XML Schema, Tree Logic and Sheaves Automata

Silvano Dal Zilio* and Denis Lugiez

Laboratoire d'Informatique Fondamentale de Marseille
CNRS (UMR 6166) and Université de Provence

Abstract. XML documents, and other forms of semi-structured data, may be roughly described as edge labeled trees; it is therefore natural to use tree automata to reason on them. This idea has already been successfully applied in the context of *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. In this paper, we define a tree logic that directly embeds XML Schema as a plain subset as well as a new class of automata for unranked trees, used to decide this logic, which is well-suited to the processing of XML documents and schemas.

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. XML documents, and other forms of semi-structured data [1], 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.* [9], a typed functional language with extended pattern-matching operators for XML documents manipulation. In this tool, the types of XML documents 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 [11], 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 [6], an alternative proposal to XML Schema. Various extension of tree automata [2] and monadic tree logic 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

* work partially supported by ATIP CNRS "Fondements de l'Interrogation des Donnes Semi-Structures" and by IST Global Computing PROFUNDIS.

Robert Nieuwenhuis (Ed.): RTA 2003, LNCS 2706, pp. 246–263, 2003.

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 almost 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 order of occurrence. 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 &-operator. By restricting our study to deterministic automata, 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. We start by defining the syntax of XML documents and XML schema. A distinctive aspect of our simplified schema language is to include the operator &. In Sect. 4, we present a tree logic intended for querying XML documents. This logic can be interpreted as a direct extension of the schema language with logical operators. The logic deliberately resembles (and extends on some points) TQL, a query language for semi-structured data based on the *ambient logic* [5, 4]. We present a similar logic, with the difference that we deal both with ordered and unordered data structures, while TQL only deals with multisets of elements. Another difference with TQL lies in the addition of arithmetical constraints. In this extended logic, it becomes for instance possible to express constraints on the number of occurrences of an element, 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 is purely motivated by the presence of &, it incidentally provides a model for *cardinality constraint on repetitions*, $e\{m, n\}$, that matches k repetitions of the expression e, with $m \leqslant k \leqslant n$.

In Sect. 5, we introduce a new class of automata for unranked trees, called *Sheaves Automata* (SA), that is used to decide our extended tree logic. In the transition relation of SA, we combine the general rules for regular tree automata with regular word expression and counting constraints. In this framework, regu-

lar word expressions allow us to express constraints on sequences of elements and are used when dealing with sequential composition of documents, as in the *hedge automata* approach. Correspondingly, the counting constraints are used with the &-operator. The counting constraints are *Presburger arithmetic* formulas on 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 numerical.

The choice of Presburger arithmetic is not exclusively motivated by the fact that it is a large class of constraints over natural numbers, which increases the expressiveness of our logic while still remaining decidable. Indeed, we prove that Presburger constraints arises naturally when we consider schemas that combine interleaving, &, and recursive definitions (see Sect. 3). Another reason is that this extension preserves many enjoyable properties of regular tree automata: the class of languages recognized by sheaves automata is closed under union and intersection, testing for emptiness is decidable, ..., while adding some new ones, like the fact that recognizable languages are closed by composition of sequential and commutative operators. Even so, the gain in expressiveness is significant as such. Indeed, Muscholl, Schwentick and Seidl have very recently proposed a new and independent class of automaton very close to our model for the sole purpose of making numerical queries on XML documents [12].

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: typing a document by a schema, computing the set of documents typed by a schema, computing the set of documents typed by the difference of two schemas ... In proving these results, we also make clear how simple syntactical restrictions on schemas improve the complexity of simple operations.

Omitted proofs may be found in a long version of this paper [8].

2 Documents and Schemata

XML documents are a simple textual representation for unranked, edge labeled trees. In this report, we follow the notations found in the XDuce system [9] 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 [3], an attempt to formalize some of the core ideas found in XML Schema.

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.

