_s & b[q_s] \to q_b &
\end{array}$$

We will see, when we define the transition relation, that this particular automaton accepts terms with as many a's than b's at each node, like for example the terms $a[\epsilon] \cdot b[\epsilon] \cdot b[\epsilon] \cdot a[\epsilon]$ and $b[\epsilon] \cdot a[c \cdot b[\epsilon] \cdot a[\epsilon]]$. ■

If we drop the Presburger's arithmetic constraint and restrict to type 3 rules of the form $\mathit{True} \vdash \mathit{Reg}(Q_{\mathcal{A}}) \to q$, we get *hedge automata* [4]. Conversely, if we drop the regular word expression and restrict to rules of the form[2] $\phi(\# q_1, \ldots, \# q_p) \vdash \mathit{AnyT} \to q$, we get a class of automata which enjoys all the good properties of regular tree automata, that is closure under boolean operations, a determinisation algorithm, decidability of the test for emptiness, ... When both counting and regular word constraints are needed, some of these properties are no longer valid (at least in the case of non-deterministic SA.)

5.2 Transition Relation

The *transition relation* of an automaton $\mathcal{A}$, denoted $d \to_{\mathcal{A}} q$, is the transitive closure of the relation defined by the following three inference rules. When there is no ambiguity on the automaton, we simply write $\to$ instead of $\to_{\mathcal{A}}$.

Transition Relation: $\to_{\mathcal{A}}$

(Type 1)

$$\frac{c \to_{\mathcal{A}} q \in R}{c \to_{\mathcal{A}} q}$$

(Type 2)

$$\frac{d \to_{\mathcal{A}} q' \quad n[q'] \to_{\mathcal{A}} q \in R}{n[d] \to_{\mathcal{A}} q}$$

(Type 3)

$$\frac{\begin{array}{c} e_1 \to_{\mathcal{A}} q_{j_1} \ \ldots \ e_n \to_{\mathcal{A}} q_{j_n} \quad \phi(\# q_1, \ldots, \# q_p) \vdash \mathit{Reg}(Q_{\mathcal{A}}) \to_{\mathcal{A}} q \in R \quad (n \geqslant 2) \\ q_{j_1} \cdot \ldots \cdot q_{j_n} \in \mathit{Reg}(Q_{\mathcal{A}}) \quad \models \phi \, \#(q_{j_1} \cdot \ldots \cdot q_{j_n}) \end{array}}{e_1 \cdot \ldots \cdot e_n \to_{\mathcal{A}} q}$$

The rule for constrained transitions (type 3 rules), can only be applied to sequences of length at least 2. Therefore it could not be applied to the empty sequence, ϵ, or to sequence made of one element, like $a[q]$ for example. It could be possible to extend the transition relation for type 3 rules to these two particular cases, but it would needlessly complicate our definitions and proofs without adding expressivity.

Example 5 *Let $\mathcal{A}$ be the automaton defined in Example 4 and d be the document $a[c] \cdot b[a[c] \cdot b[c]]$. A possible accepting run of the automaton is given below:*

$$\begin{array}{llllll} d & \to & a[c] \cdot b[a[q_s] \cdot b[c]] & \to & a[q_s] \cdot b[a[q_s] \cdot b[c]] & \to & a[q_s] \cdot b[a[q_s] \cdot b[q_s]] \\ & \to & q_a \cdot b[a[q_s] \cdot b[q_s]] & \to & q_a \cdot b[a[q_s] \cdot q_b] & \to & q_a \cdot b[q_a \cdot q_b] \\ & \stackrel{\star}{\to} & q_a \cdot b[q_s] & \to & q_a \cdot q_b & \stackrel{\star}{\to} & q_s \end{array}$$

Transitions marked with a $\star$ symbol (transitions 7 and 9) use the only constrained rule of $\mathcal{A}$. It is easy to check that, in each case, the word used in the constraints is $q_a \cdot q_b$, that this word belongs to $(q_a \mid q_b)*$ and that it contains as many q_a's than q_b's (its Parikh mapping is $(1, 1, 0)$.) More generally, the automaton $\mathcal{A}$ accepts only documents in which every sequences contains as many a's than b's as top elements. ■

[2] In this setting, AnyT refers to the "all-accepting" regular word expression $(q_1 \mid \ldots \mid q_p)*$.

As it is usual with automata, we say that a document (or term) d is *accepted* by a sheaves automaton $\mathcal{A}$ if there is a final state $q \in Q_{\text{fin}}$ such that $t \to_{\mathcal{A}} q$. The language $\mathcal{L}(\mathcal{A})$ is the set of terms accepted by $\mathcal{A}$. Our example shows that SA can accept languages which are very different from regular tree languages, in fact closer to those accepted by context-free languages. For example, the constrained rule in Example 4 can be interpreted as: "*the word $q_1 \cdot \ldots \cdot q_n$ belongs to the context-free language of words with as many q_a's than q_b's.*" It is even possible to write constraints defining languages which are not even context-free, like $q_a^n \cdot q_b^n \cdot q_c^n$ (just take the Presburger's constraint $(\# q_a = \# q_b) \wedge (\# q_b = \# q_c)$ in Example 4.)

5.3 Properties of Sheaves Automata

In this section, we survey several properties of sheaves automata, like the closure under boolean operations or the decidability of the test for emptiness, and we study the complexity of some basic problems, like checking whether a document is accepted by an automaton.

An important practical result is that, contrary to the case of regular tree automata, the class of deterministic sheaves automata is strictly weaker than the class of sheaves automata. We say that a sheaves automaton is *deterministic* if and only if a term reach at most one state. This strict separation between the discriminative power of deterministic and non-deterministic automata is mainly a result of interactions between counting and regular constraints and it retrospectively supports our choice to separate these two sorts of properties in our tree logic.

