XML schema, tree logic and sheaves automata

Silvano Dal Zilio, Denis Lugiez

XML Schema, Tree Logic and Sheaves Automata

Silvano Dal Zilio[⋆]**, Denis Lugiez**

Laboratoire d'Informatique Fondamentale de Marseille, CNRS and Université de Provence

The date of receipt and acceptance will be inserted by the editor

Abstract XML documents may be roughly described as unranked, ordered trees and it is therefore natural to use tree automata to process or validate them. This idea has already been successfully applied in the context of Document Type Definition (DTD), the simplest standard for defining document 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 introduce Sheaves Logic (SL), a new tree logic that extends the syntax of the — recursion-free fragment of — W3C XML Schema Definition Language (WXS). Then we define a new class of automata for unranked trees that provides decision procedures for the basic questions about SL: model-checking; satisfiability; entailment. The same class of automata is also used to answer basic questions about WXS, including recursive schemas: decidability of type-checking documents; testing the emptiness of schemas; testing that a schema subsumes another one.

Key words Tree automata – Modal logic – XML – XML Schema

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. Since XML documents and other forms of semi-structured data [1] can be described as unranked, ordered trees (an unranked tree is a finite labeled tree where nodes can have an arbitrary number of children), it is natural to use tree automata to reason on them and apply the classical connection between automata, logic and query languages.

⋆ This work was partially supported by ATIP CNRS "Fondements de l'Interrogation des Données Semi-Structurées" and by IST Global Computing PROFUNDIS.

This approach has already been successfully applied 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 Definitions* (DTD), the simplest standard for defining constraints on the shape of XML documents. A good example is the XDuce system of Pierce, Hosoya *et al.* [15], a statically typed functional language with "tree grep"-style patterns for traversing and manipulating XML. In this tool, types are modeled by regular tree automata (which are similar in spirit to DTD) and the typing of pattern matching expressions is based on closure operations on automata.

DTD is a schema language, that is, a description of document types expressed in terms of constraints on the structure and content of valid documents. The schemas expressible with DTD are sometimes too rigid and inadequate for many purposes. For instance, a document may become invalid after permutation of some of its elements. Several schema languages have been proposed to overcome these limitations, such as RELAX-NG [6] or the W3C XML Schema Definition Language (WXS) [3]. The specification of WXS is based on a notion of complex types that defines the content model of groups of elements. There are three possible grouping operators in WXS: (1) the `sequence` group that constrains elements to appear in the same order as they are declared; (2) the `choice` group that constrains only one element in a group to appear in an instance of the schema; and (3) the `all` group that constrains all the elements in the group to appear in any order. Informally, `sequence` and `choice` allows the expression of regular constraints (as with DTD), while the `all` group operator provides a simplified version of the $\&$ connector of SGML.

Contributions. Our first contribution is an automata theoretic approach for WXS, that relies on a new class of tree automata, named *sheaves automata* (SA). We define a simplified version of WXS that embeds regular tree expressions and sequential composition $(\cdot)$ together with an associative-commutative operator $(\&)$ to model the `all` group. Since we focus on the interactions between regular constraints and the `all` group, we leave out several other features of WXS, like mixed-content models, primitive types, or redefinition for example. To the best of our knowledge, it is the first work applying automata theory to WXS that considers the `all` group operator. Given a schema, we can built a sheaves automaton that recognizes the set of well-typed documents (Proposition 16). This property yields a procedure to decide whether a document is well-typed (Theorem 4) and to decide type inclusion (Theorem 6). Our approach provides a compact and efficient way to deal with the interleaving operator $\&$ without replacing the composition $E_1 \ \& \ \ldots \ \& \ E_p$ of p elements by a regular expression matching all the possible permutations of the E_i's.

The second contribution is a new modal logic for trees, the *sheaves logic* (SL), that extends the basic constructions of WXS with logical operators. This logic deliberately resembles TQL [4,5], a logic for unordered trees at the basis of a query language for semi-structured data. By design, every formula of SL directly relates to a deterministic sheaves automaton. As a result, we obtain the decidability of the *model-checking problem* (Theorem 3), that is finding if a document conforms

to a given schema, and of the *satisfiability problems*, that is finding if the model of a schema is empty (Theorem 2). There are several benefits in using logic instead of directly compiling WXS definitions into sheaves automata: SL offers a concrete syntax for describing languages recognizable by an SA; it is a test bed for possible extensions of WXS; it may be used as the basis of a query language that uses sheaves automata for traversing and manipulating documents. Also, from a theoretical point of view, the automata and the logic defined in this paper are interesting in their own rights. Indeed, a subclass of SA has already been used to obtain decidability results for the static fragment of the ambient logic [13].

Our third contribution is an extensive study of the properties of *sheaves automata*. Actually, the decidability results mentioned above directly follow from these properties. We prove that standard constructions (product, closure under union and intersection) and algorithms (decision of emptiness and membership) can be adapted to this class, but that there is no determinization algorithm. Actually, we exhibit a language accepted by a non-deterministic sheaves automaton that cannot be accepted by a deterministic automaton (Proposition 4). Furthermore, we show that the class of languages accepted by sheaves automata is not closed under complementation (Proposition 8).

Content of the paper. We start by defining a simplified syntax for XML documents and XML Schema. In Section 3, we introduce some basic mathematical tools used in the remainder of the paper and explain how counting constraints on documents may arise from the boolean combination of WXS definitions. In Section 4, we present the *Sheaves Logic* (SL), a new tree logic intended to describe validity constraints on XML documents. Section 5 introduces a new class of automata for unranked trees, called *Sheaves Automata* (SA), that is used to decide sheaves logic. In Section 6, we apply automata techniques to obtain decidability results for the sheaves logic, then the same tool is used to solve problems related to documents validation with respect to WXS definitions. Before concluding, we report on work related to logic and automata for unranked trees in the context of XML.

2 Documents and Schema

We define a simplified syntax for XML documents and XML schema and describe schema validation as a type checking process for documents.

XML documents may be seen as a simple textual representation for unranked, ordered labeled trees. In this article, we follow the notations of [15] and choose a simplified version of XML documents by leaving aside attributes and entities among other things. Most of the simplifications and notation conventions used here are also found in the presentation of MSL [3], an attempt to formalize some of the core ideas found in WXS.

2.1 XML Documents

We assume there are disjoint sets of *constants* and *tag* names. We let $c, c', \ldots$ range over constants and $a, b, \ldots$ range over tags. A document d is an ordered sequence of elements $a_1[d_1] \cdot \ldots \cdot a_n[d_n]$. A document may be empty, denoted ϵ, and documents may be concatenated, denoted $d \cdot d'$. This composition operation is associative with identity element ϵ.

Elements and Documents

$e ::=$	element or constant
$\quad a[d]$	element labeled a, containing d
$\quad c$	constant (any type)
$d ::=$	document
$\quad e_1 \cdot \ldots \cdot e_n$	document composition (with $n \geq 0$)

The XML specification states that a well-formed document must have a *root element*, that is, a unique top-level element. Hence, in our setting, a well-formed XML document is an element. We consider a finite set of primitive types, like `String` or `Integer` for instance. A primitive type is a set of atomic data constants and we use the notation `Datatype` to stand for any particular primitive type. We assume that every constant c belongs to a unique primitive type `Datatype`, denoted $c \in$ `Datatype`. A formal description of how types are associated to constants escape the scope of our study. We will also not consider subtyping relations between primitive types (as expressible in WXS).

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

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

2.2 Syntax of Schema

Schemas are the types of documents. We assume an infinite set of *schema variables* ranged over by $X, Y, \ldots$ We consider two separate syntactical categories for schemas: E for element schema definitions and T for top-level schemas. The notation $Reg(E_1, \ldots, E_p)$ stands for a regular expression on the elements $(E_i)_{i \in 1..p}$. It can be the empty sequence ϵ, any element E_i with $i \in 1..p$, the concatenation of two expressions $R \cdot R'$, choice $R \mid R'$ or iteration R^* where R, R' are regular expressions on the elements $(E_i)_{i \in 1..p}$.

Negation and conjunction of regular expressions are not required since they can be derived from these operations, nonetheless, in the remainder of the paper, we use the notation $\overline{Reg}$ to stand for a regular expression that matches the complement language of Reg and the notation $Reg \cap Reg'$ for an expression matching the intersection of the languages of Reg and Reg'.

Syntax of Schema (WXS)

$E ::=$	Element schema
$a[T]$	element with tag a and interior matching T
$a[T]?$	optional element
`Datatype`	datatype constant
$T ::=$	Top level schema
X	variable
$Reg(E_1, \ldots, E_p)$	regular expression of elements
$E_1 \;\&\; \cdots \;\&\; E_n$	interleaving composition ($n \geq 2$)
$AnyT$	any type (match everything)

A (top-level) 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 sub-document satisfying the schema T. An optional element $a[T]?$ matches one or zero occurrence of $a[T]$. The most original operator is the *interleaving connector*, $E_1 \;\&\; \cdots \;\&\; E_n$, which describes documents containing (exactly) elements matching E_1 to E_n regardless of their order. This operator corresponds to `all` groups in the concrete syntax of WXS. It is possible to describe the possible interleavings of a finite set of elements using a regular expression, for instance $|_{\sigma \text{ permutation of } 1..n} E_{\sigma(1)} \cdot \ldots \cdot E_{\sigma(n)}$ for the above case, but the size of this encoding is exponentially bigger than the size of the original expression. The interleaving operator gives a simple notation for such expressions and we shall see how sheaves automata provide an effective way to cope with this operator even in the presence of recursion. Our simplified description of WXS also contains the constant $AnyT$ — *Any Type* in WXS terminology — which matches every document and stands for the most general type.

The type of a document may be given by a set of recursive schema definitions together with the type associated to its root element, that is by an equation of the form

$$a[\,X_j\,] \;\; with \;\; X_1 = T_1, \ldots, X_n = T_n \,,$$

where X_j ($j \in 1..n$) is the type of the root element and $T_1, \ldots, T_n$ are top-level schema that only contain variables in $X_1, \ldots, X_n$. To comply with the WXS standard, we assume that there is only one equation $X_i = T_i$ for each variable X_i and that T_i is not a variable. These assumptions can be relaxed without changing our main results.

Example 2 The following schema matches the book entry given in Example 1:

$$\begin{aligned} book[Book] \; with \; Book &= auth[\texttt{String}] \;\&\; title[\texttt{String}] \\ &\quad \;\&\; date[\texttt{Integer}] \;\&\; ref[Ref]?, \\ Ref \;\; &= (entry[Book])^* \end{aligned}$$

The definition of the type for books entries consists of two equations. The first equation states that a bibliographical item includes three mandatory fields — author, $auth[\texttt{String}]$, title, $title[\texttt{String}]$, and publication year $date[\texttt{Integer}]$ — and one optional field for references (the order in which these fields appear

is irrelevant). The equation for Ref states that a reference is a possibly empty sequence of book entries. This type could be expressed as follows using the WXS concrete syntax:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="book"  type="Book"/>
  <xsd:complexType name="Book">
    <xsd:all>
      <xsd:element name="auth" type="xsd:string"/>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="date" type="xsd:integer"/>
      <xsd:element name="ref" type="Ref"
                   minOccurs="0" maxOccurs="1"/>
    </xsd:all>
  </xsd:complexType>
  <xsd:complexType name="Ref">
    <xsd:sequence>
      <xsd:element name="entry" type="Book"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Our simplified specification leaves out many features of WXS like complex datatypes, mixed content models, element and attribute groups, "object-oriented features" (like substitution groups and redefinitions), ... and focus instead on the interactions between the `all` and `sequence` group operators. Our syntax also captures some of the constraints put on these operators:

- an `all` group can only contain individual element declarations and not `choice` or `sequence` elements,
- no element may appear more than once in the "content model" of an `all` group, that is, the values of the `minOccurs` and `maxOccurs` attributes[1] must be 0 or 1.

For example, the terms $E_1 \cdot (E_2 \,\&\, E_3)$ and $(E_1 \,\&\, E_2)^*$ are ill-formed with our syntax and in the WXS specification.

In contrast, we do not limit regular expressions Reg to be 1-unambiguous, meaning that the typical algorithm used to test whether a word matches Reg does not require any look-ahead. This constraint, known in WXS as the *Unique Particle Attribution Rule*, also appears in the specification of DTD. We do not consider either the *Consistent Declaration Rule*, an equivalent restriction for interleaving compositions, $(a_1[T_1] \,\&\, \cdots \,\&\, a_p[T_p])$, which specifies that for all $i, j \in 1..p$, if $a_i = a_j$ then $T_i = T_j$. The motivation to include these restrictions in WXS is to keep schema processors simple to implement and to obtain a one pass typing property. These restrictions are not necessary to prove that every WXS definition

[1] For simplicity we have chosen iteration S^* and option $a[T]?$ instead of a more general repetition operator $E\{m, n\}$, expressible using `minOccurs` and `maxOccurs`.

may be interpreted by a sheaves automaton (see Section 6.1), but they can lessen the complexity of the constraints appearing in the SA obtained by our translation.

2.3 Semantics of Schema

We make explicit the role of schema as a type system for documents and define the relation $\mathcal{S} \vdash d : T$, meaning that the document d satisfies the schema T in the environment $\mathcal{S}$. In our setting, an environment is a set of equations $X_1 = T_1, \ldots, X_n = T_n$ obtained from a type declaration $a[\,X\,]\ with\ \mathcal{S}$. Hence $\mathcal{S}$ always defines a unique mapping between variables and top-level schema. We denote $\mathcal{S}(X)$ the unique type T associated to X in the environment $\mathcal{S}$ if it exists.

We say that a document $a[d]$ is of type $a[X]\ with\ \mathcal{S}$ if and only if $\mathcal{S} \vdash d : X$.

For the sake of readability, we use the 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:

$$inter(e_1 \cdot \ldots \cdot e_n) = \{e_{\sigma(1)} \cdot \ldots \cdot e_{\sigma(n)} \mid \sigma \text{ permutation of } 1..n\}\ .$$

The relation $w \in Reg(a_1, \ldots, a_n)$ means that w is a word recognized by the regular expression $Reg(a_1, \ldots, a_n)$. In the following, we use this relation in situations where the letters are element formulas and write $E_{i_1} \cdot \ldots \cdot E_{i_k} \in Reg(E_1, \ldots, E_n)$.

Valid Documents (WXS)

$$\frac{\mathcal{S} \vdash d : T}{\mathcal{S} \vdash a[d] : a[T]} \qquad \frac{\mathcal{S} \vdash d : a[T]}{\mathcal{S} \vdash d : a[T]?} \qquad \frac{}{\mathcal{S} \vdash \epsilon : a[T]?}$$

$$\frac{c \in \texttt{Datatype}}{\mathcal{S} \vdash c : \texttt{Datatype}} \qquad \frac{\mathcal{S}(X) = T \quad \mathcal{S} \vdash d : T}{\mathcal{S} \vdash d : X}$$

$$\frac{d = e_1 \cdot \ldots \cdot e_k \quad \mathcal{S} \vdash e_j : E_{i_j} \quad i \in 1..k \quad E_{i_1} \cdot \ldots \cdot E_{i_k} \in Reg(E_1, \ldots, E_n)}{\mathcal{S} \vdash d : Reg(E_1, \ldots, E_n)}$$

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

The reader may easily check from these rules that the document defined in Example 1 has the type $book[Book]$ given in Example 2.

