Logic Wrappers and XSLT Transformations for Tuples Extraction from HTML

Costin Bădică[1] and Amelia Bădică[2]

[1] University of Craiova, Software Engineering Department
Bvd.Decebal 107, Craiova, RO-200440, Romania
badica_costin@software.ucv.ro
[2] University of Craiova, Business Information Systems Department
A.I.Cuza 13, Craiova, RO-200585, Romania
ameliabd@yahoo.com

Abstract. Recently it was shown that existing general-purpose inductive logic programming systems are useful for learning wrappers (known as L-wrappers) to extract data from HTML documents. Here we propose a formalization of L-wrappers and their patterns, including their syntax and semantics and related properties and operations. A mapping of the patterns to a subset of XSLT that has a formal semantics is outlined and demonstrated by an example. The mapping actually shows how the theory can be applied to obtain efficient wrappers for information extraction from HTML.

1 Introduction

Many Web resources can be abstracted as providing relational information as sets of tuples, including: search engines result pages, product catalogues, news sites, product information sheets, a.o. Recently, we have experimentally shown that general-purpose inductive logic programming (ILP hereafter) systems might be useful for learning logic wrappers (i.e. L-wrappers) to extract tuples from Web pages written in HTML ([4–6]). Our wrappers use patterns defined as logic rules. Technically, these rules are acyclic conjunctive queries over trees ([12]). Their patterns are matched against XHTML information sources to extract the relevant information. Here we complement this work by giving a formalization of L-wrappers and their patterns using directed graphs. Then we show how L-wrappers can be efficiently implemented using XSLT ([9]) and corresponding transformation engines.

Web pages and XML can be regarded as semi-structured data modeled as labeled ordered trees ([1]). In this paper we study the syntax, semantics, and properties of patterns used in L-wrappers, in the more general context of information extraction from semi-structured data. This study serves at least three purposes: i) as a concise specification of L-wrappers; this enables a theoretical investigation of their properties and operations and allows comparisons with related works; ii) as a convenient way for mapping L-wrappers to XSLT for efficient processing using available XSLT processing engines; the mathematically sound

approach enables also the study of the correctness of the implementation of L-wrappers using XSLT (however this issue is not addressed in this paper; only an informal argument is given); iii) furthermore, the experimental work reported in [6] revealed some scalability problems of applying ILP to learn L-wrappers of arity greater than three (i.e. tuples containing at least three attributes). There, this was explained by the large number of negative examples required, that grows exponentially with the tuple arity. Here we show that this problem can be tackled as follows: in order to learn a wrapper to extract tuples of arity $k \geq 3$, we suggest learning $k-1$ wrappers to extract tuples of arity 2 and then use the pattern merging operator to merge them, rather than learning the wrapper in one shot. This approach is demonstrated using an example.

The work described in this paper is part of an ongoing research project that investigates the application of general-purpose ILP systems (like FOIL [18] or Aleph [2]), logic representations of wrappers and XML technologies (including the XSLT transformation language [9]) to information extraction from the Web.

The paper is structured as follows. First, we present the syntax and semantics of extraction patterns. Syntax is defined using directed graphs, while semantics and sets of extracted tuples are defined using a model-theoretic approach. Then, we discuss pattern properties – subsumption, equivalence, and operations – simplification and merging. We follow with a case study inspired by our previous work on using ILP to learn L-wrappers for HTML. In the case study we analyze the example L-wrappers from paper [6]. We discuss their combination using pattern merging and describe their mapping to XSLT. An explanation of why this mapping works is also given. Note that the application of ILP, logic representations and XML technologies to information extraction from the Web is not an entirely new field; several approaches and tool descriptions have already been proposed and published ([3, 8, 11, 13, 14, 16, 19, 20]; see also the survey in [15]). Therefore, before concluding, we follow with a summary of these relevant works, briefly comparing them with our own work. The last section of the paper contains some concluding remarks and points to future research directions.