5.3.1 Deterministic SA are less Powerful than Non-deterministic SA

Let consider the following language, L, over a two letters alphabet, $\Sigma = \{a, b\}$:

$$L = \{\ w_1 \cdot w_2 \cdot w_3 \cdot w_4 \mid \begin{array}{l} w_1, w_3 \in a*, w_2, w_4 \in b*, \\ \#_a\, w_1 = \#_b\, w_2 \geqslant 1, \\ \#_a\, w_3 = \#_b\, w_4 \geqslant 1 \end{array} \quad \}$$

The language L consists of the terms $a^n \cdot b^n \cdot a^m \cdot b^m$, with $n, m > 0$. We can identify each word in L with a document and define a non-deterministic automaton, $\langle Q, Q_{\text{fin}}, R \rangle$, accepting all the documents in L. This automaton is such that $Q = \{qa_1, qa_2, qb_1, qb_2, q_s\}$, with $Q_{\text{fin}} = \{q_s\}$, and has the following five transition rules:

$$\begin{array}{l} a \to qa_1 \qquad b \to qb_1 \qquad a \to qa_2 \qquad b \to qb_2 \\ (\# qa_1 = \# qb_1) \wedge (\# qa_2 = \# qb_2) \vdash qa_1 * \cdot qb_1 * \cdot qa_2 * \cdot qb_2 * \to q_s \end{array}$$

We show that the language L cannot be accepted by a deterministic SA, and therefore prove our separation result between the expressivity of deterministic and non-deterministic sheaves automata.

Proposition 5.1 *There is no deterministic sheaves automaton accepting L.*

Proof Assume there is a deterministic automaton, $\mathcal{A}$, accepting L. Let qa (resp. qb) be the unique state reached by a (resp. b). We will use $\#qa$ and $\#qb$ as the variable names that refer to the number of occurrences of qa and qb in Presburger's constraints.

Given the special structure of the language L, we can assume some extra conditions on the constrained rules of the deterministic automaton. Indeed, in an accepting run of $\mathcal{A}$, a constrained transition rule may only be applied to a word of $(qa|qb)*$. Therefore we may assume that Reg is a regular expression on the alphabet $\{qa, qb\}$ only, and that the only free variables in the formula ϕ are $\#qa$ and $\#qb$.

Since the language L is infinite and that the number of transition rules are finite, there is at least one constrained rule, $(\star)$ $\phi \vdash Reg \to q_s$, such that both ϕ is satisfied by an infinite number of values for $\#qa$ and $\#qb$ and Reg accepts an infinite number of words.

By definition of the language L, the terms accepted by the rule $(\star)$ are of the form $t_{(n,m)} = qa^n \cdot qb^n \cdot qa^m \cdot qb^m$ and, by hypothesis, the set of words $t_{(n,m)}$ accepted by Reg should be infinite. Using a standard "pumping lemma" on the minimal deterministic finite state automaton (FSA) associated to Reg, it must be the case that Reg accepts a much larger set of words. More precisely, if $\max(Reg)$ is the size of the minimal deterministic finite state automaton (FSA) associated with Reg, then there exists two natural numbers, k and l, such that for all $m, n \geqslant \max(Reg)$, if $t_{(m,n)}$ is accepted by Reg, then the following word is accepted by Reg for all $\lambda, \mu \geqslant 0$:

$$qa^{n+\lambda.k} \cdot qb^n \cdot qa^{m+\mu.l} \cdot qb^m$$

The proof of this property is similar to the proof of the standard pumping lemma for FSA and is based on the fact that the number of states in the FSA associated to Reg is finite, whereas the set of recognized words if infinite. Therefore, for each part of an accepted word of size greater than $\max(Reg)$, there should correspond a cycle in the automaton. For example, in the case where $n, m > \max(Reg)$ and $t_{(n,m)}$ is accepted, there are two states q_1, q_2 of the FSA for Reg such that an accepting run for $t_{(n,m)}$ is as follows:

$$\begin{array}{rcl}
\text{position in } t_{(n,m)} & & \quad\; p_1\; p_2 \qquad\qquad\qquad\quad p_3\; p_4 \\
t_{(n,m)} & = & a \cdot \ldots\ldots \cdot a \cdot b \cdot \ldots \cdot b \cdot a \cdot \ldots \cdot a \cdot b \cdot \ldots\ldots \cdot b \\
 & & \quad\;\downarrow\;\downarrow \qquad\qquad\qquad\quad \downarrow\;\downarrow \qquad\qquad\quad \downarrow \\
\text{states reached} & & \quad\; q_1\; q_1 \qquad\qquad\qquad\quad q_2\; q_2 \qquad\qquad\quad q \quad \text{(final)}
\end{array}$$

Let $k = |p_2| - |p_1|$ and $l = |p_4| - |p_3|$. Then k is the length of the part of a^n that can be iterated without modifying the final state reached by $t_{(n,m)}$, and similarly for l and a^m. Moreover, since the automata implementing Reg is deterministic, every accepting run should include the cycles of size k and l that we have identified (for words of sufficient length.)

Next, we choose some values of n, m such that $n, m \geqslant \max(Reg) + k.l$ and that $t_{(n,m)}$ is accepted by $(\star)$. This is always possible since the set of words "accepted" is infinite. Since $n, m \geqslant \max(Reg) + k.l$ we may also write these two numbers $n = n_0 + k.l$ and $m = m_0 + k.l$, with $n_0, m_0 \geqslant \max(Reg)$.

By definition of the transition relation we have both:

(1) $qa^{n_0+k.l} \cdot qb^n \cdot qa^{m_0+k.l} \cdot qb^m$ is accepted by *Reg*

(2) $\models \phi(n_0 + m_0 + 2.k.l, n + m)$

By property (1) and our (extended) pumping lemma, we have that $t = qa^{n_0+2.k.l} \cdot qb^n \cdot qa^{m_0} \cdot qb^m$ is also accepted by *Reg*. Indeed, we only need to "pump" l times the first series of a and to "reverse-pump" k times the second.

By property (2), since the Parikh mapping of t is equal to the mapping of $t_{(n,m)}$, we have that ϕ is satisfied by t. Therefore the word t is accepted by the rule ($\star$), that is by $\mathcal{A}$. This contradicts the fact that t is not in L, the language recognized by the automaton. ■

Next, we define some constructions for basic operations on SA. Since the class of deterministic sheaves automata is strictly weaker than the class of sheaves automata, we will try to preserve determinism as much as possible. This results in constructions for basic operations that are a little bit more complex than the classical ones.

In these constructions, it may be necessary to add a state to an automaton without changing its transition relation. The only complication in doing so is that, in constrained rules, the set of states is used both as the alphabet of the regular expressions and as the set of variables in the Presburger's formula. For the sake of simplicity, we allow to write the new constrained rules as the old ones.

5.3.2 Completion

A sheaves automaton is *complete* when every term reach some state. For instance, the automaton $\mathcal{A}$ defined in Example 4 is not complete since the term $a[c] \cdot a[c]$ does not reach any state (the transitions of the automaton are blocked when it reach the expression $q_a \cdot q_a$.)