3 Presburger Arithmetic, Parikh Mapping and Counting Constraints

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

3.1 Presburger Arithmetic

Some computational aspects of sheaves automata rely on arithmetical properties over the semigroup $(\mathbb{N}, +)$ of natural numbers with addition. Formulas of Presburger arithmetic, also called *Presburger constraints*, are defined in the table below. We assume an infinite set of *integer variables* ranged over by $N, M, \ldots$. We let $n, m, \ldots$ range over integer values. Presburger constraints allow us to define a substantial class of (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, $\exists Z.(X = Z + Z + 1)$.

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

We denote $\phi(\mathbf{N})$ a Presburger formula whose free variables are all in $\mathbf{N} = (N_1, \ldots, N_p)$ and we use the notation $\models \phi(n_1, \ldots, n_p)$ when $\phi\{N_1 \leftarrow n_1\} \ldots \{N_p \leftarrow n_p\}$, the formula ϕ where the variables $(N_i)_{i\in 1..p}$ have been substituted by the values $(n_i)_{i\in 1..p}$, is satisfied.

Presburger arithmetic is decidable, which means that for every formula $\phi(\mathbf{N})$ we can decide if there exists $(n_1, \ldots, n_p)$ such that $\models \phi(n_1, \ldots, n_p)$. Nonetheless, the complexity of deciding validity can be very high [14]: every algorithm which decides the truth of a Presburger constraint ϕ has worst case runtime of at least $2^{2^{cn}}$ for some constant c, where n is the length of ϕ. Conversely there is also a known triply exponential upper-bound in the worst case [26], i.e. the complexity of checking the satisfiability of a formula ϕ is in time at most $2^{2^{2^{cn}}}$ for some constant c. Furthermore the problem is NP-complete for the existential fragment of Presburger arithmetic.

The constraints arising in the study of our simplified fragment of WXS will stay in a simple fragment of Presburger arithmetic. For the sheaves logic, instead, we will consider the most general class of constraints.

3.2 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 Σ^* is a n-tuple 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. Parikh's theorem states that the Parikh mapping of a context-free language is definable by a Presburger formula and that this formula can be explicitly computed. If the language L is regular, a Presburger formula representing the Parikh mapping of L can be computed in linear time [28]. 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 Section 4, for example, when we test the emptiness of the language accepted by a sheaves automaton.

3.3 Relation with XML Schema

In the following section we study a modal logic for documents that directly embeds WXS. This logic is obtained by extending the syntax given in Section 2.2 with operators for disjunction and negation (as usual), and by adding arithmetical constraints on the number of elements to the interleaving composition. For instance, it is possible to define formulas of the form $\exists N_1, N_2 : (N_1 = N_2) : N_1\, a[True] \;\&\; N_2\, b[True]$, meaning that a valid document should be the composition of n_1 elements labeled a and n_2 elements labeled b, regardless of their order, with the constraint that $n_1 = n_2$.

To motivate the use of counting constraints, we consider an example that shows how a boolean combination of $\&$-compositions introduces "counting capabilities" to schema. The following example cannot be directly expressed in WXS, but could be obtained when computing the intersection, composition and interleaving of sets of documents recognized by schemas (which may arise, for example, when typing queries):

$$title[\texttt{String}] \;\&\; auth[\texttt{String}] \;\&\; \neg(title[True] \;\&\; AnyT)\,. \qquad (1)$$

Assume we have documents composed of only three tags: $author$, $title$ and $date$. Informally, the expression $\neg(title[True] \;\&\; AnyT)$ matches documents that do not have a $title$ element (it cannot be decomposed into any document, matched by $True$, composed with an element labeled $title$). Hence a bibliographical entry matching (1) must contain *exactly* one field labeled $title$ and *at least* one field labeled $auth$. These constraints could be expressed more directly using Presburger constraints by saying that a valid entry must be of the form $n_t\, title[\texttt{String}] \;\&\; n_a\, auth[\texttt{String}] \;\&\; n_d\, date[AnyT]$ where (n_t, n_a, n_d) is a sequence satisfying the constraint $\phi(N_t, N_a, N_d) =_{\text{def}} (N_t = 1 \wedge N_a \geqslant 1)$. Therefore we can characterize the set of documents matching (1) by the pair made of the formula ϕ and the sequence of element schemas $(title[\texttt{String}], auth[\texttt{String}], date[AnyT])$. As a consequence, it appears that boolean combinations of schemas may be used to define a mixture of regular and counting constraints on the sequence of elements occurring in a document.

One of the main contributions of this paper is to show that a boolean combination of schemas can be related to a triple $(Reg, \phi, (E_1, \ldots, E_n))$ made of a sequence of element formulas, a Presburger constraint ϕ with n variables, and a

regular expression Reg with atoms in $E_1, \ldots, E_n$. We use this "normal form" for WXS definitions as a basis for defining a new class of tree automata, which in turn is used to prove the decidability of type-checking documents (Theorem 4). We also prove that, if arbitrary conjunction and concatenation of schemas were allowed, then type-checking becomes undecidable. This last result is obtained through the study of a modal logic for documents, defined in Section 4, that extends WXS.

4 Modal Logics for Documents

Now, we define a modal logic for documents, the "General Document Logic" (GDL), that extends the basic constructs of the W3C XML Schema (sequential and interleaving composition) with counting constraints and logical connectives (but without recursive definitions). This logic is in the spirit of the Tree Query Logic (TQL) of Cardelli and Ghelli [5], a modal logic for unranked, unordered trees that has recently been proposed as the basis of a query language for semi-structured data. We show that our first attempt to extend WXS is too expressive (Proposition 1, the satisfaction problem for GDL is undecidable) and identify a decidable fragment of GDL, called the *Sheaves Logic* (SL).

4.1 Syntax of Formulas

The formulas of GDL, ranged over by $D, A, B, \ldots$ are given by the following grammar. Aside from the usual propositional logic operators, formulas are built from three main ingredients: (1) *element formulas* $a[D]$ to express properties of a single element in a document; (2) *regular formulas* $Reg(D_1, \ldots, D_p)$ corresponding to regular expressions on sequences of documents; (3) *counting formulas* $\exists \mathbf{N} : \phi(\mathbf{N}) : N_1 D_1 \ \& \ \ldots \ \& \ N_p D_p$ to express counting constraints on bags of elements, that is in situations where the order of the elements is irrelevant. This last operator is the main addition to the logic and is used to match documents obtained by interleaving n_1 documents matching $D_1, \ldots, n_p$ documents matching D_p, such that $(n_1, ..., n_p)$ satisfies the Presburger formula ϕ.

Syntax of Formulas (GDL)

$E ::=$	Element
$\quad a[D]$	element with tag a and formula D
$\quad$ `Datatype`	datatype constant
$D ::=$	
$\quad E$	element
$\quad Reg(D_1, \ldots, D_p)$	regular expression on formulas
$\quad \exists \mathbf{N} : \phi(\mathbf{N}) : N_1 D_1 \ \& \ \ldots \ \& \ N_p D_p$	generalized interleaving, $\mathbf{N} = (N_1, \ldots, N_p)$
$\quad True$	any document (truth formula)
$\quad D \vee D'$	disjunction
$\quad \neg D$	negation

The generalized interleaving operator is inspired by the relation between schema and counting constraint given in Section 3.3. This operator is useful to express constraints on documents more expressive than with WXS. For example, it is possible to define a type equivalent to $(E_1^* \ \& \ E_2)$, that matches documents made only of elements matching E_1 but one matching E_2, using the formula $\exists N_1, N_2 : (N_1 \geqslant 0) \wedge (N_2 = 1) : N_1 E_1 \ \& \ N_2 E_2$. For the sake of simplicity, we introduce a unique terminal `Datatype` whereas a more realistic approach would use several ones. Our results can be easily restated in this extended framework which simply adds some tedious technical complications.

4.2 Satisfaction Relation

We define the relation $d \models D$, meaning that the document d satisfies the formula D. This relation is defined inductively on the definition of D. As with the definition of the WXS semantics, we use the relation $W \in Reg(D_1, \ldots, D_p)$ meaning that W belongs to the language generated by Reg (observe that, in this case, W is a word on the alphabet $\{D_1, \ldots, D_p\}$ made of formulas). We also use the notation $W \in \phi(D_1, \ldots, D_p)$ when the Parikh mapping of the sequence W satisfies the formula ϕ, that is, $\models \phi(n_1, \ldots, n_p)$ where n_j is the number of occurrences of D_j in W.

Satisfaction (GDL)

$d \models a[D]$	iff $(d = a[d']) \wedge (d' \models D)$
$d \models$ `Datatype`	iff $(d = c) \wedge (c \in$ `Datatype`$)$
$d \models Reg(D_1, \ldots, D_p)$	iff $d = d_1 \cdot \ldots \cdot d_k \ \wedge$ $\exists i_1, \ldots, i_k . \bigwedge_{j \in 1..k} d_j \models D_{i_j} \ \wedge$ $D_{i_1} \cdot \ldots \cdot D_{i_k} \in Reg(D_1, \ldots, D_p)$
$d \models \exists \mathbf{N} : \phi(\mathbf{N}) : N_1 D_1 \ \& \ \ldots \ \& \ N_p D_p$	iff $d \in inter(d_1 \cdot \ldots \cdot d_k) \wedge$ $\exists i_1, \ldots, i_k . \bigwedge_{j \in 1..k} d_j \models D_{i_j} \ \wedge$ $D_{i_1} \cdot \ldots \cdot D_{i_k} \in \phi(D_1, \ldots, D_p)$
$d \models True$	always true
$d \models D \vee D'$	iff $(d \models D) \vee (d \models D')$
$d \models \neg D$	iff not $(d \models D)$

4.3 Expressiveness of the Logic

We start by defining some syntactic sugar in order to give examples of schemas expressible in GDL. We use 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.

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

Likewise, we define the notation $(a_i[S] \ \& \ \cdots)$ for the formula satisfied by documents containing at least one element matching $a_i[S]$. We assume here a finite

set of possible values for tags, say $\{a_1, \ldots, a_k\}$, but it is possible to handle an unbounded number of tags using a minor extension of the logic (see for example the approach taken in [13]).

$$(a_i[S] \ \& \cdots) \ =_{\text{def}} \ \exists M, N_1, \ldots, N_k : (M = 1) \wedge \textstyle\bigwedge_{i \in 1..k} (N_i \geqslant 0) : \\ M a_i[S] \ \& \ N_1 a_1[True] \ \& \cdots \& \ N_k a_k[True]$$

As 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 5 entries written by Knuth or Lamport.

$$\exists N, M : (N + M \geqslant 5) : \begin{pmatrix} N\, book[(auth[\text{"Knuth"}] \ \& \cdots)] \\ \& \ M\, book[(auth[\text{"Lamport"}] \ \& \cdots)] \end{pmatrix}$$

The following theorem states that GDL is too expressive.

Proposition 1 *The satisfaction problem for GDL is undecidable.*

Proof We show that given a two-counter machine, there is a formula matching exactly the set of terminating computations of the machine. Therefore, deciding the satisfiability of GDL formulas would imply deciding the halting problem for two-counter machines which is undecidable. The complete proof is given in Appendix A.1. □

4.4 A Decidable Document Logic

We define a fragment of GDL by restricting regular expressions and interleaving operators to act only on element formulas. We call this subset the *Sheaves Logic* (SL). We prove (Theorem 2) that the satisfaction problem for SL is decidable. The property follows from a reduction to a new class of tree automata, the so-called *sheaves automata*, in the sense that the set of documents matched by a formula in SL will correspond to the set of terms accepted by an automaton.

Syntax of Formulas (SL)

$E ::=$	Element
$\quad a[D]$	element with tag a and formula D
$\quad$ `Datatype`	datatype constant
$D ::=$	
$\quad Reg(E_1, \ldots, E_p)$	regular expression on elements
$\quad \exists \mathbf{N} : \phi(\mathbf{N}) : N_1 E_1 \ \& \ \ldots \ \& \ N_p E_p$	generalized interleaving, $\mathbf{N} = (N_1, \ldots, N_p)$
$\quad True$	any document (truth formula)
$\quad D \vee D'$	disjunction
$\quad \neg D$	negation

The definition of the satisfaction relation for SL can be slightly simplified in the case for generalized interleaving. The definition for the other operators is unchanged.

Satisfaction (SL)

$$
\begin{array}{lll}
d \models \mathit{Reg}(E_1, \ldots, E_p) & \text{iff} & d = e_1 \cdot \ldots \cdot e_k \ \wedge \\
 & & \exists i_1, \ldots, i_k . \bigwedge_{j \in 1..k} e_j \models E_{i_j} \ \wedge \\
 & & E_{i_1} \cdot \ldots \cdot E_{i_k} \in \mathit{Reg}(E_1, \ldots, E_p) \\
d \models \exists \mathbf{N} : \phi(\mathbf{N}) : N_1 E_1 \ \& \ \ldots \ \& \ N_p E_p & \text{iff} & d = e_1 \cdot \ldots \cdot e_k \ \wedge \\
 & & \exists i_1, \ldots, i_k . \bigwedge_{j \in 1..k} e_j \models E_{i_j} \ \wedge \\
 & & E_{i_1} \cdot \ldots \cdot E_{i_k} \in \phi(E_1, \ldots, E_p)
\end{array}
$$

We can explain the difference in expressiveness between SL and GDL on a simple example. The formula $AB =_{\text{def}} (a[True]^* \cdot b[True]^*) \wedge \exists (N_a, N_b) : (N_a = N_b) : N_a\, a[True] \ \& \ N_b\, b[True])$ is in SL. This formula matches document of the form $a[\_]^n \cdot b[\_]^n$ (think of the word language $a^n \cdot b^n$) that can be recognized with a one-counter automata. The formula $AB \cdot AB$, which is in GDL but not in SL, matches documents of the form $a[\_]^n \cdot b[\_]^n \cdot a[\_]^m \cdot b[\_]^m$, with $n, m > 0$. We prove, see Proposition 4, that this language cannot be accepted by any deterministic sheaves automaton.

Next, we state that we can always assume that the element formulas $E_1, \ldots, E_p$ occurring in a regular expression or a generalized interleaving operator are pairwise disjoint, i.e. the models of E_i and E_j are disjoint when $i \neq j$. This property is used in the proof of Theorem 1.