Elements and Documents

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

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

$$book[auth[\text{"Knuth" }] \cdot title[\text{"Art of Computer Programming" }] \cdot year[1970]]$$

A schema may be interpreted as the type of a document. Our definition mostly follows the presentation made in MSL [3]. 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
$S_1 \cdot S_2$	sequential composition
$S + S$	choice
$S*$	indefinite repetition
$T ::=$	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 contain a single top element tagged with a and enclosing a subdocument satisfying the schema T. An optional element, $a[T]?$, matches one or zero occurrence of $a[T]$. The constructors for schemata include the standard operators found in regular expression languages, where $S \cdot S'$ stands for concatenation and $S + S'$ 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 *Any Type* in XML Schema terminology.

Example 2. Assuming that `String` and `Year` are the types associated to string and date constants, the following schema matches the book entry given in Example 1: $(book[title[\texttt{String}] \,\&\, auth[\texttt{String}] \,\&\, year[\texttt{Year}]?)$.

The distinction of a top-level schema allows us to express some of the constraints on the interleaving operator found in the XML specification. For example, & must appear as the sole child at the top of an element schema, that is, terms like $E_1 \cdot (E_2 \,\&\, E_3)$ or $(E_1 \,\&\, E_2)*$ are ill-formed.

To capture some situations arising in practice, we may enrich schemata by recursive definitions presented by a system of equations. This can be simply obtained by enriching the syntax with variables, $X, Y, \ldots$, and an operator for recursive schema definition, $(S \textbf{ where } X_1 = S_1, \ldots, X_n = S_n)$, where the X_i's are bound variable names.

Example 3. We may extend book entries with a *ref* element listing the entries cited in the book:

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

Next, we make explicit the role of schemas as a type system for documents and define the relation $d : S$, meaning that the document d satisfies the schema S. This relation is based on an auxiliary function, $inter(d)$, which computes the *interleaving* of the elements in d, that is the set of documents obtainable from d after permutation of its elements: $inter(e_1 \cdot \ldots \cdot e_n) = \{e_{\sigma(1)} \cdot \ldots \cdot e_{\sigma(n)} \mid \sigma \text{ permutation of } 1..n\}$.

In the long version of this paper [8], we define a more complex relation, $X_1 : S_1, \ldots, X_n : S_n \vdash d : S$, to type documents using recursive schemas.

Good Documents

$$\frac{d : T}{a[d] : a[T]} \qquad \frac{d : T}{a[d] : a[T]?} \qquad \frac{}{\epsilon : a[T]?} \qquad \frac{\text{cst} \in \texttt{Datatype}}{\text{cst} : \texttt{Datatype}} \qquad \frac{}{\epsilon : \epsilon} \qquad \frac{d_1 : S_1 \quad d_2 : S_2}{d_1 \cdot d_2 : S_1 \cdot S_2}$$

$$\frac{d : S}{d : S + S'} \qquad \frac{d : S'}{d : S + S'} \qquad \frac{d_1 : S, \ldots, d_n : S}{d_1 \cdot \ldots \cdot d_n : S*} \qquad \frac{}{d : AnyT} \qquad \frac{d \in inter(e_1 \cdot \ldots \cdot e_n) \quad e_1 : E_1 \quad \ldots \quad e_n : E_n}{d : E_1 \,\&\, \ldots \,\&\, E_n}$$

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 Arithmetic and Words

Some computational aspects of our tree automaton rely on arithmetical properties over the group $(\mathbb{N}, +)$ of natural numbers with addition. The first-order theory of equality on this structure, also known as Presburger arithmetic, is decidable. Formulas of Presburger arithmetic, also called *Presburger constraint*, are given by the following grammar. We use $N, M, \ldots$ to range over integer variables and $n, m, \ldots$ to range over integer values.

Presburger Constraint

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

Presburger constraints allow us to define flexible, yet decidable, properties over positive integers 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 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.