From any automaton $\mathcal{A}$, we can build a complete automaton, $\mathcal{A}_c$, that recognize the same language than $\mathcal{A}$. The completed automaton is obtained using the following algorithm:

add a new junk state, say $q_\perp$,

if there is a constant c without any type 1 rules associated to it, add the rule $c \to q_\perp$,

add the rule $a[q_\perp] \to q_\perp$ for any unary function symbol $a[\_]$ in the signature,

add the rule $\mathit{True} \vdash (Q \cup \{q_\perp\})*, q_\perp, (Q \cup \{q_\perp\})* \to q_\perp$,

Let $(\phi_i \vdash \mathit{Reg}_i \to q_{j_i})_{i \in 1..k}$ be the finite family of constrained rules in $\mathcal{A}$, then for every subset I of $1..n$, add the rule: $\bigwedge_{i \in J} \neg\phi_i \vdash \bigcap_{i \notin J} \overline{\mathit{Reg}_i} \to q_\perp$.

Proposition 5.2 *The automaton $\mathcal{A}_c$ is complete, it accepts the same language than $\mathcal{A}$ and it is deterministic if $\mathcal{A}$ is deterministic.*

Proof The proof is by structural induction on d. ■

5.3.3 Product, Union and Intersection

Given two automata $\mathcal{A} = (Q, Q_{\text{fin}}, R)$ and $\mathcal{A}' = (Q', Q'_{\text{fin}}, R')$, we can construct the *product automaton*, $\mathcal{A} \times \mathcal{A}'$, that will prove useful in the definition of the automata for union and intersection. The product $\mathcal{A} \times \mathcal{A}'$ is the automaton $\mathcal{A}^{\times} = (Q^{\times}, \emptyset, R^{\times})$ such that:

$Q^{\times} = Q \times Q' = \{(q_1, q'_1), \ldots, (q_p, q'_l)\}$,

for every type 1 rules $a \to q \in R$ and $a \to q' \in R'$, the rule $a \to (q, q')$ is in $R^{\times}$,

for every type 2 rules $n[q] \to s \in R$ and $n[q'] \to s' \in R'$, the rule $n[(q, q')] \to (s, s')$ is in $R^{\times}$,

for every type 3 rules $\phi \vdash \mathit{Reg} \to q \in R$ and $\phi' \vdash \mathit{Reg}' \to q' \in R$, the rule $\phi^{\times} \vdash \mathit{Reg}^{\times} \to (q, q')$ is in $R^{\times}$, where $\mathit{Reg}^{\times}$ is the regular expression corresponding to the product $\mathit{Reg} \times \mathit{Reg}'$ (this expression can be obtained from the product of an automaton accepting Reg by an automaton accepting Reg'). The formula $\phi^{\times}$ is the product of the two formulas $\phi \times \phi'$ obtained as follows. Let $\#(q, q')$ be the variables associated to the numbers of occurrences of the state (q, q'), then:

$$\phi^{\times} = \phi(\sum_{s \in Q'} \#(q_1, s), \ldots, \sum_{q' \in Q'} \#(q_p, q')) \wedge \phi'(\sum_{q \in Q} \#(q, q'_1), \ldots, \sum_{q \in Q} \#(q, q'_l))$$

Proposition 5.3 *There is a transition $d \to (q, q')$ in the product automaton $\mathcal{A} \times \mathcal{A}'$ if and only if $d \to_{\mathcal{A}} q$ and $d \to_{\mathcal{A}'} q'$.*

Proof The proof is by structural induction on d. ■

Given two automata, $\mathcal{A}$ and $\mathcal{A}'$, it is possible to obtain an automaton accepting the language $\mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{A}')$ and an automaton accepting $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}')$. The intersection $\mathcal{A} \cap \mathcal{A}'$ may be simply obtained from $\mathcal{A} \times \mathcal{A}'$ by setting the set of final states to:

$$Q^{\cap}_{\text{fin}} = (Q_{\text{fin}} \times Q'_{\text{fin}}) = \{(q, q') \mid q \in Q_{\text{fin}} \ \wedge \ q \in Q'_{\text{fin}}\}$$

Similarly, the union $\mathcal{A} \cup \mathcal{A}'$ may be obtained from $\mathcal{A} \times \mathcal{A}'$ by setting the set of final states to:

$$Q^{\cup}_{\text{fin}} = (Q_{\text{fin}} \times Q') \cup (Q \times Q'_{\text{fin}}) = \{(q, q') \mid q \in Q_{\text{fin}} \ \vee \ q \in Q'_{\text{fin}}\}$$

The union automaton may also be obtained using a simpler construction: take the union of the states of $\mathcal{A}$ and $\mathcal{A}'$ (supposed disjoint) and modify type 3 rules accordingly. It is enough to simply add the new states to each type 3 rules together with an extra counting constraint stating that the corresponding coefficients must be nil.

Proposition 5.4 *The automaton $\mathcal{A} \cup \mathcal{A}'$ accepts $\mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{A}')$ and $\mathcal{A} \cap \mathcal{A}'$ accepts $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}')$. Moreover, the union and intersection automaton are deterministic whenever both $\mathcal{A}$ and $\mathcal{A}'$ are deterministic.*

5.3.4 Complement

Given a deterministic automaton, $\mathcal{A}$, we may obtain a deterministic automaton that recognizes the complement of the language $\mathcal{L}(\mathcal{A})$ simply by exchanging final and non-final states. This property does not hold for non-deterministic automata.

Proposition 5.5 *The class of languages accepted by non-deterministic sheaves automata is not closed under complementation.*

Proof We show that given a two-counter machine, there is a non-deterministic automaton accepting the set of bad computations of the machine. Therefore, if the complement of this language was also accepted by some automaton, we could easily derive an automaton accepting the (good) computations reaching a final state, hence decide if the machine halts. The halting problem for deterministic two-counter machine is a well-known undecidable problem.