Proposition 2 *From a regular formula $\mathit{Reg}(E_1, \ldots, E_p)$ of SL we can build an equivalent formula $\mathit{Reg}'(E'_1, \ldots, E'_m)$ such that the element formulas $E'_1, \ldots, E'_m$ are pairwise disjoint. Likewise, from a counting formula $\exists \mathbf{N} : \phi(\mathbf{N}) : N_1\, E_1 \ \& \ \ldots \ \& \ N_p\, E_p$ of SL we can build an equivalent formula $\exists \mathbf{M} : \phi'(\mathbf{M}) : M_1\, E'_1 \ \& \ \ldots \ \& \ M_m\, E'_m$ such that the element formulas $E'_1, \ldots, E'_m$ are pairwise disjoint.*

Proof We build a sequence of pairwise disjoint element formulas $E'_1, \ldots, E'_m$ such that any regular (resp. counting) formula on the sequence $E_1, \ldots, E_p$ is equivalent to a regular (resp. counting) formula on $E'_1, \ldots, E'_m$. Since `Datatype` is already disjoint from any $a[D]$, without loss of generality, we can assume $E_i = a_i[D_i]$ for all $i \in 1..p$. Let I_a denote the set of indexes i such that $a_i = a$. For each non-empty subset I of I_a, we set $D^a_I =_{\text{def}} \bigwedge_{j \in I} D_j$ if $I = I_a$, otherwise $D^a_I =_{\text{def}} \bigwedge_{j \in I} D_j \wedge \bigwedge_{j \in I_a \setminus I} \neg D_j$. By construction, the formulas D^a_I are pairwise disjoint and E_i is equivalent to $a_i[\bigvee_{I \subseteq I_{a_i}, i \in I} D^{a_i}_I]$.

Let $E^i_1, \ldots, E^i_{n_i}$ be the sequence of formulas $a_i[D^{a_i}_I]$ for all set of index I such that $i \in I$ and $I \subseteq I_{a_i}$. By construction, an element $e = a[d]$ models $a[D^a_I]$ if and only if I is the set of indexes i such that $d \models D_i$, or equivalently $I = \{i \mid e \models E_i\}$. Hence, $d \in \mathit{Reg}(E_1, \ldots, E_p)$ iff $d \in \mathit{Reg}\{E_1 \leftarrow (E^1_1 \mid \ldots \mid E^1_{n_1})\} \ldots \{E_p \leftarrow (E^p_1 \mid \ldots \mid E^p_{n_p})\}$, the regular expression obtained from Reg by substituting E_k with the expression $(E^k_1 \mid \ldots \mid E^k_{n_k})$ for all $k \in 1..p$. (This property follows from a simple structural induction on the syntax of Reg.)

Likewise, since each E_i is equivalent to the disjoint sum of the E^i_j's, for $j \in 1..n_i$, the counting formula $\exists \mathbf{N} : \phi(\mathbf{N}) : N_1\, E_1 \ \& \ \ldots \ \& \ N_p\, E_p$ is equivalent to the formula: $\exists \mathbf{M} : \phi(\sum_{j \in 1..n^1_1} M^1_j, \ldots, \sum_{j \in 1..n^p_p} M^p_j) : M^1_1\, E^1_1 \ \& \ \ldots \ \& \ M^p_{n_p}\, E^p_{n_p}$, which yields the result for a counting formula. □

5 A New Class of Tree Automata

We define a new class of tree automata, named sheaves automata (SA), specifically designed to operate with WXS. A main distinction with other automata-theoretic approaches is that we do not focus on regular expressions over paths but instead concentrate on the `all` group operator (denoted & in our simplified syntax), which is one of the chief additions of WXS with respect to DTD.

In the transition relation of SA, we combine the general rules for regular tree automata with regular word expressions and counting constraints. In this framework regular word expressions allow us to express constraints on sequences of elements and are used when dealing with sequential composition of documents (the `sequence` operator of WXS). Correspondingly, the Presburger constraints are used when dealing with interleaving composition (the `all` group of WXS) and appear as the counterpart of regular expressions when the order of the elements is not relevant.

We assume an infinite set of *states* ranged over by $q, q', \ldots$ A (bottom-up) sheaves automaton $\mathcal{A}$ is a triple $\langle Q, Q_{\text{fin}}, R\rangle$ where Q is a finite set of states of cardinality $|Q| = p$, and Q_{fin} is a set of final states included in Q, and R is a set of transition rules. Transition rules are of three kinds:

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

In type 3 rules, $\mathit{Reg}(Q)$ is a regular expression on the alphabet $Q = \{q_1, \ldots, q_p\}$ and $\phi(N_1, \ldots, N_p)$ is a Presburger arithmetic formula with free variables $N_1, \ldots, N_p$. 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 "$\cdot$" (the only nodes with an unbounded arity). Intuitively, the variable N_i denotes the number of occurrences of the state q_i in a run of the automaton and a type 3 rule may fire if we have a term of the form $e_1 \cdot \ldots \cdot e_n$ such that:

- each e_i leads to a state $q_{j_i} \in Q$;
- the word $q_{j_1} \cdot \ldots \cdot q_{j_n}$ is in the language defined by $\mathit{Reg}(Q)$;
- 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 $\mathit{Reg}(Q)$, we will use $\#q_i$ instead of N_i for the names of integer variables.

Example 3 Let the signature be $\{c, a[\_], b[\_]\}$. We define the automaton $\mathcal{A}$ by the set of states $Q = \{q_a, q_b, q_s\}$, the set of final states $Q_{\text{fin}} = \{q_s\}$ and the following set of five transition rules:

$$c \to q_s \quad a[q_s] \to q_a \quad b[q_s] \to q_b \quad (\#q_a = \#q_b) \vdash (q_a \mid q_b \mid q_s)^* \to q_s$$

We show in Example 4, after defining the transition relation, that this particular automaton accepts terms with as many a's as b's in the children of a node, as in the example $b[\epsilon] \cdot a[c \cdot b[\epsilon] \cdot c \cdot a[\epsilon]]$.

The constant $True$ stands for any tautology in Presburger arithmetic (for example $\exists X.(X = X)$). Likewise, we use All_Q for the regular expression $(q_1 \mid \ldots \mid q_p)^*$ that matches all possible words in the alphabet Q. If we drop the Presburger arithmetic constraint and restrict to type 3 rules of the form $True \vdash Reg(Q) \to q$, we get *hedge automata* [20]. Conversely, if we drop the regular word expression and restrict to rules of the form $\phi(\#q_1, \ldots, \#q_p) \vdash All_Q \to q$, we get a class of automata which enjoys all the good properties of regular tree automata [7, 13], that is closure under boolean operations, a determinization algorithm, decidability of the test for emptiness, *etc.* When both counting and regular word constraints are needed, some of these properties are no longer valid, as shown below. For instance, we prove in Proposition 4 that non-deterministic SA are not closed under determinization.

5.1 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 relation defined by the following three rules.

Transition Relation: $\to$

(type 1)	(type 2)	(type 3)
		$e_1 \to q_{j_1} \quad \ldots \quad e_n \to q_{j_n}$
	$d \to q'$	$q_{j_1} \cdot \ldots \cdot q_{j_n} \in Reg \quad \models \phi\#(q_{j_1} \cdot \ldots \cdot q_{j_n})$
$\dfrac{c \to q \in R}{c \to q}$	$\dfrac{a[q'] \to q \in R}{a[d] \to q}$	$\dfrac{\phi \vdash Reg \to q \in R \quad (n \neq 1)}{e_1 \cdot \ldots \cdot e_n \to q}$

The rule for constrained transitions (type 3 rules), can only be applied to sequences of length different from 1. Therefore it could not be applied to a sequence of only one element. It is possible to extend the transition relation for type 3 rules to also take into account this particular case, but it would needlessly complicate our definitions and proofs without adding expressivity.

Example 4 Let $\mathcal{A}$ be the automaton defined in Example 3 and d be the document $c \cdot a[\epsilon] \cdot c \cdot b[a[\epsilon] \cdot b[\epsilon]]$. The following proof tree describes how to use the transition rules of $\mathcal{A}$ to accept d:

$$\dfrac{c \to q_s \qquad \dfrac{\epsilon \to q_s}{a[\epsilon] \to q_a} \qquad c \to q_s \qquad \dfrac{\dfrac{\dfrac{\epsilon \to q_s}{a[\epsilon] \to q_a} \quad \dfrac{\epsilon \to q_s}{b[\epsilon] \to q_b} \quad (\star)}{a[\epsilon] \cdot b[\epsilon] \to q_s}}{b[a[\epsilon] \cdot b[\epsilon]] \to q_b} \quad (\star)}{c \cdot a[\epsilon] \cdot c \cdot b[a[\epsilon] \cdot b[\epsilon]] \to q_s}$$

The transition $\epsilon \to q_s$ and the two transitions marked with a $(\star)$-symbol use the only constrained rule of $\mathcal{A}$. The words used to check the constraints are ϵ, $q_a \cdot q_b$ and $q_s \cdot q_a \cdot q_s \cdot q_b$. It is easy to check that these words belongs to $All_Q = (q_a \mid q_b \mid q_s)^*$ and that they contain as many q_a's as q_b's (their respective Parikh mapping are $(0,0,0)$, $(1,1,0)$ and $(1,1,2)$).

Our example shows that SA can accept languages which are different from regular tree languages. For instance, as shown by Example 4, we can recognize trees in which the sequence of children of every node contains as many a's as b's. Indeed, the constrained rule in Example 3 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 as q_b's*." It is even possible to write constraints defining languages which are not 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 3).

As 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 $d \to_{\mathcal{A}} q$. The language $\mathcal{L}(\mathcal{A})$ is the set of terms accepted by $\mathcal{A}$. An automaton is *deterministic* iff two distinct rules have incompatible premises, i.e.:

- For any pair of distinct type 1 rules $c \to q$, $c' \to q'$, we have $c \neq c'$,
- For any pair of distinct type 2 rules $a[q] \to r$, $a'[q'] \to r'$, we have $a \neq a'$ or $q \neq q'$,
- For any pair of distinct type 3 rules $\phi \vdash Reg \to q$ and $\phi' \vdash Reg' \to q'$, there is no sequence of states in $Reg \cap Reg'$ satisfying $\phi \wedge \phi'$.

By construction a deterministic sheaves automaton is *unambiguous*, in that a term reaches at most one state. Given a sheaves automaton, it is possible to check if this automaton is deterministic. The only difficult case is for type 3 rules: by Parikh's theorem, we can compute a Presburger formula ψ that matches exactly the Parikh mapping of the regular language $Reg \cap Reg'$ and then check the validity of $\psi \wedge \phi \wedge \phi'$.

In the following, we will only consider *complete automata*, such that every term reaches some state. This can be done without loss of generality since, for any automaton $\mathcal{A}$ it is always possible to build an equivalent complete automaton $\mathcal{A}_c$.

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

Proof The construction is similar to the standard construction for finite state automata: add one sink state with the corresponding transition rules. The only technical point is to preserve determinism (obtained using the closure of regular expressions and Presburger formulas under boolean combinations). □

In the following sections, we enumerate several properties of our new class of automata.

5.2 Deterministic SA are less Powerful than Non-deterministic SA.

The following proposition states a first discrepancy between the properties of sheaves automata and regular tree automata.

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

Proof We prove that the language L, consisting of the terms $(a[\epsilon])^n \cdot (b[\epsilon])^n \cdot (a[\epsilon])^m \cdot (b[\epsilon])^m$, with $n, m > 0$, is not recognizable by a deterministic SA, although there is a non-deterministic SA accepting L. The proof that a deterministic automaton cannot recognize the language L is based on an adaptation of the pumping lemma, whereas we exhibit a non-deterministic automaton that recognizes L. The complete proof is given in Appendix A.2. □

5.3 A Determinizable Subclass

In this section, we prove that in some cases it is possible to compute a deterministic automaton accepting the same language as a given automaton. This is the case for the class of *separated* automata, defined below.

We say that an automaton is *separated* if and only if each type 3 rule either has the form $True \vdash Reg \rightarrow q$ or the form $\phi \vdash All_Q \rightarrow q$. In other words, in all type 3 rule, either the regular part or the counting part is trivial (but the same state q may appear on the right-hand part of a counting rule *and* of another regular rule).

Proposition 5 *Let $\mathcal{A}$ be a separated automaton, then there exists a deterministic sheaves automaton accepting the same language as $\mathcal{A}$.*

Proof The proof relies on an adaptation of the subset construction. The complete proof is given in Appendix A.3. □

We stress that the deterministic automaton computed from a non-deterministic separated automaton is not necessarily (actually usually not) separated.

5.4 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. Let us recall that given two languages L on the alphabet Σ, L' on the alphabet Σ', the product $L \times L'$ is the language on $\Sigma \times \Sigma'$ consisting of words w such that $\pi_1(w) \in L$ and $\pi_2(w) \in L'$ where π_1, π_2 are the morphisms such that $\pi_1((a, a')) = a, \pi_2((a, a')) = a'$ for each $a \in L, a' \in L'$. If L and L' are regular languages, then $L \times L'$ is a regular language.

Assume $Q = \{q_1, \ldots, q_p\}$ and $Q' = \{q'_1, \ldots, q'_l\}$ are the states of the automata $\mathcal{A}$ and $\mathcal{A}'$. 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 pair of 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 pair of type 2 rules $a[q] \to s \in R$ and $a[q'] \to s' \in R'$, the rule $a[(q, q')] \to (s, s')$ is in $R^\times$,
- for every pair of type 3 rules $\phi \vdash Reg \to q \in R$ and $\phi' \vdash Reg' \to q' \in R$, the rule $\phi^\times \vdash Reg^\times \to (q, q')$ is in $R^\times$, where $Reg^\times$ is the regular expression corresponding to the product $Reg \times Reg'$. 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 6 *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 $d \to_{\mathcal{A}'} q'$.*

Proof The proof for the first implication, i.e. $d \to (q, q')$ in $\mathcal{A} \times \mathcal{A}'$ implies $d \to_{\mathcal{A}} q$ and $d \to_{\mathcal{A}'} q'$, is a straightforward induction on the derivation of $d \to (q, q')$. The proof for the converse property is similar.

Assume $d \to (q, q')$ in the automaton $\mathcal{A} \times \mathcal{A}'$. We only study the case where the last transition rule is a type 3 rule $\phi^\times \vdash Reg^\times \to (q, q')$, coming from the product of the rules $\phi \vdash Reg \to q$ in $\mathcal{A}$ and $\phi' \vdash Reg' \to q'$ in $\mathcal{A}'$. Hence $d = e_1 \cdot \ldots \cdot e_n$ where $e_i \to_{\mathcal{A} \times \mathcal{A}'} (q_{j_i}, q'_{k_i})$ for all $i \in 1..n$.

Let $w^\times$ be the sequence of states (in $\mathcal{A} \times \mathcal{A}'$) used in the derivation of the transition, that is, $w^\times = (q_{j_1}, q'_{k_1}) \cdot \ldots \cdot (q_{j_n}, q'_{k_n})$. By definition of the transition relation we have $\models \phi^\times \#(w^\times)$.

Let $w = q_{j_1} \cdot \ldots \cdot q_{j_n}$ be the sequence of states of $\mathcal{A}$ in $w^\times$ and $w' = q'_{k_1} \cdot \ldots \cdot q'_{k_n}$. By induction, we have $e_i \to_{\mathcal{A}} q_{j_i}$ and $e_i \to_{\mathcal{A}'} q'_{k_i}$ for all $i \in 1..n$. By definition of Parikh mapping, we have $\#_q(w) = \sum_{q' \in Q'} \#_{(q,q')}(w^\times)$ (and a similar condition for w'). Hence, by definition of $\phi^\times$, it follows that $\models \phi \#(w)$ and $\models \phi' \#(w')$. Therefore $d \to_{\mathcal{A}} q$ and $d' \to_{\mathcal{A}'} q'$ as needed. □

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:

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

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 zero. We choose a more complex construction to preserve determinism.

