Automata- and Logic-based Pattern Languages for Tree-structured Data

Frank Neven[1]* and Thomas Schwentick[2]

[1] University of Limburg
[2] Friedrich-Schiller-Universität Jena

Keywords: attribute grammars, automata, formal languages, logic, query evaluation, XML

Abstract. This paper surveys work of the authors on pattern languages for tree-structured data with XML as the main application in mind. The main focus is on formalisms from formal language theory and logic. In particular, it considers attribute grammars, query automata, tree-walking automata, extensions of first-order logic, and monadic second-order logic. It investigates expressiveness as well as the complexity of query evaluation and some optimization problems. Finally, formalisms that allow comparison of attribute values are considered.

1 Introduction

There is a powerful evolving technology around XML. A lot of proposals for query languages, pattern languages and the like are emerging. Some of these are complemented with working implementations. However, as such, there is no real standard for an XML query language. Although there are some requirements for an XML query language [31] and the latest XQuery working draft receives a lot of attention [6], it is not clear what the desired expressive power of such a query language should be. Furthermore, by and large, the expressive power and complexity of the existing languages is not well understood. This paper surveys research done by the authors on the expressiveness of query constructs for tree-structured data (that is, XML). The main focus is on logic and formalisms from formal language theory.

Let us look back at the history of query languages for relational databases. There a standard for expressive capabilities emerged from the tight connection between some query mechanisms and logic. More precisely, the well-known equivalence between relational algebra, core SQL and, first-order logic. First-order logic is such a natural and robust notion that it is no surprise that there are so many equivalent formalisms of the same expressive power. But the connection did not only help to find a standard. Also database theory profited a lot from it, e.g., it helped to understand what kinds of queries can be expressed or not expressed and to clarify semantical issues. The compositional nature of first-order logic further supported the development of a theory of views. In a

* Research Assistant of the Fund for Scientific Research, Flanders.

sense, first-order logic can be seen as a link between the operational nature of relational algebra and the declarative, user-oriented nature of SQL.

Coming back to XML the natural question arises: Is there a logic that could play a similar benchmark role for query languages for XML as first-order logic did for relational databases? One immediate difference is that XML data is primarily tree-structured. Obviously, this difference in representation affects the way queries are asked. However, when searching for a natural logic on trees there is one obvious candidate that was well investigated already 30 years ago: *monadic second-order logic* (MSO). That is, the extension of first-order logic (FO) where quantification over sets of nodes is allowed. There is a large body of research on that logic on trees. [30] The expressive power of MSO logic exactly matches the recognition power of several kinds of tree automata (e.g., bottom-up automata that combine information about the tree in one pass from the leaves to the root) and other formalisms like attributed grammars. It gives rise to a very robust class: the class of *regular tree languages* which is almost as robust as the class of regular string languages.

Further support for the consideration of MSO logic comes from an observation about existing query languages for XML. Most of them allow a kind of pattern matching along paths in an XML document; some of them by means of regular expressions. But there is a matching logic with the expressive power of regular expressions. By Büchi's Theorem it is MSO logic. Of course, there are also various kinds of string automata that capture the same level of expressive ability. By the way, there is an analogy to the case of relational databases that there is an operational way of specification by automata, a logical way of specification by MSO logic and a user-oriented declarative specification by regular expressions.

The mentioned connections strongly indicate that besides logic also automata might be helpful in the design and understanding of query formalisms for XML data. Further support in that direction is given by the very successful application of automata theoretic methods in the realm of verification. There also tree-structured data is very important and there is a tight interplay between logics (here: temporal logics) and automata.

Altogether we deal in this article with four main topics.

- The use of logical formulas as query mechanisms for tree-structured data, especially formulas of MSO and fragments of it;
- The use of parallel automata models (like bottom-up automata and attribute grammars). What is the impact of the knowledge about regular tree languages in the context of XML? There are two mismatches one has to deal with (a third one will be mentioned shortly): (1) For XML we are mainly interested in querying tree-structured documents rather than checking properties of a document; (2) For XML we have to deal with trees of unbounded degree (i.e., unranked trees) in contrast to the bounded degree trees in classical regular tree languages.
- The use of sequential automata models. Although it seems obvious that sequential automata models on trees have limited power compared with par-

allel automata this has not been proved yet. As these automata are related to XSLT, the understanding of the exact expressive power of such automata is a relevant topic.

– The fourth issue is orthogonal to the first three and it is related to the third mismatch. XML documents might contain arbitrary text, numbers, references etc., whereas the logics and automata mentioned so far work on trees with a fixed finite set of labels. Actually, whether one allows arbitrary data values or not is a main dividing line for theoretical work on tree-structured data. Whereas subqueries like `name = ''Johnson''` can still be handled in the finite alphabet framework, propositions like `value at vertex` x = `value at vertex` y cannot, in general. Whereas it is straightforward to equip MSO with the ability to deal with data values there are several possibilities how to extend automata models in that direction. We describe some results on such models.

It should be noted, that queries that do not compare values of different nodes are very common and are also frequent as subqueries. Therefore it is reasonable to try to understand them as good as possible. Nevertheless, the general formal model allows not only a finite set of labels but also attributes that might take values from an infinite domain. But we restrict access to these data values as we only allow equality and inequality comparisons of data values.

The main goals of the work surveyed here are as follows.

– Understand the expressive power of the various formalisms;
– Find out their evaluation complexity; Maybe find new formalisms with a better evaluation complexity;
– In particular, find formalisms that have a good combined evaluation complexity, i.e., formalisms for which evaluation is efficient even if the query itself is considered as part of the input;
– Investigate decidability issues, e.g., related to satisfiability and containment of queries in the various formalisms.

The rest of the paper is structured as follows. In Section 2 we introduce the necessary definitions. In Section 3, we consider pattern languages equivalent to MSO. In Section 4, we look at sequential formalisms. In Section 5, we study pattern languages based on fragment of MSO. In Section 6, we investigate formalisms with the ability to compare attribute values. Finally, we present some conclusions in Section 7.