Two-counter machines are devices made from a finite set of states, $\mathcal{Q}$, some being termed final, a pair of counters, C_1, C_2, which are natural numbers variables, and a transition relation, $\delta \subseteq \mathcal{Q} \times \{0,1\}^2 \times \mathcal{Q} \times \{-1,0,1\}^2$. A configuration, $\mathbb{C}$, is a triple (Q, C_1, C_2), where Q is a state in $\mathcal{Q}$. We say that the configuration $\mathbb{C} = (Q, C_1, C_2)$ derive the configuration $\mathbb{C}' = (Q', C_1', C_2')$, denoted $\mathbb{C} \implies \mathbb{C}'$, if there is some transition $(Q, x_1, x_2, Q', x_1', x_2') \in \delta$ such that for all $i \in \{1,2\}$:

if $C_i = 0$ then $x_i = 0$ else $x_i = 1$, that is we can test whether the first counter is nil or not,

$C_i' = C_i + x_i'$

We also require that if $x_i = 0$ then $x_i' \geqslant 0$, that is, we cannot decrease the value of a null counter. All these conditions can be described by a Presburger's arithmetic formula. For instance, consider the transition rule $(Q, 0, 1, Q', 1, -1)$ that requires that we are in state Q, checks if the first-counter is zero, that the second one is strictly positive, goes to state Q', increments the first counter and decrement the second one. The machine is deterministic when the transition relation is a function. The corresponding operations on counters are described by the following formula, where we may may replace the expression $C_2 > 0$ with the Presburger's formula $\exists N.(C_2 = 1+N)$, and $(C_2' = C_2 - 1)$ with the formula $(C_2' + 1 = C_2)$:

$$(C_1 = 0) \wedge (C_2 > 0) \wedge (C_1' = 1) \wedge (C_2' = C_2 - 1)$$

A computation is a sequence of configurations $\mathbb{C}_0, \mathbb{C}_1, \ldots$ such that for all $i \geqslant 0$, $\mathbb{C}_{i-1} \implies \mathbb{C}_i$. It is undecidable whether there exists a computation that may reach a configuration with a final state, also called a *halting configuration*.

To simulate the computations of a deterministic two-counter machine, we use the following signature.

a constant Q for each state $Q \in \mathcal{Q}$ of the two-counter machine, two constants C_1 and C_2 to indicate the beginning of each counter,

a constant 1 used for counting. We represent the natural number n in unary format, by n successive occurrences of the symbol 1.

With our convention, a sequence of configurations is a document accepted by the word expression $(\bigcup_{Q \in \mathcal{Q}} Q, C_1, 1*, C_2, 1*)*$. Therefore there is a SA accepting the set of all sequences of configurations (a regular automaton will be enough) and also a SA accepting the set of all sequences ending in a halting state. The construction of an automaton accepting only the bad sequences of configurations, that is those not matching the definition of δ, is as follows:

the automaton has states q_Q (for each state of the counter machine $Q \in \mathcal{Q}$), q_{C_1} ,q_{C_2}, 1_{C_1}, 1_{C_2}, $\perp$, as well as a unique final state, q_{error}. The state q_{C_1}, q_{C_2} are used to locate the "start of counter value" symbols C_1 and C_2 and are associated to two type 1 rules: $C_1 \to q_{C_1}$ and $C_2 \to q_{C_2}$;

the constant 1 can reach (non-deterministically) five different states, 1_{C_1}, 1_{C_2}, $1_{C'_1}$, $1_{C'_2}$ and $\perp$. We have five type 1 rules, $1 \to 1_{C_i}$ and $1 \to 1_{C'_i}$ for all $i \in \{1, 2\}$ and $1 \to \perp$. The first four states are used to identify the value of the counter we are interested in, while $\perp$ is used for configurations of the machine whose counters values is not interesting.

there is one constrained rule for each pair of states (Q, Q') such that there is a transition $(Q, x_1, x_2, Q', x'_1, x'_2)$ in δ (we use the wildcard symbol $q\_$ to denote any state of the kind q_Q for $Q \in \mathcal{Q}$):

$$\phi \vdash \left(\begin{array}{c} (q\_, q_{C_1}, \perp*, C_2, \perp*)*, \\ (q_Q, q_{C_1}, 1_{C_1}*, q_{C_2}, 1_{C_2}*), (q_{Q'}, q_{C_1}, 1_{C'_1}*, q_{C_2}, 1_{C'_2}*), \\ (q\_, q_{C_1}, \perp*, q_{C_2}, \perp*)* \end{array} \right) \to q_{error}$$

where $\phi(\# 1_{C_1}, \# 1_{C_2}, \# 1_{C'_1}, \# 1_{C'_2})$ is the Presburger's formula stating that the values of the counters do not agree with any of the transitions in δ from state Q to state Q'.

Let $\mathcal{L}$ be the language recognized by the non-deterministic SA. The intersection of the complement of $\mathcal{L}$ with the language of sequences of configurations ending with a final state is the set of computations of the two-counter machine reaching a final state. If it were accepted by a sheaves automaton, we would have a decision procedure for two-counter machines, which leads to a contradiction. ∎

5.3.5 Membership

In this section, we consider the problem of checking whether a document, d, is accepted by a non-deterministic automaton $\mathcal{A}$. We use the notation $|d|$ for the number of elements occurring in d and $|S|$ for the number of elements in a set S. The size of an automaton, $|\mathcal{A}|$, is the number of symbols occurring in its definition.

Assume there is a function *Const* such that, for all constraints ϕ, the evaluation of $\phi(N_1, \ldots, N_p)$ can be done in time $O(\mathit{Const}(p, n))$ whenever the integer variables $N_1, \ldots, N_p$ are instantiated by values less than or equal to n. For quantifier-free Presburger's formula (and if n is in binary notation) such a function is given by $K.p.\log(n)$, where K is the greatest coefficient occurring in ϕ. For arbitrary situations, that is for formulas involving any quantifiers alternation (which is very unlikely to occur in practice), the complexity is doubly exponential for a non-deterministic algorithm.

Proposition 5.6 *For an automaton $\mathcal{A} = \langle Q, Q_{\mathit{fin}}, R\rangle$, the problem $d \overset{?}{\in} \mathcal{L}(\mathcal{A})$ can be decided in time $O(|d|.|R|.\mathit{Const}(|Q|, |d|))$ for a deterministic automaton and in time $O(|d|^2.|Q|.|R|.$ $\mathit{Const}(|Q|, |d|))))$ for a non-deterministic automaton.*

Proof The proof is standard in the case of deterministic automata. Otherwise, there are $|d|.|Q|$ possible labeling of the tree d by states of Q, and we check the applicability of each rules at each internal node. ∎

5.3.6 Test for Emptiness

We give an algorithm for deciding emptiness that combines a marking algorithm with a test to decide if the combination of a regular expression and a Presburger's constraint is satisfiable.