Decidability of Presburger arithmetic may be proved using a connection with *semilinear sets* of natural numbers. A *linear set* of $\mathbb{N}^n$, $L(\boldsymbol{b}, P)$, is a set of vectors generated by linear combination of the periods $P = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_k\}$ (where $p_i \in \mathbb{N}^n$ for all $i \in 1..k$), with the base $\boldsymbol{b} \in \mathbb{N}^n$, that is, $L(\boldsymbol{b}, P) =_{\text{def}} \left\{\boldsymbol{b} + \sum_{i \in 1..k} \lambda_i \boldsymbol{p}_i \mid \lambda_1, \ldots, \lambda_k \in \mathbb{N}\right\}$. A *semilinear set* is a finite union of linear sets and the models of Presburger formulas (with p free variables) are semilinear sets of $\mathbb{N}^p$. An important result is that semilinear sets are closed under *union*, *sum* and *iteration*, 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).

3.1 Parikh mapping

Another mathematical tool needed in the presentation of our new class of automaton is the notion of *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 $\Sigma*$ 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 use the notation $\#_a(w)$ for the number of occurrences of a in w, or simply $\# a$ when there is no ambiguity.

The *Parikh mapping* of a set of words is the set of Parikh mappings of its elements. When the set of words is a regular language, the Parikh mapping can be easily computed and it corresponds to the model of a Presburger formula. Furthermore, when the regular language is described by a regular expression, *reg*, we can compute the Parikh mapping in time $O(|reg|)$ (using regular expressions of semilinear sets). For example, if a_i is the i^{th} letter in Σ, then $\#(a_i)$ is the linear set $L(\boldsymbol{u}_i, \emptyset)$, where $\boldsymbol{u}_i$ is the i^{th} unit vector of $\mathbb{N}^n$, and the mapping of a sequential composition expression, $reg_1 \cdot reg_2$, is the linear set $\#(reg_1) + \#(reg_2)$.

Proposition 1. *The Parikh mapping of a regular language is a semilinear set.*

This property is useful when we consider the intersection of a regular word language with a set of words whose Parikh mapping satisfies a given Presburger constraint. This is the case in Sect. 4, for example, when we test the emptiness of the language accepted by a Sheaves automaton.

3.2 Relation with XML Schema

We clarify the relation between Presburger 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 schemas.

Let $a_1, \ldots, a_p$ be distinct element tags and d be a "flat document", i.e. of the form $a_{i_1}[\epsilon] \cdot \ldots \cdot a_{i_k}[\epsilon]$, then $\#(.)$ provides a straightforward mapping from d to $\mathbb{N}^p$. Suppose now that we slightly relax the syntactic constraints on schemas in order to accept expressions of the form $((E_1 \,\&\, \cdots \,\&\, E_n) + E)$ and $(E_1 \,\&\, \cdots \,\&\, E_n \,\&\, X)$. Then, for any Presburger constraint, ϕ, it is possible to define an (extended recursive) schema that matches the vectors of integers satisfying ϕ. For example, the schema X **where** $X = ((a_1 \,\&\, a_2 \,\&\, X) + \epsilon)$ is associated to the formula $\#\, a_1 = \#\, a_2$ (there are as many a_1's than a_2's) and X **where** $X = ((a_1 \,\&\, a_1 \,\&\, X) + \epsilon)$ is associated to $\exists N. \#\, a_1 = N + N$ (there is an even number of a_1's).

Proposition 2. *For every Presburger formula ϕ, there is a schema, S, such that $d : S$ iff $\models \phi\, \#(d)$.*

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

In the next section, we introduce a modal logic for documents that directly embeds counting constraint. Indeed, Proposition 2 indicates that it is necessary to take into account Presburger constraints when dealing with the &-operator. Moreover, aside from the fact that counting constraints add expressiveness to our logic, another reason for adding Presburger formulas is that we extend the set of recognizable trees while still preserving good (and decidable) closure properties.