2 Preliminaries

In this section we introduce the basic formalisms used throughout the paper, including the notion of (attributed) trees as abstraction of XML documents and the logical framework.

2.1 Trees and XML

We start with the necessary definitions regarding trees. Let Σ be a finite alphabet. The Σ-symbols will correspond to the element names of the XML document. To use trees as adequate abstractions of actual XML documents, we extend them with attributes from a finite set A that take values from an infinite (recursively enumerable) domain $\mathbf{D} = \{d_1, d_2, \ldots\}$. In the sequel we assume some fixed Σ and $\mathbf{D}$.

A *tree domain* τ *over* $\mathbb{N}$ is a subset of $\mathbb{N}^*$, such that if $v \cdot i \in \tau$, where $v \in \mathbb{N}^*$ and $i \in \mathbb{N}$, then $v \in \tau$. Here, $\mathbb{N}$ denotes the set of natural numbers without zero. If $i > 1$ then also $v \cdot (i-1) \in \tau$. The empty sequence, denoted by ε, represents the root. We call the elements of τ *vertices.* A vertex w is a *child* of a vertex v (and v the *parent* of w) if $vi = w$, for some i.

In this article, we only consider *finite* tree domains.

Definition 1. An *attributed* (Σ, A)*-tree* is a triple $t = (\mathrm{dom}(t), \mathrm{lab}_t, (\lambda_t^a)_{a \in A})$, where $\mathrm{dom}(t)$ is a tree domain over $\mathbb{N}$, $\mathrm{lab}_t : \mathrm{dom}(t) \to \Sigma$ is a function, and for every $a \in A$, $\lambda_t^a : \mathrm{dom}(t) \to \mathbf{D}$ is a partial function.

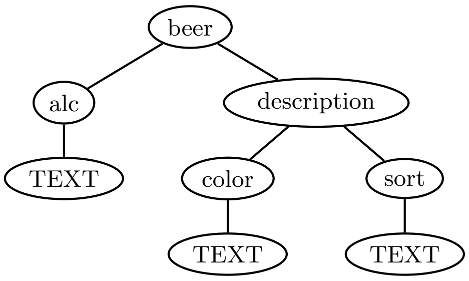
When Σ and A are clear from the context or not important, we sometimes say tree rather than (Σ, A)-tree. Note that in the definition of a tree there is no a priori bound on the number of children that a node may have.

We describe next the representation of XML documents as (Σ, A)-trees. We represent XML elements by means of the finite set Σ of labels and attribute values by the functions λ_t^a. Maximal sequences of character data are represented by nodes (inevitable leaves) that are labeled by a special element TEXT of Σ that is not used otherwise. The actual data is represented by an attribute PC from $\mathbf{D}$ also not used otherwise (see example).

Example 2. The XML document

```
<beer name="Grimbergen Trippel">
   <alc> 9 </alc>
    <description>
      <color> blonde </color>
      <sort> trappist </sort>
    </description>
</beer>
```

is faithfully modeled by the tree

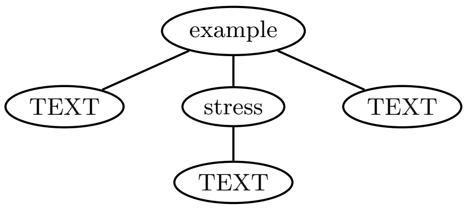


where $\lambda_t^{\mathrm{name}}(\varepsilon)$ = "Grimbergen Trippel", $\lambda_t^{\mathrm{PC}}(11)$ = "9", $\lambda_t^{\mathrm{PC}}(211)$ = "blonde", and $\lambda_t^{\mathrm{PC}}(221)$ = "trappist". Here, the domain $\mathbf{D}$ is assumed to contain the subset {"Grimbergen Trippel", "9", "blonde", "trappist"}.

The XML specification allows also mixed content elements like

```
<example>
  This is not a very
  <stress> meaningful </stress>
  sentence.
</example>
```

which can be modelled by the tree



with $\lambda_t^{\mathrm{PC}}(1)$ = "This is not a very", $\lambda_t^{\mathrm{PC}}(21)$ = "meaningful" and $\lambda_t^{\mathrm{PC}}(3)$ = "sentence".

$\square$

As we already pointed out in the introduction, most of the query mechanisms considered in this article do not allow the comparison of data values of different nodes. It turns out that in these cases one can model XML documents adequately by trees without data values, i.e., by $(\Sigma, \emptyset)$-trees. We refer to such trees simply as *Σ-trees*. The reason for this is as follows. For the purposes of this article, there is no need to fix a uniform representation of XML documents into trees but rather it is sufficient to consider representations *relative to the given query*. E.g., if the query asks for trappist beers and more than 9 % alcohol, we can represent XML documents by trees over an alphabet that has a label *trappist & 9 or less % alcohol* as well as a label for *trappist and more than 9% alcohol* and so on. In this manner and in the absence of data value comparisons between nodes, only a finite amount of information (depending on the query) is needed from the potentially infinite set of data values. This finite amount of information can be represented by Σ.

Hence, in sections 3 and 5 we only consider Σ-trees whereas in section 6 we consider general (Σ, A)-trees.

2.2 Queries and Patterns

As argued by Fernandez, Siméon, and Wadler [13] for the case of XML, queries on tree-structured data consist roughly of a *pattern* clause and a *constructor* clause. The purpose of the pattern language is to identify the different parts of the document that have to be combined to obtain the output document. The constructing part, on the other hand, indicates how the selected parts should be assembled. Such queries can, for instance, be written as

$$\texttt{WHERE } \varphi(\bar{x}), \texttt{ CONSTRUCT } \mathrm{result}(t),$$

where φ is a pattern selecting vertices and t is a tree containing at leaves special constructs like $\mathrm{yield}(x)$, $\mathrm{lab}(x)$, $\mathrm{subtree}(x)$ indicating that at this position the yield, the label, or the subtree rooted at the matched vertex for x should be plugged in. Note that the `WHERE` clause maps trees to an intermediate result of relational nature which is then transformed into a tree by the `CONSTRUCT` clause.