Proposition 7 *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.*

Assume $\mathcal{A}$ is a complete and deterministic sheaves automaton. In most cases, a state q of $\mathcal{A}$ can appear on the right hand side of different rules, possibly of different types. Actually, it is always possible to obtain an automaton equivalent to $\mathcal{A}$ such that a state q cannot be the right-hand side of a type (1) or (2) rule and the right-hand side of a type (3) rule: replace $\mathcal{A}$ by the intersection $\mathcal{A} \cap \mathcal{B}$, where $\mathcal{B}$ is the sheaves automaton $\langle Q_{\mathcal{B}}, Q_{\mathcal{B}}, R_{\mathcal{B}} \rangle$ such that $Q_{\mathcal{B}} = \{q_0, q_3\}$, all states of $\mathcal{B}$ are final and the rules in $R_{\mathcal{B}}$ are $a[q_0] \to q_0$, $a[q_3] \to q_0$ (for all tag a that appears in $\mathcal{A}$), $True \vdash \epsilon \to q_0$, and $True \vdash (q_0 \mid q_3)^+ \to q_3$. Accordingly, we can always assume that regular and counting constraints in type (3) rules mention only states that are not the right-hand side of a type (3) rule. We assume that this condition is met by the automata considered in the remainder of the section.

5.5 Complement

Given a deterministic complete automaton $\mathcal{A}$ we 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 8 *Non-deterministic sheaves languages are 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 were also accepted by some automaton, we could derive an automaton accepting the (good) computations reaching a final state, hence decide if the machine halts. This is not possible since the halting problem for two-counter machines is undecidable. The complete proof is given in Appendix A.4. □

5.6 Membership

We consider the problem of checking whether a document d is accepted by an automaton $\mathcal{A}$, which we write $d \stackrel{?}{\in} \mathcal{L}(\mathcal{A})$. We use the notation $|d|$ for the number of elements occurring in d and $|S|$ for the number of elements in a set S.

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 $O(Cost(p, n))$ 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. In the general case, that is for formulas involving any alternation of quantifiers (which is very unlikely to occur in practice), the complexity is at least doubly exponential for a non-deterministic algorithm.

Proposition 9 *The problem $d \stackrel{?}{\in} \mathcal{L}(\mathcal{A})$, where $\mathcal{A} = \langle Q, Q_{fin}, R \rangle$ is a deterministic automaton, can be decided in time $O(|d| \cdot |R| \cdot Cost(|Q|, |d|))$.*

The proof is similar to the proof for tree automata. For non-deterministic automata, we prove that the problem is NP-complete even for *simple* sheaves automata, i.e., separated automata such that $Cost(p, n)$ is polynomial and where each regular expression occurring in a type 3 rule is trivial.

Proposition 10 *For a non-deterministic simple automaton $\mathcal{A} = \langle Q, Q_{fin}, R\rangle$, the problem $d \in \mathcal{L}(\mathcal{A})$ is NP-complete.*

Proof Membership to NP is easy: guess a labeling of d by states of the automaton where the root is labeled by an accepting state, then check that the labeling is correct. NP-completeness is shown by encoding 3-SAT. Given an instance of 3-SAT on the propositional variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$, we construct a term d and an automaton $\mathcal{A}$ such that $d \in \mathcal{L}(\mathcal{A})$ if and only if the 3-SAT instance is satisfiable. The signature used to encode a 3-SAT instance consists of the document composition $\cdot$, one constant 0 and one tag name a. The set of states of the automaton is $Q = \{q_0, q_\perp, q_S, q_1, \ldots, q_n\}$ where q_S is the unique final state.

Before describing the transition rules of $\mathcal{A}$ we detail the construction of the Presburger constraint $\#(C)$ associated to a 3-clause C. We define

$$\#(C) =_{\text{def}} \Sigma_{x_i \text{ occurs positively in } C} \#q_i + \Sigma_{x_i \text{ occurs negatively in } C}(1 - \#q_i) \geq 1$$

For instance, if C is the clause $x_1 \vee \neg x_2 \vee x_3$, then $\#(C)$ is the constraint $\#q_1 + (1 - \#q_2) + \#q_3 \geq 1$. When the $\#q_i$'s belong to the set $\{0, 1\}$, a conjunction of clauses $\bigwedge_{j\in J} C_j$ is satisfiable if and only if the Presburger constraint $\bigwedge_{j\in J} \#(C)$ is satisfiable.

Now we define the rules of $\mathcal{A}$. Type 1 and 2 rules of the automaton are $0 \to q_0$, $a[q_\perp] \to q_\perp$, $a[q_{i-1}] \to q_i$ and $a[q_{i-1}] \to q_\perp$ for $i \in 1..n$. The unique type 3 rule of $\mathcal{A}$ is:

$$\begin{aligned}\big(\textstyle\bigwedge_{i\in 1..n}(\#q_i \leq 1)\big) \wedge (\#q_\perp \geq 0) \wedge (\#q_S = 0)\\ \wedge (\#q_0 = 0) \wedge \big(\textstyle\bigwedge_{j\in 1..m} \#(C_j)\big) \vdash All_Q \to q_S\ .\end{aligned}$$

Assume $a^k[0]$ is the term $a[\ldots a[0]\ldots]$ obtained by nesting k occurrences of the tag a, that is, $a^{k+1}[0] =_{\text{def}} a[(a^k[0])]$ and $a^1[0] = a[0]$. The term d used in our encoding is $a[0] \cdot a^2[0] \cdot \ldots \cdot a^n[0]$. Remark that the document $a^i[0]$ may reach (non-deterministically) either q_i or $q_\perp$, therefore we can represent the assignment $x_i = 1$ by the transition $a^i[0] \to q_i$ and the assignment $x_i = 0$ by $a^i[0] \to q_\perp$.

Clearly, the sizes of d, Q and R are polynomial in the size of the initial problem and we have $d \in \mathcal{L}(\mathcal{A})$ iff there is an assignment to the x_i's satisfying the 3-SAT problem. To conclude, the unique type 3 rule of $\mathcal{A}$ states that d is accepted iff there is an assignment of the x_i's satisfying all the clauses $C_1, \ldots, C_m$. □

5.7 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 Section 3.2 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** $c \to q \in R$ **then** $Q_M = Q_M \cup \{q\}$

if $a[q'] \to q \in R$ 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*

Since Algorithm 1 builds an increasing sequence of subsets of the (finite set of) states of the automaton the procedure terminates.

Proposition 11 *A state q is marked by Algorithm 1, that is $q \in Q_M$, if and only if there exists a document d such that $d \to q$.*

Proof Assume $d \to q$, we prove that q is marked by Algorithm 1 by induction on the derivation. The proof of the converse property is even simpler: we build for each state marked by Algorithm 1 a *witness* d that is a document such that $d \to q$. □

We can also give a result on the complexity of this algorithm. Assume $\mathcal{A} = \langle Q, Q_{\text{fin}}, R \rangle$ is an automaton such that $Cost_{\mathcal{A}}$ bounds the time complexity to decide the constraints of $\mathcal{A}$, i.e., for any Q' subset of Q, the satisfiability of the restriction to Q' of the constraints occurring in the type 3 rules of $\mathcal{A}$ can be tested in $O(Cost_{\mathcal{A}})$.

Proposition 12 *Given an automaton $\mathcal{A}$, to decide whether $\mathcal{L}(\mathcal{A})$ is empty or not can be done in time $O(|Q| \cdot |R| \cdot Cost_{\mathcal{A}})$.*

A linear complexity bound holds 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.

5.8 Splitting an Automaton

We conclude this section on SA with the definition of constructions that allow the modification of an automaton while preserving determinacy and the set of recognized documents. In each of these transformations the goal is to single out a set of states that distinguish terms based on some auxiliary condition: either the terms are elements of the form $a[d]$; or they match a given a regular expression; or they satisfy a given counting constraint.

Splitting states matching elements of the form $a[d]$. Assume $\mathcal{A} =_{\text{def}} \langle Q, Q_{\text{fin}}, R\rangle$ is a complete and deterministic sheaves automaton. Let Q' be a subset of Q and a be some given tag name. We want to single out the terms reaching a state q in $\mathcal{A}$ such that the last rule used in the transition is $a[q'] \to q$ with $q' \in Q'$. In the particular case where $Q' = Q_{\text{fin}}$, this means isolating the states reached by terms $a[d]$ such that d is accepted by $\mathcal{A}$. This construction is used in the proof of the definability theorem to build an automaton accepting the models of $a[D]$ from an automaton accepting the models of D.

Assume $Q = \{q_1, \ldots, q_p\}$. We define a new automaton $\mathcal{A} \ltimes a[Q']$ (called $\mathcal{A}$ split by $a[Q']$) with states $Q \cup \{\bar{q}_1, \ldots, \bar{q}_p\}$ such that $a[d] \to \bar{q}_i$ in $\mathcal{A} \ltimes a[Q']$ iff $a[d] \to_{\mathcal{A}} q_i$ for $i \in 1..p$ and $d \to_{\mathcal{A}} q_j$ with $q_j \in Q'$. The set of rules of the automaton $\mathcal{A} \ltimes a[Q']$ is as follows:

- for each rule $c \to q$ of $\mathcal{A}$ the rule $c \to q$ is in $\mathcal{A} \ltimes a[Q']$,
- for each rule $a[q'] \to q$ of $\mathcal{A}$ if $q' \in Q'$ then the rules $a[q'] \to \bar{q}$ and $a[\bar{q}'] \to \bar{q}$ are in $\mathcal{A} \ltimes a[Q']$, otherwise the rules $a[q'] \to q$ and $a[\bar{q}'] \to q$ are in $\mathcal{A} \ltimes a[Q']$,
- for each rule $\phi(N_1, \ldots, N_p) \vdash Reg(q_1, \ldots, q_p) \to q$ of $\mathcal{A}$ the rule $\phi' \vdash Reg' \to q$ is in $\mathcal{A} \ltimes a[Q']$, where $Reg' =_{\text{def}} Reg((q_1 \mid \bar{q}_1), \ldots, (q_p \mid \bar{q}_p))$ and $\phi' =_{\text{def}} \phi(N_1 + N'_1, \ldots, N_p + N'_p)$.

The set of final states of $\mathcal{A} \ltimes a[Q']$ depends on the application that motivates the splitting.

The automaton $\mathcal{A} \ltimes a[Q']$ is deterministic. We prove this property by contradiction, the only difficult case being type (3) rules. Assume w is a sequence of states in $Q \cup \{\bar{q}_1, \ldots, \bar{q}_p\}$ that satisfies two distinct type (3) rules of $\mathcal{A} \ltimes a[Q']$. Hence the application mapping $\bar{q}_i$ to q_i for all $i \in 1..p$ in w yields a sequence satisfying two distinct type (3) rules of $\mathcal{A}$, which contradicts the fact that $\mathcal{A}$ is deterministic.

Proposition 13 *Assume $\mathcal{A} = \langle Q, Q_{fin}, R\rangle$ is a complete deterministic sheaves automaton with states $Q = \{q_1, \ldots, q_p\}$. The following equivalences hold:*

(i) $a[d] \to_{\mathcal{A} \ltimes a[Q']} \bar{q}$ iff $d \to_{\mathcal{A}} q'$ and $q' \in Q'$ and $a[q'] \to q \in R$.
(ii) $a[d] \to_{\mathcal{A} \ltimes a[Q']} q$ iff $d \to_{\mathcal{A}} q'$ and $q' \notin Q'$ and $a[q'] \to q \in R$.
(iii) Assume $n \geqslant 2$ then $e_1 \cdot \ldots \cdot e_n \to_{\mathcal{A} \ltimes a[Q']} q$ iff $e_1 \cdot \ldots \cdot e_n \to_{\mathcal{A}} q$.

Proof We only consider case (iii). The proof is by induction on the term $d =_{\text{def}} e_1 \cdot \cdots \cdot e_n$, where $n \geqslant 2$. Assume $d \to_{\mathcal{A} \ltimes a[Q']} q$. By definition of $\mathcal{A} \ltimes a[Q']$ and induction hypothesis we have for all $i \in 1..n$ that $e_i \to_{\mathcal{A} \ltimes a[Q']} \bar{q}_i$ or $e_i \to_{\mathcal{A} \ltimes a[Q']} q_i$ iff $e_i \to_{\mathcal{A}} q_i$. Let w' be the sequence obtained by concatenating the states of

$\mathcal{A} \ltimes a[Q']$ obtained from e_1 up to e_n and w be the sequence $q_1 \cdot \ldots \cdot q_n$ obtained by mapping $\bar{q}_i$ to q_i in the sequence w'. The last rule used in the transition $d \rightarrow_{\mathcal{A} \ltimes a[Q']} q$ is necessarily a type (3) rule $\phi' \vdash Reg' \rightarrow q$ with $q \in Q$ such that $w' \in Reg'$ and $\models \phi'\#(w')$. This rule corresponds to a rule $\phi \vdash Reg \rightarrow q$ in R and, by definition of ϕ' and Reg', we have that $w \in Reg$ and $\models \phi\#(w)$. Therefore $d \rightarrow_{\mathcal{A}} q$ as needed. The proof for the converse property is similar. □

A corollary of Proposition 13 is that $\mathcal{A} \ltimes a[Q']$ is complete.

Splitting states according to a regular language. Assume $\mathcal{A} =_{\text{def}} \langle Q, Q_{\text{fin}}, R\rangle$ is a complete and deterministic sheaves automaton with states $Q = \{q_1, \ldots, q_p\}$. Let Reg be a regular expression on Q. We define a new automaton $\mathcal{A}_{Reg}$ with states $Q \cup \{\bar{q}_1, \ldots, \bar{q}_p\}$ such that $d \rightarrow_{\mathcal{A}_{Reg}} \bar{q}$ iff $d = e_1 \cdot \ldots \cdot e_n \rightarrow_{\mathcal{A}} q$, for all $i \in 1..n$, $e_i \rightarrow_{\mathcal{A}} q_i$ and $q_1 \cdot \ldots \cdot q_n \in Reg$. This construction is used in the proof of the definability theorem to build an automaton accepting the models of $Reg(E_1, \ldots, E_k)$ from automata accepting the models of $(E_i)_{i \in 1..k}$. The set of final states of $\mathcal{A}_{Reg}$ is the set $\{\bar{q}_i \mid q_i \in Q_{\text{fin}}\}$. The set of rules is as follows:

- for each rule $c \rightarrow q$ of $\mathcal{A}$ the rule $c \rightarrow q$ is in $\mathcal{A}_{Reg}$,
- for each rule $a[q'] \rightarrow q$ of $\mathcal{A}$ the rules $a[q'] \rightarrow q$ and $a[\bar{q}'] \rightarrow q$ are in $\mathcal{A}_{Reg}$,
- for each rule $\phi_o \vdash Reg_o \rightarrow q$ of $\mathcal{A}$ the rules $\phi_o \vdash Reg_o \cap Reg \rightarrow \bar{q}$ and $\phi_o \vdash Reg_o \cap \overline{Reg} \rightarrow q$ are in $\mathcal{A}_{Reg}$.