4 The Sheaves Logic

We extend our simplified version of XML Schema 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 Ghelli [4], 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
$\quad a[S]$	element with tag a and regular formula S
$\quad a[T]$	element with tag a and counting formula T
$\quad AnyE$	any element
$\quad$ `Datatype`	datatype constant
$S ::=$	Regular formula
$\quad \epsilon$	empty
$\quad E$	element
$\quad S \cdot S'$	sequential composition
$\quad S*$	indefinite repetition
$\quad S \vee S$	choice
$\quad \neg S$	negation
$T ::=$	Counting formula
$\quad \exists \boldsymbol{N} : \phi(\boldsymbol{N}) : N_1 E_1 \ \& \ \ldots \ \& \ N_p E_p$	generalized interleaving, $\boldsymbol{N} = (N_1, \ldots, N_p)$
$\quad T \vee T$	choice
$\quad \neg T$	negation
$A, B, \ldots ::= \quad S \mid T \mid A \vee A \mid \neg A$	Sheaves Logic Formula

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

The generalized interleaving operator is inspired by the relation between schema and counting constraint given in Sect. 2. This operator is useful to express more liberal properties on documents than with Schemas. For example, it is now possible to define the type (an example of ill-formed schemas) $E_1* \ \& \ E_2$, 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 $AnyE$ formula matches documents made of a single element. It has been chosen instead of the

less general schema $AnyT$ since it could be used in a constrained existential quantification. It is possible to model $AnyT$ using the formulas $\exists N : (N \geqslant 0) : NAnyE$ and $AnyE*$.

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 rules shared 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	$(d = \text{cst}) \ \wedge \ (\text{cst} \in \texttt{Datatype})$
$d \models \epsilon$	iff	$d = \epsilon$
$d \models S \cdot S'$	iff	$(d = d_1 \cdot d_2) \ \wedge \ (d_1 \models S) \ \wedge \ (d_2 \models S')$
$d \models S*$	iff	$(d = \epsilon) \ \vee$ $((d = d_1 \cdot \ldots \cdot d_p) \ \wedge \ (\forall i \in 1..p, d_i \models S))$
$d \models \exists \boldsymbol{N} : \phi(\boldsymbol{N}) : N_1E_1 \ \& \ \ldots \ \& \ N_pE_p$	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 $(d \models \Psi)$

Example of Formulas. We start by defining some syntactic sugar. The formula *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, ..., E_p, regardless of their order.

$$(E_1 \ \& \ \ldots \ \& \ E_p) \ =_{\text{def}} \ \exists N_1, ..., N_p : (N_1 = ... = N_p = 1) : N_1E_1 \ \& \ \ldots \ \& \ N_pE_p$$

Likewise, we define the notation $(a[S] \ \& \ \cdots)$ for the formula satisfied by documents containing at least one element matching $a[S]$:

$$(a[S] \ \& \ \cdots) \ =_{\text{def}} \ \exists M, N : (M = 1) \wedge (N \geqslant 0) : Ma[S] \ \& \ NAnyE$$

For a more complex example, let us assume that a book reference is given by the schema in Example 2. 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 that contain at least 50 entries written by Knuth or Lamport.

$$\exists N, M, X : (N + M = 50 + X) : \begin{pmatrix} N\,book[(auth[\text{"Knuth"}] \ \& \ \cdots)] \\ \& \ M\,book[(auth[\text{"Lamport"}] \ \& \ \cdots)] \end{pmatrix}$$

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 on XML schemata. A main distinction with other automata-theoretic approaches, like *hedge automata* [11] for example, is that we do not focus on regular expressions over paths but, instead, concentrate on the &-operator, which is one of the chief additions of XML Schema with respect to DTD. The definitions presented here have been trimmed down for the sake of brevity. 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.

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:

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

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, also termed *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 $\{q_1, \ldots, q_p\}$ and $\phi(N_1, \ldots, N_p)$ is a Presburger arithmetic formula with 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) rule may fire if we have a term of the form $d_1 \cdot \ldots \cdot d_n$ such that:

- each term d_i leads to a state $q_{j_i} \in Q_{\mathcal{A}}$;
- 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, that is, $\models \phi(n_1, \ldots, n_p)$, where n_i is the number of occurrences of q_i in $q_{j_1} \cdot \ldots \cdot q_{j_n}$.

To stress the connection between variables in the counting constraint ϕ and the number of occurrences of q_i matched by $\mathit{Reg}(Q_{\mathcal{A}})$, we will use $\#\,q_i$ instead of N_i as the name of integer variables.