Pattern languages, therefore, form the basic building blocks of more general query languages transforming documents into other documents. Clearly, the choice of the pattern language can affect tremendously the expressive power of the overall query language.

In this article we focus on pattern languages. In the rest of the article, the terms *query* and *pattern* will both refer to mappings from trees to relations corresponding to pattern clauses.

Many of the pattern mechanisms considered in this article can not produce arbitrary relational results but are limited to unary results, i.e., results that are sets of vertices. This is justified as the identification of relevant parts of a tree is the most common purpose of pattern clauses (see e.g., the example queries of the W3C [31]).

2.3 Logic

We view trees as logical structures (in the sense of mathematical logic [10]) over the binary relation symbols E and $<$, and the unary relation symbols $(O_\sigma)_{\sigma \in \Sigma}$. We denote this vocabulary by τ_Σ. The domain of t, viewed as a structure, equals the set of nodes of t, i.e., $\mathrm{dom}(t)$. Further, E is the edge relation and equals the set of pairs $(v, v \cdot i)$ where $v, v \cdot i \in \mathrm{dom}(t)$. The relation $<$ specifies the ordering of the children of a node, and equals the set of pairs $(v \cdot i, v \cdot j)$, where $i < j$ and $v \cdot j \in \mathrm{dom}(t)$. For each σ, O_σ is the set of nodes that are labeled with a σ.

We consider first-order (FO) and monadic second-order logic (MSO) over these structures. MSO is FO extended with quantification over set variables. We refer the unfamiliar reader to, e.g., the books by Ebbinghaus and Flum [10], or the chapter by Thomas [30].

To be able to deal with attribute values (in section 6), we allow atomic formulas of the forms $a(x) = d$ where a is an attribute and $d \in \mathrm{dom}$ and of the form $a(x) = b(y)$. The former formula holds for u in t iff $\lambda_t^a(u) = d$, the latter holds for u and v in t iff $\lambda_t^a(u) = \lambda_t^b(v)$.

2.4 Complexity Issues

Besides comparing the expressive power of the different formalisms we are typically interested in the following kinds of computational problems related to the classes $\mathcal{C}$ of queries under consideration (where $\mathcal{C}$ is a set of *representations* of queries, e.g., automata or formulas).

NON-EMPTINESS Given a query $Q \in \mathcal{C}$, is there a tree t such that $Q(t) \neq \emptyset$?
CONTAINMENT Given queries $Q_1, Q_2 \in \mathcal{C}$, is $Q_1(t) \subseteq Q_2(t)$, for all trees t?
EQUIVALENCE Given queries $Q_1, Q_2 \in \mathcal{C}$, is $Q_1(t) = Q_2(t)$, for all trees t?
QUERY EVALUATION Given a query $Q \in \mathcal{C}$ and a tree t, compute $Q(t)$?

For NON-EMPTINESS and CONTAINMENT the first question is whether they are decidable, for a given class $\mathcal{C}$. If so, for most cases we consider the complexity is EXPTIME. Note that NON-EMPTINESS is also called SATISFIABILITY in the case of Boolean queries. The QUERY EVALUATION problem comes in two flavors. Either the query is considered fixed or it is considered as part of the input (so the complexity is a function in the length of the representation of Q and t). We refer to the former as the *data complexity* of QUERY EVALUATION and to the letter as the *combined complexity* of QUERY EVALUATION.

The linear time results and the complexity results in Section 5 assume Random Access Machines with a unit cost measure as computational model. Further they require a suitable representation of the input trees.

3 Pattern languages equivalent to MSO

As demonstrated in Section 2, XML documents can be faithfully represented by attributed trees. Trees have been studied in depth in the area of formal language theory for over 30 years and many formalisms have been proposed. We re-consider some of these formalisms from the viewpoint of pattern languages. More precisely we re-examine attribute grammars, tree automata, and tree transducers.

As mentioned in the introduction, there are two essential differences between trees studied in formal language theory and attributed trees representing XML documents as considered in this paper: (*i*) trees are unranked, that is, there is no fixed bound on the number of children of a node; and (*ii*) XML trees have attributes instantiated by elements coming from an infinite domain. In this section we do only consider pattern languages which disallow comparisons of attribute values of different vertices. Hence, as explained in Section 2 we consider only Σ-trees.

3.1 Tree automata

The basic computational model in tree language theory is the tree automaton. To warm-up, let us first consider finite state machines (FSMs) on strings. FSMs process input strings from left to right by first assigning the initial state to the first letter of the input string; a state for an inner position i is then determined via the transition function based on the label of $i-1$ and the state at $i-1$. The input string is accepted when a final state is reached at the end of the string.

On trees there are two quite different types of generalizations of this mechanism. They can be characterized as *sequential* and *parallel* tree automata, respectively. Sequential tree automata are similar to string automata as they have only *one head* which moves around the tree. In order to inspect the whole tree they have to visit some vertices several times. Sequential automata will be discussed in Section 4. On the other hand, most parallel tree automata keep the *visit positions only once* paradigm of string automata. Bottom-up tree automata, for instance, process trees in one pass from the leaves to the root. That is, first initial states are assigned to leaf nodes (depending on the symbol they carry). Second, the state at an inner node is determined via the transition function based on the label of the inner node and on the states assumed at their children. Finally, the automaton accepts if a final state is reached. In this subsection we study parallel tree automata as query mechanisms for trees.

How do we define the transition function for a bottom-up tree automaton? If the arity of the input trees is bounded by n, a transition function is simply a function $\delta : \bigcup_{i=0}^{n} Q^n \times \Sigma \rightarrow Q$. However, when the arity is unbounded, we can not adapt this approach as it would imply having to define transitions for an infinite number of cases. Brüggemann-Klein, Murata, and Wood [4] proposed a solution based on regular sets of states which we explain next.