We start by defining an algorithm for checking when a word on a sub-alphabet satisfies both a given regular word expression and a given counting constraint. We consider a set of states, $Q = \{q_1, \ldots, q_p\}$, that is also the alphabet for a regular expression *Reg* and a Presburger's formula $\phi(\# q_1, \ldots, \# q_p)$.

The problem is to decide whether there is a word on the sub-alphabet $Q' \subseteq Q$ satisfying both *Reg* and ϕ. We start by computing the regular expression $\mathit{Reg}_{|Q'}$ that corresponds to the words on the alphabet Q' satisfying *Reg*. This expression can be easily obtained from *Reg* by applying a set of simple syntactical rewritings to *Reg* until no rule is applicable (where *op* denotes either the choice or concatenation operator):

$$\frac{e \to e'}{e\ op\ f \to e'\ op\ f} \qquad \frac{f \to f'}{e\ op\ f \to e\ op\ f'} \qquad \frac{e \to e'}{e* \to e'*}$$

$$\frac{q \notin Q'}{q \to \bot} \qquad \frac{}{\bot\ op\ f \to \bot} \qquad \frac{}{e\ op\ \bot \to \bot} \qquad \frac{}{\bot* \to \bot}$$

Then we compute the Parikh mapping $\#(\mathit{Reg}_{|Q'})$ (as explained in Section 3.2) and test the satisfiability of the following Presburger's formula:

$$\phi(\# q_1, \ldots, \# q_p) \ \wedge \ \bigwedge_{q \notin Q'} (\# q = 0) \ \wedge \ \#(\mathit{Reg}_{|Q'})$$

When this formula is satisfiable, we say that the constraint $\phi \vdash Reg$ restricted to Q' is satisfiable. This notion is useful in the definition of an updated version of a standard marking algorithm for regular tree automaton.

The following algorithm computes a set $Q_M \subseteq Q$ of states and returns a positive answer if and only if there is a final state reachable in the automaton.

$$
\begin{array}{lll}
Q_M = \emptyset & & \\
\textbf{repeat}\ \textbf{if}\ a \to q \in R & \textbf{then} & Q_M = Q_M \cup \{q\} \\
\qquad\quad \textbf{if}\ n[q'] \to q \in R \text{ and } q' \in Q_M & \textbf{then} & Q_M = Q_M \cup \{q\} \\
\qquad\quad \textbf{if} \begin{cases} \phi \vdash Reg \to q \in R \text{ and} \\ \text{the constraint } \phi \vdash Reg \text{ restricted} \\ \text{to } Q_M \text{ is satisfiable} \end{cases} & \textbf{then} & Q_M = Q_M \cup \{q\} \\
\textbf{until}\quad \text{no new state can be added to } Q_M & & \\
\textbf{if}\qquad Q_M \text{ contains a final state} & \textbf{then} & \text{return } \textit{not empty} \\
 & \textbf{else} & \text{return } \textit{empty}
\end{array}
$$

Proposition 5.7 *A state q is marked by the algorithm, that is it occurs in Q_M, if and only if there exists a term t such that $t \to q$.*

We may prove this claim using a proof similar to the one for regular tree automata.

We can also establish a result on the complexity of this algorithm. Let $Const_{\mathcal{A}}$ denote the maximal time required to decide the satisfiability of the constraints occurring in the type 3 rules of $\mathcal{A} = (Q, Q_{\text{fin}}, R)$.

Proposition 5.8 *The problem $L(\mathcal{A}) \stackrel{?}{=} \emptyset$ is decidable in time $O(|Q|.|R|.Const_{\mathcal{A}})$.*

The bound can be improved for regular tree automata, yielding a linear complexity. We could also get a linear bound if we have an oracle that, for each set of states $Q' \subseteq Q$ and each constraint, answer if the constraint restricted to Q' is satisfiable.

6 Results on the Tree Logic and on XML Schema

We prove our main property linking sheaves automata and the sheaves logic and use this result to derive several important properties of the simplified Schema language introduced in Section 2.

Theorem 6.1 (Definability) *For each formula Ψ of SL, we can construct a deterministic, complete, sheaves automaton $\mathcal{A}_\Psi$ accepting the models of Ψ.*

Proof By structural induction on the definition of Ψ. For each case, we describe the construction of the automaton $\mathcal{A}_\Psi$ and prove that $d \models \Psi$ if and only if $d \in \mathcal{L}(\mathcal{A}_\Psi)$.

Without loss of generality, we may strengthen the proposition with the following additional conditions:

(1) a state q occurring in the right-hand side of a constrained rule may not occur in the left-hand side of a constrained rule;

(2) a state occurring in the right-hand side of an unconstrained rule may not occur in the right-hand side of a constrained rule;

(3) Presburger's constraint may only occur when the right-hand side is not a final state, *i.e.* constrained rules are of the form $\mathit{True} \vdash \mathit{Reg}(Q) \rightarrow q$ whenever q is a final state.

We only consider the difficult cases. For the case $\Psi = \Psi \vee \Psi$ or $\neg\Psi'$, we simply use the fact that deterministic SA are closed under union and complement.

$\underline{\Psi = a[T].}$ Let $\mathcal{A}_T$ be the automaton constructed for T. Let q be a final state and q' be a state occurring in a rule $a[q] \rightarrow q'$ of $\mathcal{A}_T$. The idea is to choose the states of the form q' as the set of final states.

Let q be a final state occurring in a rule of the form $a[q] \rightarrow q'$. Whenever q' also occurs in a rule $c \rightarrow q'$ or $b[\ldots] \rightarrow q'$ of $\mathcal{A}_T$, we split q, q' in two states qa, qa' and $q\bar{a}, q\bar{a}'$ such that qa' occurs only in rules $a[qa] \rightarrow qa'$ and that $q\bar{a}'$ is used for the other rules, say $c \rightarrow q\bar{a}'$ or $b[\ldots] \rightarrow q\bar{a}'$. This is done for all such states q, q' of $\mathcal{A}_T$. The state-splitting is necessary to preserve determinism. The automaton $\mathcal{A}_\Psi$ is obtained by choosing the states qa' (where q is final in $\mathcal{A}_T$) as the set of final states.