Example 4. An example of automaton on the signature $\{c, a[\_], b[\_]\}$ is given by the set of states $Q_{\mathcal{A}} = \{q_a, q_b, q_s\}$, the set of final states $Q_{\text{fin}} = \{q_s\}$ and the following set of five transition rules:

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

We show in Example 5, after defining the transition relation, that this particular automaton accepts terms with as many a's than b's at each node, like for example $b[\epsilon] \cdot a[c \cdot b[\epsilon] \cdot c \cdot a[\epsilon]]$.

If we drop the Presburger 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* [11]. Conversely, if we drop the regular word expression and restrict to rules of the form $\phi(\#\, q_1, \ldots, \#\, q_p) \vdash (q_1 + \ldots + q_p)* \to q$, 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).

Transition Relation. The *transition relation* of an automaton $\mathcal{A}$, denoted $d \to_{\mathcal{A}} q$, or simply $\to$ when there is no ambiguity, is the transitive closure of the relation defined by the following three rules.

Transition Relation: $\to$

$$\text{(type 1)}\quad \frac{c \to q \in R}{c \to q} \qquad \text{(type 2)}\quad \frac{d \to q' \quad n[q'] \to q \in R}{n[d] \to q}$$

$$\text{(type 3)}\quad \frac{\begin{array}{c} e_1 \to q_{j_1} \quad \ldots \quad e_n \to q_{j_n} \\ q_{j_1} \cdot \ldots \cdot q_{j_n} \in \mathit{Reg} \quad \models \phi\,\#(q_{j_1} \cdot \ldots \cdot q_{j_n}) \\ \phi \vdash \mathit{Reg} \to q \in R \qquad (n \geqslant 2) \end{array}}{e_1 \cdot \ldots \cdot e_n \to q}$$

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 of only one element. 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:

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

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 $(q_a + q_b + q_s)*$ and that it contains as many q_a's than q_b's (its Parikh mapping is $(1, 1, 0)$).

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. In this example, we can recognize trees in which every sequences contains as many a's than b's as top elements. Indeed, 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 constraint $(\#\, q_a = \#\, q_b) \wedge (\#\, q_b = \#\, q_c)$ in Example 4).

As it is usual with automata, we say that a document d is *accepted* by a sheaves automaton $\mathcal{A}$ if there is a final state $q \in Q_{\text{fin}}$ such that $d \rightarrow_{\mathcal{A}} q$. The language $\mathcal{L}(\mathcal{A})$ is the set of terms accepted by $\mathcal{A}$. In the following, we will only consider *complete automaton*, such that every term reaches some state. This can be done without loss of generality since, for any SA, $\mathcal{A}$, it is always possible to build an equivalent complete automaton, $\mathcal{A}_c$ [8].

Proposition 3. *For any SA, $\mathcal{A}$, we can construct a complete automaton, $\mathcal{A}_c$, that accepts the language $\mathcal{L}(\mathcal{A})$ and it is deterministic if $\mathcal{A}$ is deterministic.*

Next, we enumerate of list of properties for our new class of automaton.

Deterministic SA are less Powerful than Non-deterministic SA. A sheaves automaton is *deterministic* if and only if a term reaches at most one state. Contrary to regular tree automata, the class of deterministic sheaves automata is strictly weaker than the class of sheaves automata. In order to preserve determinism as much as possible, we will choose constructions for basic operations on automata that are a little bit more complex than the usual ones.

Proposition 4. *There is a language accepted by a sheaves automaton that can not be accepted by any deterministic sheaves automaton.*

Proof. Using an improved "pumping lemma" [8], we prove that the language L, consisting of the terms $a^n \cdot b^n \cdot a^m \cdot b^m$, with $n, m > 0$, is not recognizable by a deterministic SA, although there is a non-deterministic SA accepting L. □

Product, Union and Intersection. Given two automata $\mathcal{A} = \langle Q, Q_{\text{fin}}, R\rangle$ and $\mathcal{A}' = \langle Q', Q'_{\text{fin}}, R'\rangle$, 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 $\mathcal{A}^\times = \langle Q^\times, \emptyset, R^\times\rangle$ such that:

- $Q^\times = Q \times Q' = \{(q_1, q'_1), \ldots, (q_p, q'_l)\}$,
- for every type (1) rules $a \rightarrow q \in R$ and $a \rightarrow q' \in R'$, the rule $a \rightarrow (q, q')$ is in $R^\times$,
- for every type (2) rules $n[q] \rightarrow s \in R$ and $n[q'] \rightarrow s' \in R'$, the rule $n[(q, q')] \rightarrow (s, s')$ is in $R^\times$,
- for every type (3) rules $\phi \vdash \mathit{Reg} \rightarrow q \in R$ and $\phi' \vdash \mathit{Reg}' \rightarrow q' \in R$, the rule $\phi^\times \vdash \mathit{Reg}^\times \rightarrow (q, q')$ 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 formulas ϕ and ϕ' obtained as follows. Let $\#(q, q')$ be the name of the variable associated to the numbers of occurrences of the state (q, q'), then:

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

Proposition 5. *We have $d \to (q, q')$ in the automaton $\mathcal{A} \times \mathcal{A}'$, if and only if both $d \to_{\mathcal{A}} q$ and $d \to_{\mathcal{A}'} q'$.*

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 $\mathcal{A} \cap \mathcal{A}'$ and the union $\mathcal{A} \cup \mathcal{A}'$ may be simply obtained from the product $\mathcal{A} \times \mathcal{A}'$ by setting the set of final states to:

$$Q^{\cap}_{\text{fin}} =_{\text{def}} \left\{(q, q') \mid q \in Q_{\text{fin}} \ \wedge \ q \in Q'_{\text{fin}}\right\}$$
$$Q^{\cup}_{\text{fin}} =_{\text{def}} \left\{(q, q') \mid q \in Q_{\text{fin}} \ \vee \ q \in Q'_{\text{fin}}\right\}$$

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 6. *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.*

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 7. *Non-deterministic Sheaves languages are not closed under complementation.*

Proof. We prove in [8] that given a two-counter machine, there is a non-deterministic automaton accepting exactly the bad computations of the machine. Thus, 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 two-counter machine halts. □

Membership. 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 *Cost* such that, for all constraints ϕ, the evaluation of $\phi(n_1, \ldots, n_p)$ can be done in time $O(\mathit{Cost}(p, n))$ whenever $n_i \leqslant n$ for all i in $1..p$. For quantifier-free Presburger 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 8. *For an automaton $\mathcal{A} = \langle Q, Q_{fin}, R\rangle$, the problem $d \stackrel{?}{\in} \mathcal{L}(\mathcal{A})$ can be decided in time $O(|d| \cdot |R| \cdot Cost(|Q|, |d|))$ for a deterministic automaton and in time $O(|d|^2 \cdot |Q| \cdot |R| \cdot Cost(|Q|, |d|)))$ for a non-deterministic automaton.*

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

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 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 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 $Reg_{|Q'}$ that corresponds to the words on the alphabet Q' satisfying *Reg*. This expression can be easily obtained from *Reg* by a set of simple syntactical rewritings. Then we compute the Parikh mapping $\#(Reg_{|Q'})$ as explained in Sect. 3 and test the satisfiability of the Presburger formula:

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

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 marking 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.

Algorithm 1. Test for Emptiness

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

Proposition 9. *A state q is marked by Algorithm 1, that is $q \in Q_M$, iff there exists a term t such that $t \to q$.*

We may prove this claim using a reasoning similar to the one for regular tree automata. We can also establish a result on the complexity of this algorithm. Let $Cost_{\mathcal{A}}$ 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 10. *The problem* $L(\mathcal{A}) \stackrel{?}{=} \emptyset$ *is decidable in time* $O(|Q|\cdot|R|\cdot Cost_{\mathcal{A}})$.

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, tells whether 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 Sect. 2.

Theorem 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. 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 constraint may only occur when the right-hand side is not a final state, *i.e.* constrained rules are of the form $True \vdash Reg(Q) \to q$ 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.