Formally, a (bottom-up) tree-automaton is a tuple $T = (Q, \Sigma, \delta, q_0, F)$ where Q is a set of states, Σ is the alphabet of labels, q_0 is the initial state and F is the set of accepting states, as in the definition of finite state machines, Finally, $\delta : Q \times \Sigma \rightarrow 2^{Q^*}$ is a transition function mapping every pair (q, σ) to a regular language over Q. T accepts a tree t if there is a labeling function $\gamma : \mathrm{dom}(t) \rightarrow Q$ such that (i) for every vertex v, with children $v_1 \cdots v_n$, $\gamma(v1) \cdots \gamma(vn) \in \delta(\gamma(v), \mathrm{lab}_t(v))$; [1] and (ii) $\gamma(\varepsilon) \in F$, that is, a final state is reached. A tree language, that is, a set of trees, is regular when it is accepted by a tree-automaton.

Using regular string languages to specify properties of the sequence of children in unranked trees lies at the core of many of the following formalisms.

3.2 Query Automata

This section is based on [23]. A *query automaton* (QA) is a two-way (i.e., up and down) deterministic finite tree automaton with a distinguished set of selecting states. At each step of a computation, there is a partial mapping s from $\mathrm{dom}(t)$ to Q. Intuitively, the set C of vertices v, for which $s(v)$ is defined contains those vertices of the tree that are currently visited by a head of the automaton. This set C has to contain exactly one vertex of each path from the root to a leaf. A computation step consists of either a *down transition* or an *up transition*. I.e., either in C, a vertex is replaced by its children, or, if all children of a certain vertex v are in C, they might be replaced by v itself. In both cases, the new vertices in C are assigned states depending on the transition function.

A vertex is selected by a QA if it is visited at least once in a selecting state. Hence, a QA can compute unary queries in a natural way: the result of a QA on a tree consists of all those nodes that are selected during the computation of the QA on that tree. We stress that QAs are quite different from the tree acceptors studied in formal language theory [14]. Although two-way tree *acceptors* are

[1] Note, that in particular, when v is a leaf, ε should be in $\delta(\gamma(v), \mathrm{lab}_t(v))$.

equivalent to one-way acceptors [20] it is straightforward to see that (two-way) QAs are not equivalent to bottom-up query automata. Indeed, a bottom-up QA, for example, cannot compute the query "select all leaves if the root is labeled with σ", simply because it cannot know the label of the root when it starts at the leaves.

Theorem 3. *On ranked trees, QAs express exactly the MSO definable unary patterns.*

For QAs on unranked trees the picture looks quite different. QAs over unranked trees cannot even express all FO definable unary patterns. The basic reason is that in the unranked case very little information can be passed from one sibling to another. To resolve this, QAs are equipped with *stay transitions* where a two-way string-automaton reads the string formed by the states at the children of a certain node, and then outputs for each child a new state. Automata are then restricted to make only a constant number of stay transitions for the children of each node. These automata are called *strong* QAs.

Theorem 4. *On unranked trees, strong QAs compute exactly all MSO-definable unary patterns.*

W.r.t. standard decision problems we obtain the following.

Theorem 5. *NON-EMPTINESS, CONTAINMENT, and EQUIVALENCE of strong QAs are complete for* EXPTIME.

Although the run time of a QA might be quadratic in the size of the tree query evaluation can be done more efficiently.

Theorem 6. *The data complexity of QUERY EVALUATION for QAs is linear time. The combined complexity of QUERY EVALUATION for QAs is PTIME.*

3.3 Attribute Grammars

Attribute grammars (AGs), as defined by Knuth [16], constitute a deeply studied general computational model for trees [9]. In brief, an AG consists of a context-free grammar and a set of rules defining annotations (called attributes) of nodes of derivation trees. As defined by Knuth, the domain of attributes can be anything and the semantic rules can be any recursive function. Further, attributes can be defined in a top-down (inherited) or a bottom-up (synthesized) way.

Towards pattern languages for tree-structured data, Neven and Van den Bussche [27] considered Boolean-valued AGs (BAGs) where attributes are restricted to be Boolean-valued and rules are propositional formulas over attributes. As an example consider the following BAG working on a list of paragraphs

$$\begin{array}{ll} L \to Lp & \mathit{odd}(0) := \neg \mathit{odd}(1) \\ L \to p & \mathit{odd}(0) := \text{true}. \end{array}$$

Here, the attribute *odd* will be true for all L-labeled nodes on an odd position when starting counting from the bottom. In brief, *odd* is true for the last L-labeled node; for an inner node *odd* is the negation of the *odd*-value of the next L. In the semantic rules the numbers 0 and 1 refer to the left-hand side and the first non-terminal of the right-hand side of a production, respectively. By designating *odd* as the result attribute, on an input tree the BAG retrieves all nodes for which *odd* is true. Although BAGs constitute a seemingly simple formalism, it can be shown that BAGs express precisely all MSO definable unary patterns [27, 3].

XML documents, however, are usually described by *extended* context-free grammars (DTDs). These are grammars with regular expressions on the right-hand sides of productions. The above context-free grammar could be specified, for instance, like $\text{List} \to L^*,\ L \to p$. One problem that arises when defining AGs to work on such grammars is that semantic rules should be able to depend on an unbounded number of attributes (as the production $\text{List} \to L^*$ does not put any restriction on the number of L's). The latter problem can be resolved by using regular languages over attribute values as semantic rules.[2] Consider the rule

$$\text{List} \to L^* \qquad \text{odd}(1) := L^*\#L(LL)^*$$

and the tree consisting of a root with label List and four children with label L. To determine whether *odd* is true for, say, the second child, we consider the string $L\#LLL^*$ where we insert the marker # in front of the second position. As this string matches the regular expression $L^*\#L(LL)*$, *odd* will be true for the second child. In general, semantic rules can also depend on the values of other attributes, as opposed to labels of nodes; we refer the interested reader to [22]. Again unary patterns can be expressed by designating some attribute as the result attribute.

The next theorem provides evidence for the robustness of the definition of AGs on unranked trees.

Theorem 7. *AGs compute exactly all MSO-definable unary patterns.*

Theorem 8. *NON-EMPTINESS, CONTAINMENT, and EQUIVALENCE of AGs are complete for* EXPTIME.