A document d is accepted by $\mathcal{A}_\Psi$ if and only if there is a run of $\mathcal{A}_\Psi$ ending with a transition $a[qa] \rightarrow_{\mathcal{A}_\Psi} qa'$. Therefore d must be of the form $a[d']$ with $d' \rightarrow_{\mathcal{A}_\Psi} qa$. By construction, there is a transition $d' \rightarrow_{\mathcal{A}_T} q$ in $\mathcal{A}_T$ with q a final state in $\mathcal{A}_T$. By induction, a document d is accepted by $\mathcal{A}_T$ if and only if $d \models T$. Then d is accepted by $\mathcal{A}_\Psi$ if and only if $d \models \Psi$.

$\underline{\Psi = \texttt{Datatype}.}$ Let $\mathcal{A}_\Psi$ be the complete, deterministic automaton with states $\{q_\perp, q_{ok}\}$, set of final states $\{q_{ok}\}$, and with rules $\mathrm{cst} \rightarrow q_{ok}$ for all constant cst in `Datatype` and $\mathrm{cst} \rightarrow q_\perp$ otherwise.

By construction, $\mathcal{A}_\Psi$ accepts a document d if and only if $d = \mathrm{cst} \in \texttt{Datatype}$, that is only if $d \models \Psi$. We rely on the fact that the set of constant in `Datatype` is finite to obtain a finite automaton, but this condition could be simply relaxed by parameterizing the automaton with a set of constants. The case $\Psi = \mathrm{cst}$ is similar.

$\underline{\Psi = \mathit{AnyE}.}$ Let $\mathcal{A}_T$ be the complete, deterministic automaton with states $\{q_\perp, q_{ok}\}$, set of final states $\{q_{ok}\}$, and rules:

(type 1) $\mathrm{cst} \rightarrow q_\perp$ for all constant cst.

(type 2) $a[q_\perp] \rightarrow q_{ok}$ and $a[q_{ok}] \rightarrow q_{ok}$ for all tag name a.

(type 3) $(\#\, q_\perp + \#\, q_{ok} \geqslant 2) \vdash (q_\perp \mid q_{ok})* \rightarrow q_\perp$, a rule that match all documents made of more than one element.

By construction, $\mathcal{A}_\Psi$ accepts a document d if and only if there is a transition $d \to q_{ok}$. Therefore, since q_{ok} appears only as the result of a type 2 rules, it must be the case that $d = a[d']$ with $d' \to q_{ok}$ or $d' \to q_\perp$. Given that $\mathcal{A}_\Psi$ is complete, every document reach either q_{ok} or $q_\perp$. Therefore, d is accepted by $\mathcal{A}_\Psi$ if and only if $d = a[d']$, that is $d \models AnyE$. Like in the previous case, we rely on the fact that the set of constants and tag names are finite to obtain a finite automaton. Like in the previous case, this condition could be relaxed by parameterizing the automaton with a set of constants and a set of tag names.

$\underline{\Psi = S*, S \vee S, \bar{S} \textbf{ or } S, S'}$. The formula Ψ is a regular expression on some alphabet $E_1, \dots, E_p$, where E_i is an element formula. By induction, there is a deterministic automaton $\mathcal{A}_i$ accepting the models of E_i for all $i \in 1..p$. Let $\mathcal{A}$ be the product automaton of the $\mathcal{A}_i$'s. A state $\mathcal{Q}$ of $\mathcal{A}$ is of the form $(q_1, ..., q_p)$, with q_i a state of $\mathcal{A}_i$. Therefore $\mathcal{Q}$ may represent terms accepted by several $\mathcal{A}_i$'s. We use the notation $\mathcal{Q} \in \text{fin}(\mathcal{A}_i)$ to say that the i^{th} component of $\mathcal{Q}$ is a final state of $\mathcal{A}_i$.

We consider the regular expression Reg_Ψ, with alphabet the set of states of $\mathcal{A}$, obtained by syntactically replacing E_i in Ψ with the expression $\bigcup\{\mathcal{Q} \mid \mathcal{Q} \in \text{fin}(\mathcal{A}_i)\}$. The complement of Reg_Ψ is denoted $\bar{Reg}_\Psi$.

For every state $\mathcal{Q}$ and rule $\phi \vdash Reg \to \mathcal{Q}$ of $\mathcal{A}$, we split $\mathcal{Q}$ into two states, $\mathcal{Q}_\Psi$ and $\bar{\mathcal{Q}}_\Psi$, and the constrained rule into two rules $\phi \vdash Reg \cap Reg_\Psi \to \mathcal{Q}_\Psi$ and $\phi \vdash Reg \cap \bar{Reg}_\Psi \to \bar{\mathcal{Q}}_\Psi$. To conclude, we choose the states of the form $\mathcal{Q}_\Psi$ (where $\mathcal{Q}$ is final in $\mathcal{A}$) as the set of final states.

The automaton $\mathcal{A}_\Psi$ is deterministic and complete and the property follows by showing that $d \models \Psi$ if and only if $d \in \mathcal{L}(\mathcal{A}_\Psi)$.

We prove the first implication. Assume d is accepted by $\mathcal{A}_\Psi$, that is $d \to \mathcal{Q}_\Psi$, where $\mathcal{Q}$ is final in $\mathcal{A}$. By construction, d must be of the form $s_1 \cdot \ldots \cdot s_m$ and for all $i \in 1..m$ we have $s_i \to (q_1^i, \dots, q_p^i)$ with:

(1) at least one q_j^i is a final state for $\mathcal{A}_j$, that is s_j is accepted by at least one $\mathcal{A}_j$;

(2) the word $(q_1^1, \dots, q_p^1) \cdot \ldots \cdot (q_1^m, \dots, q_p^m)$ is in Reg_Ψ.

By (1) and our induction hypothesis, we have that there is a mapping, ς, from $1..m$ to the formulas E_1, ..., E_p such that $s_i \models E_{\varsigma(i)}$ for all $i \in 1..m$. With our notation, we have $(q_1^i, ..., q_p^i) \in \text{fin}(\mathcal{A}_{\varsigma(i)})$. Moreover, by (2) and the definition of Reg_Ψ, we have that (for at least one of such mapping ς) the word $E_{\varsigma(1)} \cdot \ldots \cdot E_{\varsigma(m)}$ is accepted by Ψ. Therefore $d \models \Psi$ as needed.