- $\Psi = a[T]$. 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 $a[q] \to q'$ 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 $a[q] \to q'$. Whenever q' also occurs in a rule $c \to q'$ or $b[\ldots] \to q'$ 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 $a[qa] \to qa'$ and that $q\bar{a}'$ is used for the other rules, say $c \to q\bar{a}'$ or $b[\ldots] \to q\bar{a}'$. This is done for all such states q, q' of $\mathcal{A}$. 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.
- $\Psi = S*, S \vee S, \bar{S}$ **or** S, S'. In this case, Ψ is a regular expression on some alphabet $E_1, \ldots, E_n$ where E_i are element formulas. 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 $\mathcal{Q} \in \mathrm{fin}(\mathcal{A}_i)$ to say that the i^{th} component of $\mathcal{Q}$ is a final state of $\mathcal{A}_i$.
  We consider the regular expression Reg_S, with alphabet the set of states of $\mathcal{A}$, obtained by syntactically replacing E_i in Ψ with the expression $\bigcup\{\mathcal{Q} \mid \mathcal{Q} \in \mathrm{fin}(\mathcal{A}_i)\}$. The complement of Reg_S is denoted $\bar{Reg}_S$.

For every state $\mathcal{Q}$ and rule $\phi \vdash \mathit{Reg} \to \mathcal{Q}$ of $\mathcal{A}$, we split $\mathcal{Q}$ into two states, $\mathcal{Q}_S$ and $\bar{\mathcal{Q}}_S$, and the constrained rule into two rules $\phi \vdash \mathit{Reg} \cap \mathit{Reg}_S \to \mathcal{Q}_S$ and $\phi \vdash \mathit{Reg} \cap \bar{\mathit{Reg}}_S \to \bar{\mathcal{Q}}_S$. To conclude, we choose the states of the form $\mathcal{Q}_S$ (where $\mathcal{Q}$ is final in $\mathcal{A}$) as the set of final states. This automaton is deterministic and complete and the property follows by showing that $d \models \Psi$ if and only if $d \in \mathcal{L}(\mathcal{A})$.

$\Psi = \exists \boldsymbol{N} : \phi : N_1 E_1 \,\&\, \ldots \,\&\, N_p E_p$. 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 $\psi(M_1, \ldots, M_m) \vdash \mathit{Reg} \to \mathcal{Q}$, 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 formula, $\phi^{\exists}(M_1, \ldots, M_m)$, satisfied by configurations $\mathcal{Q}_{j_1} \cdot ... \cdot \mathcal{Q}_{j_n}$ 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^i_j for $j \in 1..p$, with x^i_j corresponding to a number of final states of $\mathcal{A}_j$ occurring in $\mathcal{Q}_i$.

$$\phi^{\exists} =_{\text{def}} \exists (x^i_j)_{\substack{i \in 1..m \\ j \in 1..p}} \cdot \bigwedge_{i \in 1..m} \Big(M_i = \sum_{\substack{j \in 1..p \\ \mathcal{Q}_i \in \mathrm{fin}(\mathcal{A}_j)}} x^i_j\Big) \wedge \phi\Big(\sum_{\substack{i \in 1..m \\ \mathcal{Q}_i \in \mathrm{fin}(\mathcal{A}_1)}} x^i_1, \ \ldots\ , \sum_{\substack{i \in 1..m \\ \mathcal{Q}_i \in \mathrm{fin}(\mathcal{A}_p)}} x^i_p\Big)$$

Finally, we split each constrained rule $\psi \vdash \mathit{Reg} \to \mathcal{Q}$ of $\mathcal{A}$ into the two rules $\psi \wedge \phi^{\exists} \vdash \mathit{Reg} \to \mathcal{Q}_T$ and $\psi \wedge \neg\phi^{\exists} \vdash \mathit{Reg} \to \bar{\mathcal{Q}}_T$, splitting also the state $\mathcal{Q}$ into $\mathcal{Q}_T$ and $\bar{\mathcal{Q}}_T$. The automaton $\mathcal{A}_\Psi$ is obtained by choosing the states of the form $\mathcal{Q}_S$ (where $\mathcal{Q}$ is final in $\mathcal{A}$) as the set of final states. The automaton is deterministic and complete and the property follows by showing that $d \models \Psi$ if and only if $d \in \mathcal{L}(\mathcal{A})$. □