The latter result can be used to improve optimization of Region Algebra expressions [8]. Indeed, by exhibiting a linear time translation from Region Algebra expressions to attribute grammars, one obtains that testing equivalence of Region Algebra is in EXPTIME. This should be contrasted to the hyper-exponential upper bound of Milo and Consens [8].

Corollary 9. *EQUIVALENCE of Region Algebra expressions is in* EXPTIME.

The data complexity of QUERY EVALUATION for AGs is linear time. This is clear because of the equivalence with Query Automata. But there is also a straightforward evaluation strategy which gives this time bound directly, if the evaluation of single rules can be done in constant time. The combined complexity is PTIME.

[2] Actually, some more problems arise. See [22].

3.4 Related Work

We briefly discuss some related formalisms for unranked trees. Neumann and Seidl define a μ-calculus for specifying unary patterns [28] and a push-down tree automaton model for evaluating them. These expressions can be evaluated by a one-pass traversal of the tree. Murata defines a pattern language for expressing unary patterns based on tree-regular expressions [21]. These correspond exactly to MSO. Finally, we mention that tree-transducers (over unranked trees) as a formal model for XSLT have been considered in [17]. We come back to XSLT in Section 6.

4 Pattern languages based on sequential automata

The automata model discussed in the previous section is a parallel one. For instance, a down transition assigns states to all children of the current vertex which in turn are processed rather independently. So the control of the automaton is at several nodes of the input tree simultaneously, rather than at just one. In contrast, the finite control of a tree-walking automaton (TWA) is always at one node of the input tree. The computation starts at the root in the initial state. Based on the label of the current node and its location (that is, root, first child, last child, or leaf) the automaton changes state and steps to one of the neighboring nodes (that is, parent, first child, last child, left sibling, right sibling). An automaton can express selections in the same way as query automata: all nodes that are visited in a selecting state are selected.

It is an open problem whether TWAs can express all MSO definable selection patterns. This is related to the question whether TWAs can define all regular tree languages. To be precise, a TWA accepts a tree when it selects the root. Engelfriet and Hoogeboom conjecture that TWAs are strictly weaker than tree automata [11, 12] which would imply that TWA cannot define all MSO selection patterns. In [25] it is shown that over ranked trees, TWAs accept precisely the set of trees definable in a fragment of deterministic unary transitive closure logic. Further, the conjecture is proved for a restriction of TWAs: TWAs that visit each subtree only once (and a mild generalization thereof) cannot express the set of all regular tree languages.

In research on tree-structured data, tree-walking automata are used for various purposes and appeared in various forms. Milo, Suciu, and Vianu [19], for instance, use a transducer model based on tree-walking automata as a formal model for an XML transformer encompassing most current XML transformation languages. Another occurrence of tree-walking automata is embodied in the actual XML transformation language XSLT [7] proposed by the W3C. In formal language theoretic terms, this query language can be best described as a *tree-walking tree transducer* [2].

As a third example we mention the caterpillar expressions of Brüggeman-Klein and Wood. [5] These are regular expressions over moving instructions {up, left, down, right, isLeaf, isFirst, isLast, isRoot} and Σ-symbols. For instance, the expression $a\,\mathrm{isLeaf}\,(\mathrm{up}^*\, b\, \mathrm{down}^*\, c)$, selects all a-labeled leaves that have a b-labeled ancestor who in turn has a c-labeled descendant. Again, a caterpillar accepts a tree when it selects the root. Brüggeman-Klein and Wood leave it as an open question whether caterpillar expressions can define all regular (unranked) tree languages.

Hence, results on the expressiveness of tree-walking automata could give insight in the expressiveness of actual XML transformation languages.

5 Pattern languages based on fragments of MSO

In this section we consider fragments of MSO as query mechanisms for tree-structured data. We still keep the restriction that the comparison of data values other than comparisons with fixed constants are not allowed. Hence, as discussed in Section 2 we deal with Σ-trees without data values. Section 6 treats the case where data values are present. This section is based on the work reported in [24, 29]. First we introduce an intermediate logic (FOREG) between FO and MSO logic which attempts to capture the expressive power of languages with regular path expressions in a very general way. The basic idea is to allow regular expressions over formulas that are evaluated along a vertical path or along the children of a node. Then we turn our attention to fragments of MSO, FOREG and FO respectively, for which the QUERY EVALUATION problem has low combined complexity. These fragments are obtained by restricting quantification of variables in the spirit of guarded logics [1] on one hand and by allowing vertical path expressions on the other hand.

5.1 Regular expressions

As mentioned before, the logic FOREG is an extension of FO by regular path expressions. Further, it is designed to capture also arbitrary nesting of such expressions. It uses the following two kinds of path formulas.

– If P is a regular expression (in the usual sense) over formulas with free variables r and s then $\varphi = [P]^{\downarrow}_{r,s}(x, y)$ is a *vertical path formula.*
– If P is a regular expression over formulas with free variable r then $[P]^{\rightarrow}_{r}(x)$ is a *horizontal path formula.*

We refer to path formulas also by the term *path expressions.* A simple example of a horizontal path formula is $[(O_a(r))^* O_b(r)]^{\rightarrow}_{r}(x)$.

The semantics of such formulas is defined as follows.

– Let $\varphi = [P]^{\downarrow}_{r,s}(x, y)$ be a vertical path formula. Let t be a tree and let v, w be vertices of t. Then, $t \models \varphi[v, w]$, iff w is in the subtree rooted at v and there is a labeling of the edges on the path from v to w with formulas, such that (1) each edge (u, u') is labeled with a formula $\theta(r, s)$ such that $t \models \theta[u, u']$, and (2) the sequence of labels along the path from v to w matches P.

– Let $\psi = [P]^{\rightarrow}_{r}(x)$ be a horizontal path formula. Then $t \models \psi[v]$, iff there is a labeling of the children of v with formulas, such that (1) each child w of v is labeled with a formula $\theta(r)$ such that $t \models \theta[w]$, and (2) the sequence of labels is matched by P.