By construction $\mathcal{A}_{Reg}$ is deterministic.

Proposition 14 *Assume $\mathcal{A} = \langle Q, Q_{fin}, R\rangle$ is a complete and deterministic sheaves automaton with state $Q = \{q_1, \ldots, q_p\}$. The following equivalences hold:*

(i) $a[d] \rightarrow_{\mathcal{A}} q$ *iff* $a[d] \rightarrow_{\mathcal{A}_{Reg}} q$
(ii) *Assume* $d = e_1 \cdot \ldots \cdot e_n$ *with* $n \geqslant 2$ *then* $d \rightarrow_{\mathcal{A}_{Reg}} \bar{q}$ *iff* $d \rightarrow_{\mathcal{A}} q$ *and* $q_{i_1} \cdot \ldots \cdot q_{i_n} \in Reg$ *where* $e_j \rightarrow_{\mathcal{A}} q_{i_j}$ *for all* $j \in 1..n$.
(iii) *Assume* $d = e_1 \cdot \ldots \cdot e_n$ *with* $n \geqslant 2$ *then* $d \rightarrow_{\mathcal{A}_{Reg}} q$ *iff* $d \rightarrow_{\mathcal{A}} q$ *and* $q_{i_1} \cdot \ldots \cdot q_{i_n} \in \overline{Reg}$ *where* $e_j \rightarrow_{\mathcal{A}} q_{i_j}$ *for all* $j \in 1..n$.

Proof We only consider cases (ii) and (iii). The proof is by induction on the term $d =_{\text{def}} e_1 \cdot \cdots \cdot e_n$, where $n \geqslant 2$. Assume $e_j \rightarrow_{\mathcal{A}} q_{i_j}$ for all $j \in 1..n$ and let w be the sequence $q_{i_1} \cdot \ldots \cdot q_{i_n}$. Since Reg and $\overline{Reg}$ are mutually exclusive and cover all possible cases we have either $w \in Reg$ or $w \in \overline{Reg}$. If $w \in Reg$ then we are in case (ii) and $d \rightarrow_{\mathcal{A}_{Reg}} \bar{q}$ iff $d \rightarrow_{\mathcal{A}} q$. If $w \notin Reg$ then we are in case (iii) and $d \rightarrow_{\mathcal{A}_{Reg}} q$ iff $d \rightarrow_{\mathcal{A}} q$. □

A corollary of Proposition 14 is that $\mathcal{A}_{Reg}$ is complete.

Splitting states according to a counting constraint. Assume $\mathcal{A} =_{\text{def}} \langle Q, Q_{\text{fin}}, R\rangle$ is a complete and deterministic sheaves automaton with states $Q = \{q_1, \ldots, q_p\}$. We can adapt the previous construction to the case of counting constraints. Let $\varphi(\mathbf{N})$ be a Presburger formula with free variables $N_1, \ldots, N_p$. We define a new automaton $\mathcal{A}_\varphi$ with states $Q \cup \{\bar{q}_1, \ldots, \bar{q}_p\}$ such that $d \rightarrow \bar{q}$ in $\mathcal{A}_\varphi$ iff $d = e_1 \cdot \ldots \cdot e_n$

and $\models \varphi\#(q_{i_1}\cdot\ldots\cdot q_{i_n})$ where $e_j \to_{\mathcal{A}} q_{i_j}$ for all $j \in 1..n$. This construction is used in the proof of the definability theorem to build an automaton accepting the models of $\exists \mathbf{N} : \varphi : (E_1, \ldots, E_k)$ from automata accepting the models of $(E_i)_{i\in 1..k}$. The final states of $\mathcal{A}_\varphi$ is the set $\{\bar{q}_i \mid q_i \in Q_{\text{fin}}\}$. The set of rules is as follows:

- for each rule $c \to q$ of $\mathcal{A}$ the rule $c \to q$ is in $\mathcal{A}_\varphi$,
- for each rule $a[q'] \to q$ of $\mathcal{A}$ the rules $a[q'] \to q$ and $a[\bar{q}'] \to q$ are in $\mathcal{A}_\varphi$,
- for each rule $\phi_o \vdash Reg_o \to q$ of $\mathcal{A}$ the rules $\phi_o \wedge \varphi \vdash Reg_o \to \bar{q}$ and $\phi_o \wedge \neg\varphi \vdash Reg_o \to q$ are in $\mathcal{A}_\varphi$.

By construction $\mathcal{A}_\varphi$ is deterministic.

Proposition 15 *Assume $\mathcal{A} = \langle Q, Q_{fin}, R\rangle$ is a complete and deterministic sheaves automaton with state $Q = \{q_1, \ldots, q_p\}$. The following equivalences hold:*

(i) $a[d] \to_{\mathcal{A}} q$ *iff* $a[d] \to_{\mathcal{A}_\varphi} q$

(ii) *Assume* $d = e_1 \cdot \ldots \cdot e_n$ *with* $n \geqslant 2$ *then* $d \to_{\mathcal{A}_\varphi} \bar{q}$ *iff* $d \to_{\mathcal{A}} q$ *and* $\models \varphi\#(q_{i_1} \cdot \ldots \cdot q_{i_n})$ *where* $e_j \to_{\mathcal{A}} q_{i_j}$ *for all* $j \in 1..n$.

(iii) *Assume* $d = e_1 \cdot \ldots \cdot e_n$ *with* $n \geqslant 2$ *then* $d \to_{\mathcal{A}_\varphi} q$ *iff* $d \to_{\mathcal{A}} q$ *and* $\models \neg\varphi\#(q_{i_1} \cdot \ldots \cdot q_{i_n})$ *where* $e_j \to_{\mathcal{A}} q_{i_j}$ *for all* $j \in 1..n$.

Proof The proof is similar to the proof of Proposition 14. □

A corollary of Proposition 15 is that $\mathcal{A}_\varphi$ is complete.

6 Results on the Tree Logic and on XML Schema

In this section we show that Sheaves automata provide a powerful tool to get decidability results for both SL and WXS schema. In this latter case, we show that separated automata yields more efficient procedures.

6.1 Decidability of SL

The basic idea is to associate to each formula of SL a deterministic sheaves automaton that accepts the models of the formula.

Theorem 1 (Definability) *For each formula D of SL, we can construct a deterministic, complete, sheaves automaton $\mathcal{A}_D$ accepting the models of D.*

Proof The proof is by structural induction on the formula D.

(Base Case) To obtain deterministic complete automata for `Datatype` or $True$ is straightforward.

(Case D is the formula $a[D']$) By induction hypothesis there exists a deterministic complete automaton $\mathcal{A}_{D'} = \langle Q_{D'}, Q^F_{D'}, R_{D'}\rangle$ accepting the terms satisfying D'.

We split $\mathcal{A}_{D'}$ with the set of states $Q^F_{D'}$ and the tag a as in Section 5.8. Let $\mathcal{B}$ be the automaton $\mathcal{A}_{D'} \ltimes a[Q^F_{D'}]$. We define the final states of $\mathcal{B}$ to be the states $\bar{q}$ (we follow the notations of Section 5.8) such that q' is a final state of $\mathcal{A}_{D'}$ and

there is a rule $a[q'] \to q$ in $R_{D'}$. By Proposition 13, we have $d \to_{\mathcal{B}} \bar{q}$ iff $d = a[d']$ with $d' \to_{\mathcal{A}_{D'}} q'$. Therefore the automaton $\mathcal{B}$ accepts the set of terms $a[d']$ such that $d' \models D'$, as needed.

(Case D is a regular formula $Reg(E_1, \ldots, E_p)$) For simplicity we assume that the regular expression $Reg(E_1, \ldots, E_p)$ may only match sequences of at least two elements. If this is not the case, we replace $Reg(E_1, \ldots, E_p)$ by a finite disjunction of regular expressions which are either ϵ, or an element formula E_i or a regular expression representing sequences of length at least two. We can also assume that the formulas $E_1, \ldots, E_p$ are pairwise disjoint by proposition 2.

By induction hypothesis, for each $i \in 1..p$ there is a deterministic complete automaton $\mathcal{A}_i$ accepting the models of E_i. Let $\mathcal{A}_\times$ be the product of the $\mathcal{A}_i$. It is a deterministic complete automaton and, by construction, a state of $\mathcal{A}_\times$ is a p-tuple $(q_1, \ldots, q_p)$ and $d \to_{\mathcal{A}_\times} (q_1, \ldots, q_p)$ iff $d \to_{\mathcal{A}_i} q_i$ for $i \in 1..p$. Since the E_i are pairwise disjoint, it is not possible to reach a state $(q_1, \ldots, q_n)$ containing two indices i, j such that q_i and q_j are final in E_i and E_j. These states can be safely removed from the set of states of $\mathcal{A}_\times$. We say that a state $\mathcal{Q} = (q_1, \ldots, q_p)$ of $\mathcal{A}_\times$ is final for E_i iff q_i is final for $\mathcal{A}_i$ and q_j is not final for all $j \neq i$ (by the previous remark, a state is final for one E_i at most). If the final states for E_i are $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$, we denote by $\text{Fin}(E_i)$ the regular expression $\mathcal{Q}_1 \mid \ldots \mid \mathcal{Q}_n$.

Let Reg_D be the regular expression $Reg(\text{Fin}(E_1), \ldots, \text{Fin}(E_p))$. By construction of $\mathcal{A}_\times$, $d = e_1 \cdot \ldots \cdot e_n \models Reg(E_1, \ldots, E_p)$ iff $e_i \to_{\mathcal{A}_\times} \mathcal{Q}_i$ such that $\mathcal{Q}_i$ is final for some E_{j_i} for $i \in 1..n$, and $\mathcal{Q}_1 \cdot \ldots \cdot \mathcal{Q}_n \in Reg_D$.

Let $\mathcal{A}_D$ be the deterministic and complete automaton obtained by splitting $\mathcal{A}_\times$ along Reg_D. By Proposition 14 this automaton accepts the documents $e_1 \cdot \ldots \cdot e_n$ such that $e_i \to \mathcal{Q}_{l_i}$ with $\mathcal{Q}_{l_i}$ final for some E_{j_i} (hence $e_i \models E_{j_i}$) and such that $\mathcal{Q}_{l_1} \cdot \ldots \cdot \mathcal{Q}_{l_n} \in Reg_D$, that is, $d \models Reg(E_1, \ldots, E_p)$, as needed.

(Case D is a counting formula $\exists \mathbf{N} : \phi(\mathbf{N}) : N_1\, E_1 \ \& \ \ldots \ \& \ N_p\, E_p$.) We can assume that the formulas $E_1, \ldots, E_p$ are pairwise disjoint by Proposition 2. By induction hypothesis, for each $i \in 1..p$, there is a deterministic complete automaton $\mathcal{A}_i$ accepting the models of E_i. As in the previous case we construct the product automaton $\mathcal{A}_\times$ and we say that a state $(q_1, \ldots, q_p)$ of $\mathcal{A}_\times$ is final for E_i iff q_i is a final state of $\mathcal{A}_i$ and q_j is not final for E_j for all $j \neq i$.

Assume $\mathcal{Q}_1, \ldots, \mathcal{Q}_m$ are the states of $\mathcal{A}_\times$. For all $i \in 1..p$ we denote by F_i the set of indices j such that $\mathcal{Q}_j$ is final for E_i and we denote by NF the set of indices j such that $\mathcal{Q}_j$ is not final for any E_i ($i \in 1..n$). We use the integer variable M_j to denote the number of occurrences of the state $\mathcal{Q}_j$ in a sequence. Let $\phi_D(M_1, \ldots, M_m)$ be the Presburger constraint $\phi(\Sigma_{j \in F_1} M_j, \ldots, \Sigma_{j \in F_p} M_j)$ $\wedge \bigwedge_{j \in NF}(M_j = 0)$.

Let $\mathcal{A}_D$ be the deterministic and complete automaton obtained by splitting $\mathcal{A}_\times$ along ϕ_D. By Proposition 15 this automaton accepts the documents $d = e_1 \cdot \ldots \cdot e_n$ such that $e_i \to \mathcal{Q}_{l_i}$ with $l_i \in 1..m$ and such that $W \in \phi_D$ where $W = \mathcal{Q}_{l_1} \cdot \ldots \cdot \mathcal{Q}_{l_n}$. Therefore we necessarily have $\mathcal{Q}_{l_i}$ final for some E_{j_i} (hence $e_i \models E_{j_i}$) and, if n_k is the number of states final for E_k in W, we have $\models \phi(n_1, \ldots, n_p)$. Hence $d \models \exists \mathbf{N} : \phi(\mathbf{N}) : N_1\, E_1 \ \& \ \ldots \ \& \ N_p\, E_p$, as needed.

(Case D is a formula $D_1 \vee D_2$ or $\neg D'$) Given deterministic complete automaton for D_1, D_2, D', the constructions given in Section 5 provide an immediate procedure to build a deterministic and complete automata for D. □

As a direct corollary of Theorem 1 and Propositions 9 and 12, we obtain key results on the decidability and on the complexity of the sheaves logic. Let $|Q(\mathcal{A}_D)|$ be the number of states of the SA associated to D.

Theorem 2 (Decidability) *The logic SL is decidable.*

Theorem 3 (Model Checking) *For any document d and formula D the problem $d \models D$ is decidable in time $O(|d| \cdot |R_{\mathcal{A}_D}| \cdot Cost(|Q(A_D)|, |d|))$ where $\mathcal{A}_D$ is the automaton accepting the models of D and $R_{\mathcal{A}_D}$ is the set of rules of $\mathcal{A}_D$.*

6.2 Decidability Results for WXS Schema

WXS definitions are simpler than SL formulas since they do not involve counting constraints or logical connectives. On the other hand they are more complex since they allow recursive definitions. We prove that we can relate a WXS definition to a *separated* sheaves automaton accepting the well-typed documents. This result yields the decidability of basic validation problems like type-checking, type inclusion and testing if a schema is inhabited. (Testing for type inclusion is a crucial operation when typing nested pattern-matching expressions in functional languages like XDuce and amounts to deciding whether the set of documents typed by the difference of two schema is empty.)

Proposition 16 *For every well-formed type declaration $a[X]$ with $\mathcal{S}$, we can build a complete separated sheaves automaton $\mathcal{A}$ that recognizes the set $\{d \mid \mathcal{S} \vdash d : X\}$ of documents with type X. Furthermore, the size of the automaton $\mathcal{A}$ is linear in the size (number of symbols) of the environment $\mathcal{S}$.*

Proof Assume $\mathcal{S}$ is $\{X_1 = T_1, \ldots, X_n = T_n\}$. By introducing new variables, we can always assume that any term T_i is either of the form $a[X_j]$ or is a composition involving only element formulas of the form $a[X_j]$. The separated sheaves automaton $\mathcal{A} = \langle Q, Q_F, R\rangle$ is defined as follows:

(States) We introduce a state q_{X_i} for every variable X_i in $\mathcal{S}$ ($i \in 1..n$) and a state $q_{a[X_i]}$ for every element formula $a[X_i]$ occurring in a right-hand side T_j of $\mathcal{S}$ ($j \in 1..n$). We also consider a state $q_{\texttt{Datatype}}$ for every primitive datatype used in $\mathcal{S}$.