As a direct corollary of Theorem 1 and Propositions 8 and 10, 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 2 (Decidability). *The logic SL is decidable.*

Theorem 3 (Model Checking). *For any document, d, and formula, Ψ, the problem $d \models \psi$ is decidable in time $O(|d| \cdot |\mathcal{A}_\psi| \cdot \mathit{Cost}(|Q(A_\Psi)|, |d|))$.*

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 11. *For every schema, S, there is a deterministic SA, $\mathcal{A}$, such that $L(\mathcal{A}) = \{d \mid d : S\}$, and for every recursive schema, S, there is a SA such that $L(\mathcal{A}) = \{d \mid d : S\}$.*

Proof. Similar to the proof of Theorem 1. In the case of recursive schemas, we need to introduce a special state q_X for each definition $X = T$ in S. Then we construct the automata corresponding to T and replace q_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, as well as automata-based decision procedures. We can, for example, easily define the intersection and difference of two schemas (that are not necessarily well-formed schemas).

Theorem 4 (XML Typing). *Given a document, d, and a schema, S, the problem $d : S$ is decidable.*

Theorem 5 (Satisfaction). *Given a schema S, the problem $\exists d \cdot d : S$ is decidable.*

7 Conclusion

Our contribution is a new class of automaton for unranked trees aiming at the manipulation of XML schemas. We believe it is the first work on applying tree automata theory to XML that considers the &-operator. This addition is significant in that interleaving is the source of many complications, essentially because it involves the combination of ordered and unordered data models. This led us to extend hedge automata [11] 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, like the undecidability of computing the complement of non-deterministic languages. To elude this problem, and in order to limit ourselves to deterministic automata, 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, namely that & may only appears at top-level position in an element definition.

Another source of problems is related to the size and complexity of counting constraints. While the complexity of many operations on Presburger 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. As a matter of fact, some simple syntactical restrictions on schemas yield simple Presburger formulas. For example, we may obtain polynomial complexity by imposing that each element tag in an expression $a_1[S_1]$ & ... & $a_p[S_p]$ be distinct, a restriction that also appears in the schema specification.

The goal of this work 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. Like for instance to add the capacity for the reverse traversal of a document or to extend our logic with some kind of path expression modality. These two extensions are quite orthogonal to what is already present in our logic and they could be added using some form of backtracking, like a parallel or alternating [7] variant of our tree automata, or by considering tree grammars (that is, 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. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.
2. A. Berlea and H. Seidl. Binary queries. In *Extreme Markup Languages*, 2002.
3. A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL: A model for W3C XML schema. In WWW 10, 2001.
4. L. Cardelli and G. Ghelli. A query language based on the ambient logic. In *European Symposium on Programming (ESOP)*, volume 2028 of *LNCS*, pages 1–22, 2001.
5. L. Cardelli and A. Gordon. Anytime, anywhere: Modal logic for mobile ambients. In *Principles of Programming Languages (POPL)*. ACM Press, 2000.
6. J. Clark and M. Makoto, editors. *RELAX-NG Tutorial*. OASIS, 2001.
7. H. Comon, M. Dauchet, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata and their application. To appear as a book, 2003.
8. S. Dal Zilio and D. Lugiez. XML schema, tree logic and sheaves automata. Technical Report 4631, INRIA, 2002.
9. H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *Principles of Programming Languages (POPL)*, pages 67–80. ACM Press, 2001.
10. D. Lugiez and S. Dal Zilio. Multitrees automata, Presburger's constraints and tree logics. Technical Report 08-2002, LIF, 2002.
11. M. Makoto. Extended path expression for XML. In *Principles of Database Systems (PODS)*. ACM Press, 2001.
12. A. Muscholl, T. Schwentick, and H. Seidl. Numerical document queries. In *Principle of Databases Systems (PODS)*. ACM Press, 2003.