Example 10. (a) The example formula $[(O_a(r))^* O_b(r)]^{\rightarrow}_{r}(x)$ holds at a vertex v, iff the rightmost child of v is labeled with a b and the remaining children are labelled with an a.

(b) For a more involved example consider first the formula $\varphi(x) = [(\text{true true})^*]^{\rightarrow}_{r}(x)$. Here, true stands for a formula like $r = r$ which always holds. Hence, φ holds at a vertex v if the number of children of v is even. Then the formula $\psi(x, y) = [(\varphi(r)(\neg\varphi(r)))^*]^{\downarrow}_{r,s}(x, y)$ holds for vertices v and w, if w is below v, the path from v to w is of even length and the vertices above w on this path have alternatingly even and odd degree. An example is given in Figure 1.

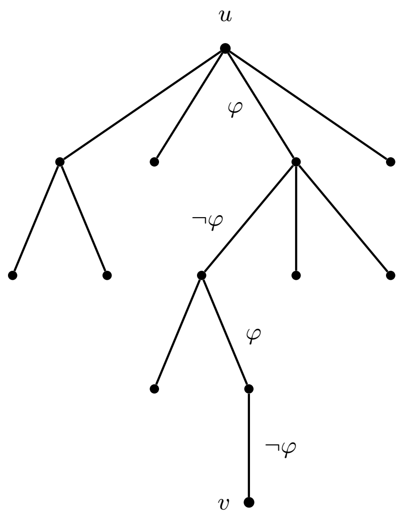


Fig. 1. The formula $\psi(x, y) = [(\varphi(r)(\neg\varphi(r)))^*]^{\downarrow}_{r,s}(x, y)$ holds for the pair (u, v) as the path can be labeled by the sequence $\varphi\neg\varphi\varphi\neg\varphi$.

The logic FOREG is the extension of FO by vertical and horizontal path formulas.[3] More formally,

(1) Every FO formula is an FOREG formula.

[3] It should be noted that this logic was called FOREG* in [24]. There, the analogue logic in which path expressions are only formed over atomic formulas was called FOREG.

(2) If P is a regular expression over FOREG formulas then $\varphi = [P]^{\downarrow}_{r,s}(x,y)$ is an FOREG formula.
(3) If P is a regular expression over FOREG formulas then $[P]^{\rightarrow}_{r}(x)$ is an FOREG formula.

Of course, FOREG can express all FO queries and every FOREG query can be expressed in MSO. It was shown in [24] that these inclusions are strict, i.e., there are MSO queries that can not be expressed in FOREG and there are FOREG queries that can not be expressed in FO.

5.2 Guarded Fragments

There is a significant difference between the data complexity and the combined complexity of the QUERY EVALUATION problem for MSO formulas. Whereas the data complexity is linear time the combined complexity is non-elementary (i.e., there is *no fixed tower of exponentials* that captures the complexity) [18]. In particular, the translation of an MSO formula into an equivalent tree automaton is non-elementary. Even for first-order formulas the combined complexity of QUERY EVALUATION is PSPACE-complete.

In this section we consider syntactic restrictions GFO, GFOREG and GMSO of the logics FO, FOREG and MSO, respectively, that have a much better combined complexity, linear time in the tree size and exponential time in the formula size. But they still can express all queries of the respective full logic. Clearly, this means, e.g., that there are MSO formulas that are much more succinct than their counterpart in the restricted fragment. But we believe that the restricted formulas allow the formulation of natural queries in a transparent way.

For the moment, we restrict attention to unary queries. In particular, all formulas considered have one free variable. Whether a vertex v is selected by a query might depend on properties of the subtree rooted at v, on properties of the path from the root to v and on properties of the remainder of the tree. It is reasonable to assume that in many cases properties of the subtree at v and of the path to v will be more important than properties of the rest of the tree.

In this spirit the fragments we define express properties of vertices in a tree in a modular way, i.e., by Boolean combinations of formulas $\varphi(x)$ that only speak about the subtree rooted at x and formulas $[P]^{\downarrow}_{r,s}(\text{root}, x)$ speaking about the path from the root to x and, if necessary, about the rest of the tree. In the latter kind of formulas, P is a regular expression over formulas $\psi(r,s)$ each of which is restricted to the subtree rooted at r. Therefore such *subtree-restricted quantification* is the basic ingredient of our logic.

We first define the syntax of GFOREG formulas. GFO formulas are then obtained as the restriction of GFOREG where only star-free regular expressions are allowed.

First of all, we make use of an additional partial order. For vertices u, v of a tree t it holds $u \preccurlyeq v$ if v is a vertex of the subtree of t that is rooted at u.

Besides the usual kind of variables $x, y, x', x_1, x_2, \ldots$ (to which we refer as *quantifier variables* in the following) we use a second kind of variables, called *expression variables*. They are only used in path expressions and are denoted by symbols like r and s.

The syntax of GFOREG formulas is defined as follows.

(i) Every atomic formula is a GFOREG formula.
(ii) If y is a quantifier variable and φ is a GFOREG formula with free quantifier variables from $\{x, y\}$ then $\exists y(x \preccurlyeq y \wedge \varphi)$ is a GFO formula.
(iii) If P is a regular expression over GFOREG formulas without free quantifier variables then $[P]^{\downarrow}_{r,s}(x, y)$ is a GFOREG formula.
(iv) If P is a regular expression over GFOREG formulas without free quantifier variables then $[P]^{\rightarrow}_{r}(x)$ is a GFOREG formula.
(v) Any Boolean combination of GFOREG formulas is a GFOREG formula.

GMSO formulas have the additional ability to quantify over sets but quantification is restricted to the current subtree. I.e., if φ is a GMSO formula, $\text{free}(\varphi) \subseteq \{x, X\}$ then $\exists X(x \preccurlyeq X \wedge \varphi)$ is a GMSO formula.