(Final states) The only final state is q_X.

(Rules) We assume that the type 1 rule $c \to q_{\texttt{Datatype}}$ is in $\mathcal{A}$ if c is a constant of type `Datatype`. The set of rules in $\mathcal{A}$ is the smallest set such that:

- for each equation $X = a[Y]$ in $\mathcal{S}$ the type 2 rule $a[q_Y] \to q_X$ is in $\mathcal{A}$.
- for each pair (a, X) such that $a[X]$ occurs in $\mathcal{S}$, the type 2 rule $a[q_X] \to q_{a[X]}$ is in $\mathcal{A}$.

- for each equation $X = Reg(E_1, \ldots, E_p)$ in $\mathcal{S}$, the constrained rule $True \vdash Reg(q_{E_1}, \ldots, q_{E_p}) \to q_X$ is in $\mathcal{A}$.
- for each equation $X = E_1 \ \& \ \ldots \ \& \ E_p$ in $\mathcal{S}$, the counting rule $\bigwedge_{j \in 1..k} \#q_{E_{i_j}} = n_j \vdash All_Q \to q_X$ is in $\mathcal{A}$, where $E_{j_1}, \ldots, E_{j_k}$ is the sequence of distinct element formulas in $E_1, \ldots, E_p$ and n_j is the number of occurrences of E_{i_j} in $E_1, \ldots, E_p$. In the special case where element formulas $E_1, \ldots, E_p$ are all distinct, the counting constraint of the type 3 rule is simply $\bigwedge_{i \in 1..p} \#q_{E_i} = 1$.
- for each equation $X = AnyT$ in $\mathcal{S}$, the rules $c \to q_X$ and $a[q] \to q_X$ and $True \vdash All_Q \to q_X$ are in $\mathcal{A}$ for all states $q \in Q$ and constants c.

By construction the automaton $\mathcal{A}$ is separated and the size of the automaton is linear in the size of the type declaration. The proposition follows by proving that for every equation $X = T$ in $\mathcal{S}$, we have $d \to q_X$ if and only if $\mathcal{S} \vdash d : X$ and $\mathcal{S} \vdash d : T$.

(Proof of the $\Rightarrow$ direction) The proof is by case analysis on the last rule of the derivation $d \to q_X$. We do not consider the case where $T = AnyT$ since it trivially entails that $\mathcal{S} \vdash d : T$.

If the last rule is a type 1 rule of the form $c \to q$ then d is a constant of the primitive type `Datatype`, $T =$ `Datatype` and $q = q_{\texttt{Datatype}}$. Hence $\mathcal{S} \vdash d : X$ and $\mathcal{S} \vdash d : T$ as needed.

If the last rule is a type 2 rule of the form $a[q_Y] \to q_X$, then $d = a[d']$ with $d' \to q_Y$ and $T = a[Y]$. By induction hypothesis, $\mathcal{S} \vdash d' : Y$ and therefore $\mathcal{S} \vdash d : X$ and $\mathcal{S} \vdash d : T$.

If the last rule of the derivation is a regular rule $True \models Reg(q_{E_1}, \ldots, q_{E_p}) \to q_X$ then $T = Reg(E_1, \ldots, E_p)$ and we have $d = e_1 \cdot \ldots \cdot e_m$ where $e_i \to q_{E_{j_i}}$ and $q_{E_{i_1}} \cdot \ldots \cdot q_{E_{i_m}} \in Reg(q_{E_1}, \ldots, q_{E_p})$. By construction of $\mathcal{A}$, $e_i = a_i[d_i]$ where $d_i \to q_{Y_{j_i}}$ and $E_{j_i} = a_i[Y_{j_i}]$. Hence, by induction hypothesis, $\mathcal{S} \vdash d_i : X_{j_i}$ for $i \in 1..m$, which entails that $\mathcal{S} \vdash d : X$ and $\mathcal{S} \vdash d : T$.

If the last rule of the derivation is a counting rule $\bigwedge_{j \in 1..k} \#q_{E_{i_j}} = n_j \vdash All_Q \to q_T$. Then $T = E_1 \ \& \ \cdots \ \& \ E_p$ and we have $d = e_1 \cdot \ldots \cdot e_p$ such that there is a permutation σ of $1..p$ with $e_i \to q_{E_{\sigma(i)}}$ for $i \in 1..p$. By definition of the rules of $\mathcal{A}$, $e_i = a_i[d_i]$ where $d_i \to q_{X_{j_i}}$ and $E_{\sigma(i)} = a_i[X_{\sigma(i)}]$. By induction hypothesis, $\mathcal{S} \vdash d_i : X_{\sigma(i)}$ for $i = 1..p$, which entails that $\mathcal{S} \vdash d : X$ and $\mathcal{S} \vdash d : T$.

(Proof of the $\Leftarrow$ direction) The proof of the converse direction is by case analysis on the last rule of the derivation of $\mathcal{S} \vdash d : T$.

Assume $T = a[Y]$ and the last rule is of the form $\mathcal{S} \vdash d' : Y$ entails $\mathcal{S} \vdash d : a[Y]$. Hence $d = a[d']$ and, by induction hypothesis, we have that $d' \to q_Y$. By construction of $\mathcal{A}$ we obtain $d = a[d'] \to q_X$ as needed. The case $T = a[Y]?$ and $d = a[d']$ is similar.

Assume $T = a[Y]?$ and $d = \epsilon$. Hence ϵ satisfies the regular expression T and we have $\epsilon \to q_X$ by definition of $\mathcal{A}$. The case where $T =$ `Datatype` and $d = c$ is a constant of type T is similar.

Assume $T = AnyT$. In this case, by construction of $\mathcal{A}$, we have $d \to q_X$ for every document d.

Assume $T = Reg(E_1, \ldots, E_p)$. Hence $d = e_1.\cdots.e_m$ and the last rule is of the form $e_j : E_{i_j}, (i \in 1..k), E_{i_1} \cdot \ldots \cdot E_{i_k} \in Reg(E_1, \ldots, E_n)$ entails $\mathcal{S} \vdash d : Reg(E_1, \ldots, E_n)$. By induction hypothesis, $d_i \to q_{X_i}$ for $i \in 1..m$ and, by construction of $\mathcal{A}$, $a_i[d_i] \to q_{a_i[X_i]}$ for all $i \in 1..m$. Hence, by definition of $\mathcal{A}$ we have $d \to q_X$ as needed. The case where T is an interleaving composition is similar. □

The previous proof can be easily enhanced to deal with richer schemas. For example, we could handle `all` groups extended with Presburger constraints on repetition operators, such as the (fictional) term $\exists N.(a[\epsilon]\{0, N\} \ \& \ b[\epsilon]\{N, \infty\})$ for instance. Note also that the proof of Proposition 16 does not rely on the *Unique Particle Attribution Rule* or the *Consistent Declaration Rule* of WXS that we mentioned in Section 2.2.

From Proposition 16, we obtain several decidability properties on schema, as well as automata-based decision procedures. For instance, we can define the intersection and difference of two schema (that are not necessarily well-formed schema).

Theorem 4 (XML Typing) *Given a type declaration $a[X]$ with $\mathcal{S}$ and a document d the problem $\mathcal{S} \vdash d : a[X]$ is in NP.*

Proof By Proposition 16, there is a separated automata $\mathcal{A}$ that recognizes documents d' such that $\mathcal{S} \vdash d' : X$. Furthermore, the automaton $\mathcal{A}$ has a size linear in the schema definition which proves, by Proposition 10, that type-checking is in NP. □

Theorem 5 (Satisfaction) *Given a type declaration $a[X]$ with $\mathcal{S}$, the problem of finding whether there exists a document d such that $\mathcal{S} \vdash d : a[X]$ is in PTIME.*

Proof Same as in the proof of Theorem 4, but using Proposition 12. By Proposition 16, there is a separated automata $\mathcal{A}$ that recognizes documents d' such that $\mathcal{S} \vdash d' : X$. Furthermore, the automaton $\mathcal{A}$ has a size linear in the schema definition which proves, by Proposition 12, that testing if a schema is inhabited is polynomial. □

Theorem 6 (Subtyping) *Given a type declaration $a[X]$ with $\mathcal{S}$ and two schemas T_1, T_2 (using only schema variables in $\mathcal{S}$), it is decidable to check whether every document of type T_1 is also of type T_2.*

Proof By Proposition 16, there is a separated automata $\mathcal{A}$ corresponding to the declaration $\mathcal{S} \cup \{Y_1 = T_1, Y_2 = T_2\}$. Since separated automata are determinizable (see Proposition 16), we can check the emptiness of the language $\mathcal{L}(\mathcal{A}_{T_1}) \cap \overline{\mathcal{L}(\mathcal{A}_{T_2})}$, where $\mathcal{A}_{T_i}$ is the automata obtained from $\mathcal{A}$ by setting the set of final states to $\{q_{Y_i}\}$. By construction, the intersection is empty iff the type T_1 is included in the type T_2. □

7 Related Work

The contributions of this paper are a new class of tree automata for unranked, ordered trees with counting constraints and a new tree logic for unranked trees. In this section, we briefly report on related work for tree automata.

Tree automata for unranked, ordered trees have been introduced by Thatcher [30, 29], and also by Pair and Quéré [27]. All the good closure and decidability properties of regular tree automata have been extended to the unranked case. Tree automata for unranked trees have been used in connection with schema transformation by Murata [19], under the name *hedge automata*. This work is at the basis of the implementation of RELAX-NG [6], an alternative proposal to WXS.

Automata for unranked, unordered trees were studied by Courcelle who also extended monadic second-order (MSO) logic by some counting constraints to capture the recognizable languages [9]. Regular languages of terms with an equational theory modulo associativity-commutativity are studied in the context of *regular AC-equational languages* [25] (where flattened terms correspond to unranked, unordered trees).

Tree automata with constraints is an old idea (see [8] for a survey of equational constraints). Counting constraints have been used by Niehren and Podelski [24] for features trees (a special case of unranked, unordered trees) in the framework of knowledge representation. The class of tree languages that they define is closed under boolean operations and can be related with a notion of regular expressions that use counting constraints (these counting constraints are less general than the one used in our work and are not combined with regular word constraints). More general counting constraints appear in [17], for an application to automated reasoning. Klaedtke and Ruess consider automata for infinite trees with an accepting condition that depends on one global Presburger formula [10]. Automata for unranked, unordered trees with MSO constraints on transitions have been used by Colcombet [7]. More complex equational constraints are studied in [16]. 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 studies are not directly concerned with schema languages and are based on ordered content models.

Query languages for unranked, unordered trees have been proposed by Cardelli and Ghelli [5] as an extension of the static fragment of ambient logic[4]. A main difference between TQL and SL is that SL formulas may express properties on both ordered and unordered sets of trees. In contrast, our logic lacks some of the operators found in TQL, like quantification over tag names, which could be added at the cost of some extra complexity. Kupferman, Sattler and Vardi [11] study a μ-calculus with graded modalities where it is possible to express that a node has at least n successors satisfying a given property. But the number n may only be a constant.

The application of tree automata to XML has been widely investigated [23], mainly with the goal of devising type systems and type-checking algorithms or as a basis for query languages. More crucially, automata theory is mentioned in several

places in the XML specifications, principally to express restrictions on DTD and schema in order to obtain almost linear complexity for simple operations.

A pioneering work on typed transformation languages for XML is the XDuce system of Pierce, Hosoya *et al.* [15], a typed functional language with extended pattern-matching operators for XML. 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 automata. For applications to schema languages, an important reference is the work of Murata [20] on *hedge automata* that have been used for querying XML documents (together with an extension to two-way automata). Another large body of work is concerned with the problem of finding more efficient algorithms or study the expressive power of regular languages (connecting these languages to monadic second-order logic). For ranked and unranked trees, Neven and Schwentick have defined query automata [22] that are two-way hedge automata that select nodes in a tree according to both a state and the current function symbol. Complexity results and the relationship with monadic second-order logic are also established. Finally, extensions of monadic second order logic with Presburger constraints have been proved undecidable [21], which shows that such extensions must be carefully designed.

Independently from our work, Muscholl, Schwentick and Seidl [21] proposed a notion of tree automata for unranked trees which is very close to our definition of sheaves automata. Despite some slight differences — in their approach, counting constraints tallies all the sub-terms that can reach a given state, while we never count the same sub-term twice in our framework — the main properties of the two classes are identical. In a subsequent article [28], the authors characterize the expressive power of deterministic automata (it is possible to associate to a deterministic automaton a formula matching the set of accepted trees) and give some efficient algorithms for the computation of counting constraints. Whereas the work in [21] defines a MSO-like logic with counting constraints and recursion (but with a restricted use of negation), we define a tree logic in the spirit of TQL [5] without recursion but with full negation. An unrestricted use of negation and recursion in our framework can easily lead to inconsistencies, and a better candidate to extend SL is guarded recursion. As a matter of fact, a restricted class of sheaves automata [13] has been used to prove complexity properties of the static fragment of ambient logic, which corresponds to a kind of regular expression language for unranked, unordered trees. In this paper, 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.

8 Conclusion

Our contribution is a new class of automaton for unranked trees aiming at the manipulation of W3C XML schema. We believe it is the first contribution on applying tree automata theory to WXS that considers the `all` group. 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 [20] 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 `all` group 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 an `all` group may only appear at top-level position in a complex type 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 schema yield simple Presburger formulas. For example, we may obtain polynomial complexity by imposing that each element tag in an `all` group $a_1[S_1]$ `&` $\ldots$ `&` $a_p[S_p]$ be distinct.

To conclude, we would like to stress that 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 WXS. An advantage of using tree automata theory for this task is that it gives us complexity results on problems related to validating documents. We also hope to use our approach to define improved restrictions on schema and to give better intuition on their impact. Another advantage of using tree automata is that it suggests multiple directions for improving our tree logic. For instance, adding the capacity for the reverse traversal of a document or an extension 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 [8] 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.

Acknowledgment

The authors are most grateful to the anonymous referees for several helpful comments and fruitful suggestions about an early draft of this paper, especially on the use of separated sheaves automata.