2 Patterns. Syntax and Semantics

We model semi-structured data as labeled ordered trees. A wrapper takes a labeled ordered tree and returns a subset of tuples of extracted nodes. An extracted node can be viewed as a subtree rooted at that node. Within this framework, a pattern can be interpreted as a conjunctive query over labeled ordered trees, yielding a set of tree node tuples as answer. The node labels of a labeled ordered tree correspond to attributes in semi-structured databases or tags in tagged texts. Let Σ be the set of all node labels of a labeled ordered tree.

For our purposes, it is convenient to abstract labeled ordered trees as sets of nodes on which certain relations and functions are defined. Note that in this paper we are using some basic graph terminology as introduced in [10].

Definition 1. *(Labeled ordered tree) A* labeled ordered tree *is a tuple $t = \langle T, E, r, l, c, n \rangle$ such that:*

i) (T, E, r) *is a rooted tree with root* $r \in T$*. Here,* T *is the set of tree nodes and* E *is the set of tree edges ([10]).*
ii) $l : T \to \Sigma$ *is a node labeling function.*
iii) $c \subseteq T \times T$ *is the parent-child relation between tree nodes.* $c = \{(v, u)|$ *node* u *is the parent of node* $v\}$*.*
iv) $n \subseteq T \times T$ *is the next-sibling linear ordering relation defined on the set of children of a node. For each node* $v \in T$*, its* k *children are ordered from left to right, i.e.* $(v_i, v_{i+1}) \in n$ *for all* $1 \leq i < k$*.*

A pattern is a labeled directed graph. Arc labels denote conditions that specify the tree delimiters of the extracted information, according to the parent-child and next-sibling relationships (eg. is there a parent node ?, is there a left sibling ?, a.o). Vertex labels specify conditions on nodes (eg. is the tag label *td* ?, is it the first child ?, a.o).

Conditions specified by arc and node labels must be satisfied by the extracted nodes and/or their delimiters. A subset of graph vertices is used for selecting the items for extraction.

We adopt a standard relational model. Associated to each information source is a set of distinct attributes. Let $\mathcal{A}$ be the set of attribute names.

Definition 2. *(Syntax) An* (extraction) pattern *is a tuple* $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ *such that:*

i) (V, A) *is a directed graph.* V *is the finite set of vertices and* $A \subseteq V \times V$ *is the set of directed edges or arcs.*
ii) $\lambda_a : A \to \{'c', 'n'\}$ *is the labeling function for arcs. The meanings of* $'c'$ *and* $'n'$ *are:* $'c'$ *label denotes the parent-child relation and* $'n'$ *label denotes the next-sibling relation.*
iii) $\lambda_c : V \to \mathcal{C}$ *is the labeling function for vertices. It labels each vertex with a condition from the set* $\mathcal{C} = \{\emptyset, \{'f'\}, \{'l'\}, \{\sigma\}, \{'f', 'l'\}, \{'f', \sigma\}, \{'l', \sigma\}, \{'f', 'l', \sigma\}\}$ *of conditions, where* σ *is a label in the set* Σ *of symbols. In this context, the meanings of* $'f'$*,* $'l'$ *and* σ *are:* $'f'$ *label requires the corresponding vertex to indicate a first child;* $'l'$ *label requires the corresponding vertex to indicate a last child;* σ *label requires the corresponding vertex to indicate a node labeled with* σ*.*
iv) $U = \{u_1, u_2, \ldots, u_k\} \subseteq V$ *is the set of* pattern extraction vertices *such that for all* $1 \leq i \leq k$*, the number of incoming arcs to* u_i *that are labeled with* $'c'$ *is 0.* k *is called the* pattern arity*.*
v) $D \subseteq \mathcal{A}$ *is the set of attribute names defining the relation scheme of the information source.* $\mu : D \to U$ *is a one-to-one function that assigns a pattern extraction vertex to each attribute name.*

Note that according to point iv) of definition 2, an extraction pattern does not state any condition about the descendants of an extracted node; i.e. it looks only at its siblings and its ancestors. This is not restrictive in the context of patterns for information extraction from HTML; see for example the rules in Elog$^-$, as described in [13].