We prove the second implication. Assume $d \models \Psi$. Then d is of the form $s_1 \cdot \ldots \cdot s_m$ and there is a mapping, ς, from $1..m$ to the formulas E_1, ..., E_p, such that $s_i \models E_{\varsigma(i)}$ for all $i \in 1..m$ and such that $E_{\varsigma(1)} \cdot \ldots \cdot E_{\varsigma(m)}$ is accepted by Ψ. Let I_j be the set of indices l such that $s_j \models E_l$ (hence $s_j \not\models E_l$ if and only if $l \notin I_j$). By construction, we have that $s_j \to (q_1^j, \dots, q_p^j)$ where q_l^j is a final state of $\mathcal{A}_l$ for all indices $l \in I_j$. Moreover, the word $(q_1^1, \dots, q_p^1) \cdot \ldots \cdot (q_1^m, \dots, q_p^m)$ is accepted by Reg_Ψ (and therefore is not accepted by $\bar{Reg}_\Psi$.) By definition of $\mathcal{A}_\Psi$, there is a type 3 rule matching this case such that $d \to \mathcal{Q}_\Psi$ with $\mathcal{Q}_\Psi$ a final state of $\mathcal{A}_\Psi$. Therefore d is accepted by $\mathcal{A}_\Psi$ as needed.

$\Psi = \exists \mathbf{N} : \phi : N_1 E_1 \ \& \ \ldots \ \& \ N_p E_p$. By induction, there is a deterministic automaton $\mathcal{A}_i$ accepting the models of E_i for all $i \in 1..p$. The construction is similar to a determinisation process. Let $\mathcal{A}$ be the product automaton of the $\mathcal{A}_i$'s and let $\{\mathcal{Q}_1, ..., \mathcal{Q}_m\}$ be the states of $\mathcal{A}$. A state $\mathcal{Q}$ of $\mathcal{A}$ is of the form $(q_1, ..., q_p)$, with q_i a state of $\mathcal{A}_i$, and it may therefore represent terms accepted by several $\mathcal{A}_i$'s. We use the notation $\mathcal{Q} \in \mathrm{fin}(\mathcal{A}_i)$ to say that the i^{th} component of $\mathcal{Q}$ is a final state of $\mathcal{A}_i$.

The constrained rules of $\mathcal{A}$ are of the form $\psi(M_1, \ldots, M_m) \vdash \mathit{Reg} \rightarrow \mathcal{Q}$, where M_i stands for the number of occurrences of the state $\mathcal{Q}_i$ in a run. The idea is to define a Presburger's formula, $\phi^{\exists}(M_1, \ldots, M_m)$, satisfied by configurations $\mathcal{Q}_{j_1} \cdot \ldots \cdot \mathcal{Q}_{j_n}$ containing a number of final states of the $\mathcal{A}_i$'s satisfying ϕ, and to augment all the type 3 rules with this counting constraint. To define $\phi^{\exists}$, we decompose M_i into a sum of integer variables, x_j^i for $j \in 1..p$, with x_j^i corresponding to a number of final states of $\mathcal{A}_j$ occurring in $\mathcal{Q}_i$.

$$\phi^{\exists} \ =_{\mathrm{def}} \quad \exists (x_j^i)_{\substack{i \in 1..m \\ j \in 1..p}} \cdot \bigwedge_{i \in 1..m} (M_i = \sum_{\substack{j \in 1..p \\ \mathcal{Q}_i \in \mathrm{fin}(\mathcal{A}_j)}} x_j^i \ ) \ \wedge \ \phi(\sum_{\substack{i \in 1..m \\ \mathcal{Q}_i \in \mathrm{fin}(\mathcal{A}_1)}} x_1^i, \ \ldots \ , \ \sum_{\substack{i \in 1..m \\ \mathcal{Q}_i \in \mathrm{fin}(\mathcal{A}_p)}} x_p^i \ )$$

Finally, we split each constrained rule $\psi \vdash \mathit{Reg} \rightarrow \mathcal{Q}$ of $\mathcal{A}$ into the two rules $\psi \wedge \phi^{\exists} \vdash \mathit{Reg} \rightarrow \mathcal{Q}_\Psi$ and $\psi \wedge \neg\phi^{\exists} \vdash \mathit{Reg} \rightarrow \bar{\mathcal{Q}}_\Psi$, splitting also the state $\mathcal{Q}$ into $\mathcal{Q}_\Psi$ and $\bar{\mathcal{Q}}_\Psi$. The automaton $\mathcal{A}_\Psi$ is obtained by choosing the states of the form $\mathcal{Q}_\Psi$ (where $\mathcal{Q}$ is final in $\mathcal{A}$) as the set of final states.

The automaton $\mathcal{A}_\Psi$ is deterministic and complete and the property follows by showing that $d \models \Psi$ if and only if $d \in \mathcal{L}(\mathcal{A}_\Psi)$.

We prove the first implication. Assume d is accepted by $\mathcal{A}_\Psi$, that is $d \rightarrow \mathcal{Q}_\Psi$, where $\mathcal{Q}$ is final in $\mathcal{A}$. By construction, d must be of the form $s_1 \cdot \ldots \cdot s_m$ and for all $i \in 1..m$ we have $s_i \rightarrow (q_1^i, \ldots, q_p^i)$ and q_l^i is final in $\mathcal{A}_l$ if and only if $s_i \models E_l$. Since we have an accepting run, it must be the case that the counting constraint $\phi^{\exists}$ is satisfied. Therefore, we can find a decomposition of the s_i's in n_1 documents satisfying E_1, ..., n_p documents satisfying E_p, such that $\models \phi(n_1, ..., n_p)$. For example, we can choose $n_j = \sum\{x_j^i \mid i \in 1..m, \mathcal{Q}_i \in \mathrm{fin}(\mathcal{A}_j)\}$, where the x_j^i's are values obtained from solving the constraint $\phi^{\exists}$. Therefore $d \models \Psi$ as needed.

We prove the second implication. Assume $d \models \Psi$. Then d is of the form $s_1 \cdot \ldots \cdot s_m$ and there is a partition of the s_i's in p distinct groups, of respective size $n_1, ..., n_p$, such that, for all $i \in 1..p$, the documents in the i^{th} group satisfies E_i and that $\models \phi(n_1, ..., n_p)$. By induction hypothesis, we have for each $i \in 1..m$ that $s_i \rightarrow_{\mathcal{A}} (q_1^i, ..., q_p^i)$, where $\mathcal{A}$ is the product of the automaton E_1, ..., E_p and where q_l^i is final in $\mathcal{A}_l$ if and only if $s_i \models E_l$. For all $i \in 1..m$ and $j \in 1..p$, take $x_j^i = 1$ if s_i is in the partition of the sub-documents satisfying E_j and $x_j^i = 0$ otherwise. These values for the x_j^i's provide a solution to the counting constraint $\phi^{\exists}$. Therefore, there is a transition $d \rightarrow \mathcal{Q}_\Psi$ in Ψ with $\mathcal{Q}_\Psi$ a final state, that is, d is accepted by $\mathcal{A}_\Psi$ as needed. ∎

As a direct corollary of Th. 6.1 and Propositions 5.6 and 5.8, we obtain key results on the decidability and on the complexity of the sheaves logic. Let $|Q(\mathcal{A}_\Psi)|$ be the number of states of the SA associated to Ψ.