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 A. Gordon. Anytime, anywhere: Modal logic for mobile ambients. In *Principles of Programming Languages (POPL)*. ACM Press, 2000.
5. L. Cardelli and G. Ghelli. A query language based on the ambient logic. In *European Symposium on Programming (ESOP)*, LNCS vol. 2028, pp. 1–22. Springer, 2001.
6. J. Clark and M. Makoto, editors. *RELAX-NG Tutorial*. OASIS, 2001.
7. T. Colcombet. Rewriting in the partial algebra of typed terms modulo AC. *Electronic Notes in Theoretical Computer Science*, 68, 2002.
8. H. Comon, M. Dauchet, F. Jacquemard, S. Tison D. Lugiez, and M. Tommasi. *Tree Automata on their application*. 1999. Available at `http://www.grappa.univ-lille3.fr/tata/`
9. B. Courcelle. The Monadic Second-Order Logic of Graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.
10. F. Klaedtke and H. Ruess. Monadic Second-order Logics with Cardinalities. In *International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS vol. 2719, pp. 681–696. Springer, 2003.
11. O. Kupferman, U. Sattler, and M. Vardi. The Complexity of the Graded μ-Calculus. in *International Conference on Automated Deduction (CADE)*, LNCS vol. 2392, pp. 423–437. Springer, 2002.
12. S. Dal Zilio and D. Lugiez. XML schema, tree logic and sheaves automata. Technical Report 4631, INRIA, 2002.
13. S. Dal Zilio, D. Lugiez, and C. Meyssonnier A Logic you Can Count On. In *Principles of Programming Languages (POPL)*, pp. 135-146. ACM Press, 2004.
14. M. Fischer, and M.O. Rabin. Super-exponential complexity of Presburger arithmetic. In *SIAM-AMS Symposium in Applied Mathematics*, vol. 7, pp. 27–41, 1974.
15. H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *Principles of Programming Languages (POPL)*, pp. 67–80. ACM Press, 2001.
16. D. Lugiez. Multitree automata that count. *Theoretical Computer Science*, 333:225–263, 2005.
17. D. Lugiez and J.L. Moysset. Tree automata help one to solve equational formulae in AC-theories. *Journal of Symbolic Computation*, 18(4):297–318, 1994.
18. D. Lugiez and S. Dal Zilio. Multitrees automata, Presburger's constraints and tree logics. Technical Report 08-2002, LIF, 2002.
19. M. Murata. DTD transformation by patterns and contextual conditions. In *SGML/XML*. GCA, 1997.
20. M. Murata. Extended path expression for XML. In *Principles of Database Systems (PODS)*. ACM Press, 2001.
21. A. Muscholl, T. Schwentick, and H. Seidl. Numerical document queries. In *Principle of Databases Systems (PODS)*. ACM Press, 2003.
22. F. Neven and T. Schwentick. Query automata on finite trees. *Theoretical Computer Science*, 275:633–674, 2002.
23. F. Neven. Automata theory for XML researchers. Sigmod Record, 31(3), 2002.
24. Joachim Niehren and Andreas Podelski. Feature automata and recognizable sets of feature trees. In *TAPSOFT*, LNCS vol. 668, pp. 356–375. Springer, 1993.

25. H. Ohsaki. Beyond the regularity: Equational tree automata for associative and commutative theories. In *International Workshop on Computer Science Logic (CSL)*, LNCS vol. 2142, pp. 539–553. Springer, 2001.
26. D. C. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16:323–332, 1978.
27. C. Pair and A. Quéré. Définition et étude des bilangages réguliers. *Information and Control*, 13(6):565–593, 1968.
28. H. Seidl, A. Muscholl, T. Schwentick and P. Habermehl. Counting in trees for free. In *Proceedings 31st International Colloquium on Automata, Languages, and Programming (ICALP)*, LNCS vol. 3142, pp. 1136–1149. Springer, 2004.
29. J.W. Thatcher. Characterizing derivation trees of context-free grammars through a generalization of automata theory. *Journal of Computer and System Sciences*, 1:317-322, 1967.
30. J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2:57–82, 1968.

A Omitted Proofs

We give the proofs of propositions 1, 4, 5 and 8.

A.1 Proof of Proposition 1: GDL is undecidable

We show that given a two-counter machine, there is a formula of GDL matching exactly the set of terminating computations of the machine. Since the reachability problem for two-counter machines is a well-known undecidable problem, we get that GDL is undecidable.

Two-counter machines are devices made from a finite set of states Q, some being termed final, a pair of two nonnegative counters C_1, C_2, and a transition relation $\delta \subseteq Q \times \{0,1\}^2 \times Q \times \{-1,0,1\}^2$. A configuration, $\mathbb{C}$, is a triple (q, C_1, C_2), where q is a state in Q. We say that the configuration $\mathbb{C} = (q, C_1, C_2)$ can be reduced to 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 counter i 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 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 corresponding operations on counters are described by the following formula, where we may replace the expression $C_2 > 0$ with the Presburger 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 indices $i \geqslant 1$ we have $\mathbb{C}_{i-1} \implies \mathbb{C}_i$. It is well-known that there is a fixed ("universal") two-counter machine such that it is undecidable for given input values of the counters whether there exists a computation that may reach a configuration with a final state, also called a *halting configuration*.

We encode a configuration $\mathbb{C} = (q, C_1, C_2)$ of a two-counter machine by a word in the alphabet $\Sigma = Q \cup \{a, b, c, d\}$ as follows. The encoding can be interpreted straightforwardly as an encoding on documents where we identify a letter a to an element $a[\epsilon]$ and a concatenation of words to a concatenation of documents.

$$[\![(q, C_1, C_2)]\!] \;=\; q \cdot a^{C_1} \cdot b^{C_2} \cdot q \cdot c^{C_1} \cdot d^{C_2}$$

The term a^{C_i} is the word $a \cdot \ldots \cdot a$ of length the value of the counter C_i. The redundancy in the encoding of counter values is a technical trick that will prove helpful in the construction of the formula matching the admissible sequences.

We start by defining the formula A_q that matches words of the form $[\![(q, C_1, C_2)]\!]$ for a fixed state $q \in Q$ and for arbitrary values C_1, C_2 of the counters.

$$\begin{array}{l} A_q \;=_{\text{def}}\; q \cdot a^* \cdot b^* \cdot q \cdot c^* \cdot d^* \quad \wedge \\ \qquad\qquad \exists \mathbf{N} : N_q = 2 \wedge N_a = N_c \wedge N_b = N_d : \mathbf{N} \otimes \mathbf{E} \\ \text{where } \mathbf{N} =_{\text{def}} (N_q, N_a, N_b, N_c, N_d) \\ \text{and } \mathbf{N} \otimes \mathbf{E} =_{\text{def}} N_q\, q \;\&\; N_a\, a \;\&\; N_b\, b \;\&\; N_c\, c \;\&\; N_d\, d\,. \end{array}$$

Therefore we can define the formula A_o that matches exactly sequences of machine configurations that starts from the initial configuration $\mathbb{C}_0$ and ends with an halting configuration.

$$A_o \;=_{\text{def}}\; \big(\bigvee_{q \in Q} A_q\big)^* \;\wedge\; [\![C_0]\!] \cdot \Sigma^* \;\wedge\; \Sigma^* \cdot \bigvee_{q \text{ final}} A_q\,.$$

Next, we define a formula that will distinguish valid computations from arbitrary sequences of configurations. For every transition $t = (q, x_1, x_2, q', x_1', x_2')$ in δ, we define the formula B_t that matches words of the form $q \cdot c^{C_1} \cdot d^{C_2} \cdot q' \cdot a^{C_1'} \cdot b^{C_2'}$ such that the (q', C_1', C_2') derives from (q, C_1, C_2) by the transition t.

$$B_t \;=_{\text{def}} \left(\begin{array}{l} q \cdot c^* \cdot d^* \cdot q' \cdot a^* \cdot b^* \quad \wedge \\ \quad \exists \mathbf{N}' : \left(\begin{array}{l} N_q = N_{q'} = 1 \wedge (N_c = 0 \Leftrightarrow x_1 = 0) \wedge \\ (N_d = 0 \Leftrightarrow x_2 = 0) \wedge N_a = N_c + x_1' \wedge \\ N_b = N_d + x_2' \end{array} \right) : \mathbf{N}' \otimes \mathbf{E}' \end{array} \right)$$

where $\mathbf{N}' =_{\text{def}} (N_q, N_{q'}, N_a, N_b, N_c, N_d)$
and $\mathbf{N}' \otimes \mathbf{E}' = N_q\, q \;\&\; N_{q'}\, q' \;\&\; N_a\, a \;\&\; N_b\, b \;\&\; N_c\, c \;\&\; N_d\, d\,.$

The conjunction of the formulas $A_q \cdot A_{q'}$ and $q \cdot a^* \cdot b^* \cdot B_t \cdot q' \cdot c^* \cdot d^*$ matches only words of the form $[\![(q, C_1, C_2)]\!] \cdot [\![(q', C_1', C_2')]\!]$ such that (q', C_1', C_2') derives from (q, C_1, C_2) by the transition t. Hence we can define the formula B_o that matches sequences of configurations obtained from transitions of the machine:

$$B_o \;=_{\text{def}}\; \big(\bigvee_{q \in Q} q \cdot a^* \cdot b^*\big) \cdot \big(\bigvee_{t \in \delta} B_t\big)^* \cdot \big(\bigvee_{q \in Q} q \cdot c^* \cdot d^*\big)\,.$$

Therefore the formula $A_o \wedge B_o$ matches only the valid, halting computations of the machine and, if GDL was decidable, it will be decidable to check whether the machine halts.

A.2 Proof of Proposition 4: Non-deterministic sheaves automata are strictly more expressive than deterministic ones

As in the previous proof, we identify the concatenation of elements of the form $a[\epsilon], b[\epsilon]$ to a word on the alphabet $\Sigma = \{a, b\}$. Let us consider the following language L over Σ:

$$L = \{\ w_1 \cdot w_2 \cdot w_3 \cdot w_4 \mid\ w_1, w_3 \in a^*, w_2, w_4 \in b^*, \\ \#_a w_1 = \#_b w_2 \geqslant 1,\ \#_a w_3 = \#_b w_4 \geqslant 1\ \}$$

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:

$$a \to qa_1 \qquad b \to qb_1 \qquad a \to qa_2 \qquad b \to qb_2$$
$$(\#qa_1 = \#qb_1) \wedge (\#qa_2 = \#qb_2) \vdash qa_1^* \cdot qb_1^* \cdot qa_2^* \cdot qb_2^* \to q_s$$

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 17 *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 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 $size(Reg)$ 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 $m, n \geqslant size(Reg)$, 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, if we consider a subpart of an accepted word of size greater than $size(Reg)$, then the accepting path of the automaton should contain at least one cycle. For example, in the case where $n, m > size(Reg)$ 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:

position in $t_{(n,m)}$	$p_1\ p_2$		$p_3\ p_4$	
$t_{(n,m)}$	$= a \cdot \ldots\ldots \cdot a \cdot b \cdot \ldots \cdot b$		$\cdot\, a \cdot \ldots \cdot a \cdot b \cdot \ldots\ldots \cdot b$	
	$\downarrow\ \downarrow$		$\downarrow\ \downarrow$	$\downarrow$
states reached	$q_1\ q_1$		$q_2\ q_2$	q (final)

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

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. □

A.3 Proof of proposition 5: Separated automata are determinizable

Let $\mathcal{A} = \langle Q, Q_F, R\rangle$ be a separated automaton, where $Q = \{q_1, \ldots, q_n\}$. The set of states of the deterministic automaton $\mathcal{A}_D$ is 2^Q (a state in $\mathcal{A}_D$ is a subset of Q) and the set of final states of $\mathcal{A}_D$ is $\mathcal{F} = \{\mathcal{Q} \in 2^Q \mid \exists q \in \mathcal{Q}\,.\, q \in Q_F\}$.

We start by giving some definitions and results before defining the rules of $\mathcal{A}_D$.

For $I \subseteq \{1,\ldots,n\}$, we denote by $\mathcal{Q}_I$ the set $\{q_i \mid i \in I\}$. As usual, we will construct a deterministic automaton $\mathcal{A}_D$, with states $(\mathcal{Q}_I)_{I\subseteq\{1,\ldots,n\}}$, such that a term reaches $\mathcal{Q}_I$ in $\mathcal{A}_D$ iff $\mathcal{Q}_I$ is the set of states that the term can reach in the non-deterministic automaton $\mathcal{A}$.

For regular constraints, given a constraint Reg on the alphabet Q, let Reg^D be the following regular expression on the alphabet 2^Q, obtained from Reg by substituting each state q_i by the expression $\Sigma_{i\in I}\mathcal{Q}_i =_{\text{def}} (\mathcal{Q}_{I_1} \mid \cdots \mid \mathcal{Q}_{I_k})$, where $I_1,\ldots,I_k$ are the subsets of $1..n$ containing i:

$$Reg^D =_{\text{def}} Reg\{q_1 \leftarrow \Sigma_{1\in I}\mathcal{Q}_I\}\ldots\{q_n \leftarrow \Sigma_{n\in I}\mathcal{Q}_I\} \tag{2}$$

Proposition 18 *Let $\mathcal{Q}_1,\ldots,\mathcal{Q}_m$ be elements of 2^Q. Then, $\mathcal{Q}_1\cdot\ldots\cdot\mathcal{Q}_m \in Reg^D$ iff there exist q_{i_1} in $\mathcal{Q}_1$, …, q_{i_m} in $\mathcal{Q}_m$ such that $q_{i_1}\cdot\ldots\cdot q_{i_m} \in Reg$.*

Proof the proof is by structural induction on the definition of Reg. The case $Reg = \epsilon$ is trivial.

(Case $Reg = q_i$) By definition $Reg^D = \Sigma_{i\in I}\mathcal{Q}_I$.
$\Rightarrow$ condition. A word in Reg^D is of the form $\mathcal{Q}_I$ with $i \in I$ (which means that $q_i \in \mathcal{Q}_I$). Hence there exists a state $q \in \mathcal{Q}_I$ such that $q = q_i$.
$\Leftarrow$ condition. The only word matching Reg is q_i and $q_i \in \mathcal{Q}_I$ implies $\mathcal{Q}_I \in Reg^D$, as needed.