A closer inspection of the definitions above shows that the fragments are essentially two-variable fragments of their respective full logics (in the case of MSO with one additional free set variable). In so far and in the general way of thinking about trees they are quite similar in spirit to temporal logics on trees. Actually, a design goal for the development of a usable query language based on these fragments could be to get rid of the explicit use of variables in a similar way as variables are omitted in temporal logics.

As already mentioned, we get the following results about the expressive power of the defined fragments.

Theorem 11. *For each FO (FOREG, MSO) formula $\varphi(x)$ there is a Boolean combination of GFO (GFOREG, GMSO) formulas $\psi(x)$ and formulas $[P]^{\downarrow}_{r,s}(root, x)$ where P consists of GFO (GFOREG, GMSO) formulas which is equivalent to φ on Σ-trees.*

The evaluation complexity is as follows.

Theorem 12. *1. There is an algorithm which computes on input (t, φ), where t is a tree and φ is a GFOREG formula, the set of all vertices v of t such that $t \models \varphi(v)$ in time $O(|t|2^{|\varphi|})$.*
2. There is an algorithm which computes on input (t, φ), where t is a tree and φ is a GMSO formula, the set of all vertices v of t such that $t \models \varphi(v)$ in time $O(|t|2^{|\varphi|^2})$.
3. There is an algorithm which computes on input (t, φ), where t is a tree and φ is a GMSO formula in which no vertical path formulas occur within the scope of any set quantification, the set of all vertices v of t such that $t \models \varphi(v)$ in time $O(|t|2^{|\varphi|})$.

In principal, the logical approach makes it very easy to go from unary queries to queries of arbitrary arity by simply passing from formulas with one free variable to formulas with more free variables. This can not be done directly in the case of the fragments considered here as the restricted use of variables is essential for them. Nevertheless, it was shown in [29] that queries of arbitrary arity can be expressed in a very similar way with one additional kind of path expressions which talk about the children of a vertex v between those two children that contain vertices u and w in their subtree, respectively. The results about expressive power go through. In the first two statements on evaluation complexity of Theorem 12 one has to replace the factor $|t|$ by $|t|^k$ in the case of k-ary queries. Furthermore, on input (t, φ) one can compute in time $O(|t|^2 2^{|\varphi|})$ a data structure which allows to check in time $O(|\overline{v}|)$, whether $t \models \varphi(\overline{v})$ holds, for a given tuple $\overline{v}$.

6 Formalisms with comparisons between data values

In this section we allow comparisons between attribute values as described in Section 2.3. In the above we showed that many formalisms for Σ-trees are equivalent to MSO. Obtaining such a correspondence for trees with data values is much harder if comparisons of values of different vertices are allowed. One of the reasons is that the data complexity of QUERY EVALUATION increases quite a bit. More precisely, it is shown in [26] that for every $i \in \mathbb{N}$, there are MSO formulas φ_i and ψ_i such that the QUERY EVALUATION on (Σ, A)-trees for φ_i and ψ_i is hard for Σ_i^P and Π_i^P, respectively. The prime reason is that attributed trees can encode graphs in a way such that even FO formulas can decode them. So when defining computational devices over attributed trees one should define powerful (complex) formalisms to capture MSO or settle for less. We choose for the latter.

This section is based on [26]. It should be noted that the results in that paper are formulated for automata on *strings* over data values. But the results we mention here easily carry over to tree automata. Therefore, our exposition moves back and forth between string automata and tree automata.

To keep our operational models simple and manageable, we only consider formalisms which are included in the regular languages when restricted to ordinary trees. It is useful to observe that for attributed trees it is no longer sufficient to equip automata with states alone. Indeed, automata should at least be able to check equality of attributes. There are two main ways to do this:

– store a finite set of positions and allow equality tests between the symbols on these positions;
– store a finite set of symbols only and allow equality tests with these symbols.

The first approach, however, leads to multi-head automata, immediately going beyond regular languages. Therefore, we instead equip tree-walking automata with a finite set of *pebbles* whose use is restricted by a stack discipline. That is, all pebbles have a number and pebble i can only be lifted when pebble $i+1$ is not placed. The automaton can test equality by comparing the attributes of the pebbled symbols. We refer to these automata as pebble automata. In the second approach, we follow[4] Kaminski and Francez [15] and extend finite tree-walking automata with a finite number of *registers* that can store attribute values. When

[4] Their model is based on strings but easily translates to trees.

walking on a tree, an automaton compares an attribute value with values in the registers; based on this comparison it can decide to store the current symbol in some register. We refer to these automata as register automata. Both models can express unary queries by means of a selection function.

There is a great mismatch in expressive power between register and pebble automata. Indeed, register automata do not fit nicely into the framework of logically defined queries or languages.

Theorem 13. – *There are Boolean FO queries not expressible by any register automaton.*
– *There are Boolean queries expressible by register automata that are not definable in MSO.*

On the one hand, register automata cannot even define all properties in FO. The basic idea is as follows. Consider strings of the form $u\#v$ where $u, v \in \Sigma^*$. Here, # functions as a delimiter. The only way an automaton can compare the data values of the left side with the ones from the right side is by storing values in the registers. Based on communication complexity theoretic arguments one can show that this does not suffice to express all of FO. On the other hand, register automata are quite strong as they can express properties not even expressible in MSO. To see this, consider strings of the same form $u\#v$ as before. Define N_u and N_v as the set of symbols occurring in attributes in u and v, respectively. It can be shown that there is a register automaton that accepts $u\#v$ iff $|N_u| = |N_v|$ while there is no such MSO sentence.

The formal model of XSLT of [2] is based on tree-walking transducers with registers. By applying Theorem 13 we get that XSLT programs without nested calls cannot define all of FO.

Pebble automata behave much better, their expressiveness lies in between FO and MSO. Indeed, pebbles provide a mechanism to instantiate variables occurring in formulas. A brute force approach to check, for instance, the formula $\exists x \exists y \delta(x, y)$, where $\delta(x, y) \equiv E(x, y) \wedge a(x) = b(y)$, is the following. Assign a pebble to x and one to y, say i_x and i_y. Put them subsequently on all possible combinations of vertices and test whether $\delta(i_x, i_y)$ holds. The latter only involves local checks. The automaton accepts when it finds such two vertices. Further, the enforced stack discipline makes sure the model behaves in a regular way. Indeed, it can be shown that pebble automata can be defined in MSO.