Theorem 6.2 (Decidability) *The logic SL is decidable.*

Theorem 6.3 (Model Checking) *For any document d and formula Ψ, the model checking problem $d \models \psi$ is decidable in time $O(|d|.|\mathcal{A}_\psi|.Const(|Q(A_\Psi)|, |d|)|$.*

Since the schema language is a plain subset of our tree logic, we can directly transfer these results to schemas and decide the relation $d : S$ using sheaves automata.

Proposition 6.4 *For every schema, S, there is a deterministic sheaves automaton, $\mathcal{A}$, such that $L(\mathcal{A}) = \{d \mid d : S\}$.*

Proof Similar to the proof of Th. 6.1. The only difference is that we must replace the choice operator, $a[T]?$, which does not occur in SL, by the formula $(\epsilon \vee a[T])$. ■

We have a similar but weaker results for recursive schemata.

Proposition 6.5 *For every recursive schema, S, there is a (non-necessarily deterministic) sheaves automaton such that $L(\mathcal{A}) = \{d \mid d : S\}$.*

Proof Similar to the proof of Proposition 6.4. For recursive schemas, we need to introduce a special state $q_\mathcal{X}$ for each definition $\mathcal{X} = T$ occurring in S. Then we construct the automata corresponding to T and replace $q_\mathcal{X}$ in $\mathcal{A}_S$ by any final state of $\mathcal{A}_T$. ■

Combined with our previous results, we obtain several decidability properties on schemas. Most importantly, we obtained an automata-based decision procedure for all these problems. We can, for example, easily define the intersection and difference of two schemas (that are not necessarily well-formed schemas.)

Theorem 6.6 (XML Typing) *Given a document, d, and a schema, S, the problem $d : S$ is decidable.*

Theorem 6.7 (Satisfaction) *Given a schema S, the problem $\exists d.d : S$ is decidable.*

7 Conclusion

Our contribution is a new class of automaton for unranked tree aiming at the manipulation of XML schemas. We believe it is the first work on applying tree automata theory to XML that consider the &-operator. This addition is significant in that interleaving is the source of many complications, mainly because it involves the combination of ordered and unordered data models. This led us to extend hedge automaton [4] with counting constraints as a way to express properties on both sequences and multisets of elements. This extension appears quite natural since, when no counting constraints occurs, we obtain *hedge automata* and, when no constraints occur, we obtain regular tree automata.

The interleaving operator has been the subject of many controversial debates among the XML community, mainly because a similar operator was responsible for difficult implementation problems in SGML. Our work gives some justifications for these difficulties, most particularly the undecidability of computing the complement of non-deterministic languages. To elude this problem, and in order to limit ourselves to deterministic automaton, we have introduced two separate sorts for regular and counting formulas in our logic. It is interesting to observe that a stronger restriction appears in the schema specification [1], namely that the &-operator may only appears at top-level position in an element definition.

Another source of problems is linked to the size and complexity of counting constraints. Again, we may find syntactical restrictions to avoid this problem. For example, we may obtain polynomial complexity by imposing that each element tags in an expression $a_1[S_1]$ & $\ldots$ & $a_p[S_p]$ be distinct. And again, a similar restriction may be found in the XML schema specification.

Our goal is not to devise a new schema or pattern language for XML, but rather to find an implementation framework compatible with schemas. An advantage of using tree automata theory for this task is that it also gives us complexity results on problems related to XML schema (and to possible extensions of schemas with logical operators.) As indicated by our previous remarks, we may also hope to use our approach to define improved restrictions on schema and to give a better intuition on their impact. Another advantage of using tree automata is that it suggests multiple directions for improvements. For example, a most desirable extension to our model is to add the capacity for the reverse traversal of a document. This could be achieved using some form of backtracking, like a parallel or alternating [13] variant of our tree automata, or by considering tree grammars (or equivalently top-down tree automata.) The same extension is needed if we want to process tree-structured data in a streamed way, a situation for which bottom-up tree automata are not well-suited.

References

[1] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn, editors. *XML Schema Part 1: Structures.* W3C (World Wide Web Consortium), 2001. Available at `http://www.w3.org/TR/xmlschema-1/`.

[2] S. Abiteboul and P. Buneman. *Data on the Web : From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[3] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *Proc. of Principles of Programming Languages (POPL)*, pages 67–80. ACM Press, 2001.

[4] Murata Makoto. Extended path expression for XML. In *Proc. of Symposium on Principles of Database Systems (PODS).* ACM Press, 2001.

[5] James Clark and Murata Makoto, editors. *RELAX NG Tutorial.* OASIS, 2001. Available at `http://www.oasis-open.org/committees/relax-ng/tutorial.html`.

[6] Alexandru Berlea and Helmut Seidl. Binary queries. In *Extreme Markup Languages*, 2002.

[7] F. Neven and T. Schwentick. Automata- and logic-based pattern languages for tree-structured data. (manuscript), 2001.

[8] L. Cardelli and A. Gordon. Anytime, anywhere: Modal logic for mobile ambients. In *Proc. of Principles of Programming Languages (POPL).* ACM Press, January 2000.

[9] L. Cardelli and G. Ghelli. A query language based on the ambient logic. In *Proc. of ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 2001.

[10] Denis Lugiez and Silvano Dal Zilio. Multitrees Automata, Presburger's Constraints and Tree Logics. Research report 08-2002, LIF, Marseille, France, June 2002. Available at `http://www.lim.univ-mrs.fr/LIF/Rapports/08-2002-Lugiez-DalZilio.html`.

[11] A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL: A model for W3C XML schema. In WWW 10, May 2001.

[12] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM-TOPLAS*, 15(4):575–631, 1993.

[13] H. Comon, M. Dauchet, F. Jacquemard, S. Tison D. Lugiez, and M. Tommasi. *Tree Automata and their application.* (to appear as a book), 1999. Available at `http://www.grappa.univ-lille3.fr/tata/`.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

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