(Case $Reg = R_1\cdot R_2$) By definition $Reg^D = R_1^D\cdot R_2^D$.
$\Rightarrow$ condition. A word $W \in Reg^D$ is of the form $W_1\cdot W_2$ with $W_i \in R_i^D$ for all $i \in \{1,2\}$. By induction hypothesis, for each occurrence of a letter $\mathcal{Q}_i$ in W_1 (resp. W_2) there exists some $q_{j_i} \in \mathcal{Q}_i$ such that the word w_1 (resp. w_2) obtained by replacing $\mathcal{Q}_i$ by q_{j_i} is in R_1 (resp. R_2). Hence $w_1\cdot w_2 \in R_1\cdot R_2$.
$\Leftarrow$ condition. Assume $w = q_{i_1}\cdot\ldots\cdot q_{i_m}$ is in Reg. Let W be a word $W = \mathcal{Q}_1\cdot\ldots\cdot\mathcal{Q}_m$ such that $q_{j_i} \in \mathcal{Q}_i$ for all $i \in 1..m$. Since w is in $R_1\cdot R_2$, we can partition w in two subwords, $w = w_1\cdot w_2$ such that $w_i \in R_i$ for all $i \in \{1,2\}$. By induction hypothesis on the expressions R_1 and R_2, the sub-words W_1, W_2 such that $W = W_1\cdot W_2$ and $|W_1| = |w_1|$ are such that $W_i \in R_i^D$ for all $i \in \{1,2\}$, as needed. The proof in the case $Reg = R^*$ is similar (we use the fact that $(R^*)^D = (R^D)^*$). $\square$

We prove a similar proposition for Presburger formulas φ with n variables. Let $I_1,\ldots,I_p$ be an enumeration of the subsets of $1..n$ and φ^D be the counting constraints in $p = 2^n$ variables defined by:

$$\varphi^D =_{\text{def}} \exists M_1^1\ldots M_p^n.\big(\textstyle\bigwedge_{i\in 1..p}(M_i = \sum_{j\in I_i} M_i^j) \wedge \varphi\big(\sum_{\{i\mid\ 1\in I_i\}} M_i^1,\ldots,\sum_{\{i\mid\ n\in I_i\}} M_i^n\big)\big) \tag{3}$$

Proposition 19 *Let $\mathcal{Q}_1,\ldots,\mathcal{Q}_m$ be elements of 2^Q. Then, $\#(Q_1\cdot\ldots\cdot Q_m) \models \varphi^D$ iff there exist q_1 in $\mathcal{Q}_1$, …, q_m in $\mathcal{Q}_m$ such that $\#(q_1\cdot\ldots\cdot q_m) \models \varphi$.*

Proof We start by proving the first implication. Let W be the word $\mathcal{Q}_1 \cdot \ldots \cdot \mathcal{Q}_m$ and assume $\#(W) \models \varphi^D$. Let M_i denote the number of occurrences of a letter $\mathcal{Q}_{I_i}$ of 2^Q in W. By definition of φ^D, there exists a decomposition $M_i = \Sigma_{j \in I_i} M_i^j$ such that $\models \varphi\big(\sum_{\{i \mid\ 1 \in I_i\}} M_i^1, \ldots, \sum_{\{i \mid\ n \in I_i\}} M_i^n\big)$.

Let w be the word $q_1 \cdot \ldots \cdot q_m$ obtained by replacing M_i^j occurrences of $\mathcal{Q}_i$ by the j^{th} letter of the alphabet Q for all $i \in 1..p, j \in I_i$. By construction $\#(w) \models \varphi$, as needed.

The proof of the converse implication is similar. Let $w = q_1 \cdot \ldots \cdot q_m$ be a word such that $\#(w) \models \varphi$ and assume W is a word of the form $\mathcal{Q}_1 \cdot \ldots \cdot \mathcal{Q}_m$ obtained from w such that $q_j \in Q_j$ for all $j \in 1..m$. (An occurrence of a letter of 2^Q in W is replaced by an occurrence of a letter $\mathcal{Q}$.)

Let M_i denotes the number of occurrences of $\mathcal{Q}_{I_i}$ in W and let M_i^j be the number of replacements of $\mathcal{Q}_{I_i}$ by q_j. By definition we have that $M_i = \sum_{j \in I_i} M_i^j$ and $\models \varphi\big(\sum_{\{i \mid\ 1 \in I_i\}} M_i^1, \ldots, \sum_{\{i \mid\ n \in I_i\}} M_i^n\big)$, as needed. □

We conclude the definition of the deterministic automaton $\mathcal{A}_D$ by defining the set of rules R_D of $\mathcal{A}_D$. The definition of the sets of type 1 and type 2 rules of $\mathcal{A}_D$ is the same as in the case of regular tree automata:

(type 1) $c \to \mathcal{Q}$ is in $\mathcal{A}_D$ if $\mathcal{Q} = \{q \in Q \mid\ \exists\, c \to q \in R\}$

(type 2) $a[\mathcal{Q}] \to \mathcal{Q}'$ is in $\mathcal{A}_D$ if $\mathcal{Q}' = \{q' \in Q \mid\ \exists q \in Q \,.\, a[q] \to q' \in R\}$

The definition of type 3 rules is more involved. Let $R_{|3} \subseteq R$ be the set of type 3 rules of $\mathcal{A}$ and let $r \in R_{|3}$ be a type 3 rule of $\mathcal{A}$. Hence r is of the form $\varphi_r \vdash \mathit{Reg}_r \to q_r$ where φ_r is a Presburger formula with n variables, and Reg_r is a regular expression on Q. Since $\mathcal{A}$ is separated, we have either $\varphi_r = \mathit{True}$ or $\mathit{Reg}_r = \mathit{All}_Q$. For every subset R' of $R_{|3}$ we build a type 3 rule $r_{R'}$ in $\mathcal{A}_D$ of the form $\varphi^D_{R'} \vdash \mathit{Reg}^D_{R'} \to \mathcal{Q}_{R'}$, where $\mathcal{Q}_{R'} = \{q_r \mid\ r \in R'\}$ (the rule is not necessarily separated). Intuitively, we have a transition $e_1 \cdot \ldots \cdot e_m \to \mathcal{Q}_{R'}$ in $\mathcal{A}_D$ that uses $r_{R'}$ as its last rule if and only if R' is the set of rules r such that $e_1 \cdot \ldots \cdot e_m \to_{\mathcal{A}} q_r$ using r as its last rule.

(type 3) for each subset R' of the set of type 3 rules of R, we define the following three sets of rules:

$$\begin{cases} R_1 = \{r \in R_{|3} \setminus R' \mid\ \varphi_r \neq \mathit{True}, \mathit{Reg}_r = \mathit{All}_Q\} & \text{(counting rules)} \\ R_2 = \{r \in R_{|3} \setminus R' \mid\ \varphi_r = \mathit{True}, \mathit{Reg}_r \neq \mathit{All}_Q\} & \text{(regular rules)} \\ R_3 = \{r \in R_{|3} \setminus R' \mid\ \varphi_r = \mathit{True}, \mathit{Reg}_r = \mathit{All}_Q\} & \text{(trivial rules)} \end{cases}$$

we use these sets to define the rule $r^D_{R'} =_{\text{def}} \varphi_{R'} \vdash \mathit{Reg}_{R'} \to \mathcal{Q}_{R'}$ such that,

(i) $\mathcal{Q}_{R'} = \{q_r \mid\ r \in R'\}$

(iia) if $R_3 \neq \emptyset$ then $\varphi_{R'} = \neg \mathit{True}$ and $\mathit{Reg}_{R'} = \emptyset$,

(iib) if $R_3 = \emptyset$ then $\begin{cases} \varphi_{R'} \equiv \bigwedge_{r \in R'} \varphi^D_r \wedge \neg(\bigvee_{r \in R_1} \varphi^D_r), \\ \mathit{Reg}_{R'} \equiv \bigcap_{r \in R'} \mathit{Reg}^D_r \cap (\overline{|_{r \in R_2} \mathit{Reg}^D_r}) \end{cases}$

where $\varphi^D_r, \mathit{Reg}^D_r$ are defined as in Propositions 18 and 19.

By construction, the constraints used in distinct transition rules are mutually exclusive, therefore the automaton $\mathcal{A}_D$ is deterministic. We conclude the proof by showing that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_D)$.

Proposition 20 *For every automata $\mathcal{A}$ the automata $\mathcal{A}_D$ accepts the same language as $\mathcal{A}$.*

Proof We prove that $d \rightarrow_{\mathcal{A}_D} \mathcal{Q}$ if and only if $\mathcal{Q} = \{q \mid\ d \rightarrow_{\mathcal{A}} q\}$. The proof is by structural induction on d. We only give the proof in the case where d is of the form $e_1 \cdot \ldots \cdot e_m$ with $m \geqslant 2$. In the cases where d is a constant c or an element $a[d']$, the proof is similar to the standard correctness proof for the determinization algorithm of tree automata.

(Proof of the $\Rightarrow$ direction) Assume $d = e_1 \cdot \ldots \cdot e_m$ $(m \geqslant 2)$ and $d \rightarrow_{\mathcal{A}_D} \mathcal{Q}$. We prove that $\mathcal{Q} = \{q \mid\ d \rightarrow_{\mathcal{A}} q\}$. By induction hypothesis we have a transition $e_j \rightarrow_{\mathcal{A}_D} \mathcal{Q}_{I_j}$ iff $\mathcal{Q}_{I_j} = \{q \mid\ e_j \rightarrow_{\mathcal{A}} q\}$. Also, by construction, the last rule used in the derivation $d \rightarrow_{\mathcal{A}_D} \mathcal{Q}$ should be a type 3 rule of the form $r^D_{R'}$.

Assume $W = \mathcal{Q}_{I_1} \cdot \ldots \cdot \mathcal{Q}_{I_m}$. By definition of $r^D_{R'}$, we have that $W \in Reg^D_{R'}$ and $\#(W) \models \varphi^D_{R'}$. As a consequence:

(i) for each $r \in R'$ and $W \in Reg^D_r$ and $\#(W) \models \varphi^D_r$,
(ii) for each $r \in R_1$, $\#(W) \not\models \varphi^D_r$ and for each $r \in R_2$, $W \notin Reg^D_r$

By Proposition 18, for each $r \in R$ we have $W \in Reg^D_r$ iff there exist $q_{i_1} \in \mathcal{Q}_{I_1}, \ldots, q_{i_m} \in \mathcal{Q}_{I_m}$ such that $q_{i_1} \cdot \ldots \cdot q_{i_m} \in Reg_r$. Since $\mathcal{A}$ is separated, for each rule r such that $Reg_r \not\equiv All_Q$, the condition φ_r is equivalent to $True$. Hence, the rule r can be applied to $d = e_1 \cdot \ldots \cdot e_m$ and $d \rightarrow q_r$, as needed.

Similarly, by Proposition 19, for each $r \in R$ we have $\#(W) \models \varphi^D_r$ iff there exist $q_{i_1} \in \mathcal{Q}_{I_1}, \ldots, q_{i_m} \in \mathcal{Q}_{I_m}$ such that such that $\#(q_{i_1} \cdot \ldots \cdot q_{i_m}) \models \varphi_r$. Since $\mathcal{A}$ is separated, for each rule r such that $\varphi_r \not\equiv True$, the regular expression Reg_r is equivalent to All_Q. Hence, the rule r can be applied to $d = e_1 \cdot \ldots \cdot e_m$ and $d \rightarrow q_r$, as needed.

(Proof of the $\Leftarrow$ direction) Assume $d = e_1 \cdot \ldots \cdot e_m$ $(m \geqslant 2)$ and $\mathcal{Q} = \{q \mid\ d \rightarrow_{\mathcal{A}} q\}$. We prove that $d \rightarrow_{\mathcal{A}_D} \mathcal{Q}$. The structure of this proof is similar to the preceding case. Since d is a sequence, if $d \rightarrow_{\mathcal{A}} q$ then the last rule used in the derivation is a type 3 rule and q is a state q_r for some type 3 rule $r \in R$. Let R be set of such rules, $R = \{r \mid\ q_r \in \mathcal{Q}\}$. Hence, with our previous notation, $\mathcal{Q} = \mathcal{Q}_R$ and we can define the subsets R_1, R_2, R_3 as above.

Let W be the word $\mathcal{Q}_{I_1} \cdot \ldots \cdot \mathcal{Q}_{I_m}$ such that $e_j \rightarrow_{\mathcal{A}_D} \mathcal{Q}_{I_j}$. By induction hypothesis, $\mathcal{Q}_{I_j} = \{q \mid\ e_j \rightarrow_{\mathcal{A}} q\}$ et $e_j \rightarrow_{\mathcal{A}_D} \mathcal{Q}_{I_j}$ for all $j \in 1..m$. The property follows by showing that the rule r^D_R of $\mathcal{A}_D$ may be applied on W to obtain $\mathcal{Q}$.

By Proposition 18, Since $\mathcal{Q}_{I_j} = \{q \mid\ e_j \rightarrow_{\mathcal{A}} q\}$, we have $W \in Reg^D_r$ for every rule $r \in R$ and $W \notin Reg^D_r$ for every $r \in R_2$. Hence $W \in Reg^D_R$. Likewise, we use Proposition 19 to prove that $W \in \varphi^D_R$. Therefore the rule r^D_R may be applied on W and we have $d \rightarrow_{\mathcal{A}_D} \mathcal{Q}_R = \mathcal{Q}$, as needed. $\square$

A.4 Proof of Proposition 8: Sheaves languages are not closed under complementation

We show that given a two-counter machine, there is a non-deterministic automaton accepting the set of bad computations of the machine (see Appendix A.1 for a definition of two-counter machines). Therefore, if the complement of this language was also accepted by some automaton, we could derive an automaton accepting the (good) computations reaching a final state. Therefore we could decide the halting problem for two-counter machines which is undecidable.

Assume we have a two-counter machine with set of states, $Q = \{q_1, \ldots, q_p\}$, final states $Q_f \subseteq Q$ and transition relation, $\delta \subseteq Q \times \{0,1\}^2 \times Q \times \{-1,0,1\}^2$, and counters C_1, C_2. We use the following signature to simulate the computations of the machine.

- a constant q for each state $q \in Q$ 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, that is by n successive occurrences of the symbol 1.

A configuration $\mathbb{C} = (q_i, C_1, C_2)$ is represented by the word $q_i \cdot C_1 \cdot a^{n_1} \cdot C_2 \cdot b^{n_2}$, where n_1, n_2 are the values of the counters C_1, C_2. As in Appendix A.1, we interpret words by their canonical representation as documents where we identify a letter a to an element $a[\epsilon]$ and a concatenation of words to a concatenation of documents.

Likewise, we can encode sequences of configurations $\mathbb{C}_0, \mathbb{C}_1, \ldots$ by concatenating the words obtained for each configuration $\mathbb{C}_i$: a sequence of configurations is a document accepted by the word expression $((\bigcup_{q\in Q} q) \cdot C_1 \cdot 1^* \cdot C_2 \cdot 1^*)^*$. 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 r_q (for each state of the counter machine $q \in Q$), $r_{C_1}, r_{C_2}, 1_{C_1}, 1_{C_2}, \bot$, as well as a unique final state, r_{error}. The state r_{C_1}, r_{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: $C_1 \to r_{C_1}$ and $C_2 \to r_{C_2}$;
- the constant 1 can reach (non-deterministically) five different states, 1_{C_1}, 1_{C_2}, $1_{C_1'}, 1_{C_2'}$ and $\bot$. We have five type 1 rules, $1 \to 1_{C_i}$ and $1 \to 1_{C_i'}$ for all $i \in \{1,2\}$ and $1 \to \bot$. The first four states are used to identify the value of the counter we are interested in, while $\bot$ is used for configurations of the machine whose counter 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 wild-card symbol $r_\_$ to denote any state of the kind r_q for $q \in Q$):

$$\phi \vdash \left(\begin{array}{l} (r_\_, r_{C_1}, \bot^*, r_{C_2}, \bot^*)^*, \\ \quad (r_q, r_{C_1}, 1^*_{C_1}, r_{C_2}, 1^*_{C_2}), (r_{q'}, r_{C_1}, 1^*_{C_1'}, r_{C_2}, 1^*_{C_2'}), \\ \qquad (r_\_, r_{C_1}, \bot^*, r_{C_2}, \bot^*)^* \end{array} \right) \to r_{error}$$

where $\phi(\#1_{C_1}, \#1_{C_2}, \#1_{C'_1}, \#1_{C'_2})$ is the Presburger 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.