Theorem 14. – *All unary FO queries are expressible by pebble automata.*
– *Every query defined by a register automaton is also definable in MSO.*

The evaluation complexity is as follows.

Theorem 15. *The data complexity of QUERY EVALUATION for register and pebble automata is in* PTIME.

7 Discussion

We surveyed work investigating well-known formalisms from formal languages and logic as a pattern languages for tree-structured data. The main focus was on expressiveness, evaluation complexity, and decision problems relevant to optimization. Although attribute grammars as well as query automata are quite expressive they are quite complicated formalisms and do not seem to be the basis for an easy-to-use pattern language. Tree-walking automata are much more intuitive. Therefore, we need to understand better their expressiveness. However, the problem whether they capture MSO hac been open for a while now and therefore appears to be difficult. The restricted logics we considered might be useful in the design of a pattern language but this requires further work. We merely touched upon the issue of comparison of data values. Undoubtedly it deserves a lot more investigation.

References

1. H. Andréka, J. van Benthem, and I. Németi. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27:217–274, 1998.
2. G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. *1st International Conference on Computational Logic*, pages 1137–1151, *Lecture Notes in Artificial Intelligence*, volume 1861. Springer, 2000.
3. R. Bloem and J. Engelfriet. A comparison of tree transductions defined by monadic second order logic and by attribute grammars. *Journal of Computer and System Sciences*, 61(1):1–50, 2000.
4. A. Brüggemann-Klein, M. Murata, and D. Wood. Regular Tree and Regular Hedge Languages over Unranked Alphabets: Version 1, April 3, 2001. Technical report, HKUST-TCSC-2001-05, Hong Kong University of Science & Technology, 2001.
5. A. Brüggemann-Klein and D. Wood. Caterpillars: A Context Specification Technique. *Markup Languages*, 2(1):81–106, 2000.
6. D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery: a query language for XML. Latest version: http://www.w3.org/TR/xquery/.
7. J. Clark. XSL Transformations (XSLT) Version 1.0. Latest version: http://www.w3.org/TR/xslt.
8. M. Consens and T. Milo. Algebras for querying text regions: Expressive power and optimization. *Journal of Computer and System Sciences*, 3:272–288, 1998.
9. P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars: Definition, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science.* Springer, 1988.
10. H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer, 1995.
11. J. Engelfriet and H. J. Hoogeboom. Tree-walking pebble automata. In J. Karhumäki, H. Maurer, G. Paun, and G.Rozenberg, editors, *Jewels are forever, contributions to Theoretical Computer Science in honor of Arto Salomaa*, pages 72–83. Springer-Verlag, 1999.
12. J. Engelfriet, H.J. Hoogeboom, and J.-P. van Best. Trips on trees. *Acta Cybernetica*, 14:51–64, 1999.
13. M. Fernandez, J. Siméon, and P. Wadler, editors. *XML Query languages: Experiences and Exemplars*, 1999. `http://www-db.research.bell-labs.com/user/simeon/xquery.html`.
14. F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 1. Springer, 1997.
15. M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

16. D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. See also *Mathematical Systems Theory*, 5(2):95–96, 1971.
17. S. Maneth and F. Neven. Structured Document Transformations Based on XSL. In R. Conner, A. Mendelzon, editors, *Research Issues in Structured and Semistructured Database Programming*, volume 1949 of *Lecture Notes in Computer Science*, pages 80-98. Springer 2000.
18. A. R. Meyer. Weak monadic second-order theory of successor is not elementary recursive. In R. Parikh, editors, *Logic Colloquim*, volume 453 of Lecture Notes in Mathematics, pages 132–154. Springer, 1975.
19. T. Milo, D. Suciu, and V. Vianu. Type checking for XML transformers. In *Proceedings of the Nineteenth ACM Symposium on Principles of Database Systems*, pages 11–22. ACM Press, 2000.
20. E. Moriya. On two-way tree automata. *Information Processing Letters*, 50:117–121, 1994.
21. M. Murata. Extended Path Expressions for XML. To appear in *Proceedings of the Twentieth ACM Symposium on Principles of Database Systems.* ACM Press, 2001.
22. F. Neven. Extensions of attribute grammars for structured document queries. In R. Conner, A. Mendelzon, editor, *Research Issues in Structured and Semistructured Database Programming*, volume 1949 of *Lecture Notes in Computer Science*, pages 99-116. Springer 2000.
23. F. Neven and T. Schwentick. Query automata. To appear in *Theoretical Computer Science.* Extended abstrast in *Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems*, pages 205–214. ACM Press, 1999.
24. F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proc. 19th Symposium on Principles of Database Systems (PODS 2000), Dallas*, pages 145–156, 2000.
25. F. Neven and T. Schwentick. On the power of tree-walking automata. To appear in *Information and Computation.* Extended abstract in *27th International Colloquium on Automata, Languages and Programming*, pages 547–560, *Lecture Notes in Computer Science*, volume 1853. Springer, 2000.
26. F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinte alphabets. Submitted.
27. F. Neven and J. Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. To appear in the *Journal of the ACM.* Extended abstract appeared in *Proceedings of the Seventeenth ACM Symposium on Principles of Database Systems*, pages 11–17. ACM Press, 1998.
28. A. Neumann and H. Seidl. Locating matches of tree patterns in forests. In V. Arvind and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 134–145, *Lecture Notes in Computer Science*, volume 1530. Springer, 1998.
29. T. Schwentick. On Diving in Trees. In M. Nielsen and B. Rovan, editors, *Mathematical Foundations of Computer Science (MFCS 2000)*, pages 660–669, *Lecture Notes in Computer Science*, volume 1893. Springer, 2000
30. W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 7. Springer, 1997.
31. World Wide Web Consortium. XML Query Requirements. Latest version: http://www.w3.org/TR/xmlquery-req.