In what follows, we provide a model-theoretic semantics for our patterns. In this setting, a labeled ordered tree is an interpretation domain for the patterns. The semantics is defined by an interpretation function assigning tree nodes to pattern vertices.

Definition 3. *(Interpretation) Let $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ be a pattern and let $t = \langle T, E, r, l, c, n \rangle$ be a labeled ordered tree. A function $I : V \rightarrow T$ assigning tree nodes to pattern vertices is called* interpretation.

Intuitively, patterns are matched against parts of a target labeled ordered tree. A successful matching asks for the labels of pattern vertices and arcs to be consistent with the corresponding relations and functions over tree nodes.

Definition 4. *(Semantics) Let $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ be a pattern, let $t = \langle T, E, r, l, c, n \rangle$ be a labeled ordered tree and let $I : V \rightarrow T$ be an interpretation function. Then I and t are* consistent *with p, written as $I, t \models p$, if and only if:*

i) If $(v, w) \in A$ and $\lambda_a((v, w)) = 'n'$ then $(I(v), I(w)) \in n$.
ii) If $(v, w) \in A$ and $\lambda_a((v, w)) = 'c'$ then $(I(v), I(w)) \in c$.
iii) If $v \in V$ and $'f' \in \lambda_c(v)$ then for all $w \in V$, $(I(w), I(v)) \notin n$.
iv) If $v \in V$ and $'l' \in \lambda_c(v)$ then for all $w \in V$, $(I(v), I(w)) \notin n$.
v) If $v \in V$ and $\sigma \in \Sigma$ and $\sigma \in \lambda_c(v)$ then $l(I(v)) = \sigma$.

A labeled ordered tree for which an interpretation function exists is called a model *of p.*

Our definition for patterns is quite general by allowing to build patterns for which no consistent labeled ordered tree and interpretation exist. Such patterns are called *inconsistent*. A pattern that is not inconsistent is called *consistent*. The following proposition states necessary conditions for pattern consistency.

Proposition 1. *(Necessary conditions for consistent patterns) Let $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ be a consistent pattern. Then:*

i) The graph of p is a DAG.
ii) For any two disjoint paths between two distinct vertices of p, one has length 1 and its single arc is labeled with $'c'$ and the other has length at least 2 and its arcs are all labeled with $'n'$, except the last arc that is labeled with $'c'$.
iii) For all $v \in V$ if $'f' \in \lambda_c(v)$ then for all $w \in V$, $(w, v) \notin A$ or $c((w, v)) =' c'$ and if $'l' \in \lambda_c(v)$ then for all $w \in V$, $(v, w) \notin A$ or $c((v, w)) =' c'$.

The proof of this proposition is quite straightforward. The idea is that a consistent pattern has at least one model and this model is a labeled ordered tree. Then, the claims of the proposition follow from the properties of ordered trees seen as directed graphs ([10]). Note that in what follows we are considering only consistent patterns.

The result of applying a pattern to a semi-structured information source is a set of extracted tuples. An extracted tuple is modeled as a function from attribute names to tree nodes, as in standard relational data modeling.

Definition 5. *(Extracted tuple) Let* $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ *be a pattern and let* $t = \langle T, E, r, l, c, n \rangle$ *be a labeled ordered tree that models a semi-structured information source. A tuple extracted by* p *from* t *is a function* $I \circ \mu : D \rightarrow T$[1]*, where* I *is an interpretation function such that* $I, t \models p$.

Note that if p is a pattern and t is a tree then p is able to extract more than one tuple from t. Let $Ans(p, t)$ be the set of all tuples extracted by p from t.

3 Pattern Properties and Operations

In this section we study pattern properties – subsumption, equivalence and operations – simplification and merging.

Subsumption and equivalence enable the study of pattern simplification, i.e. the process of removing arcs in the pattern directed graph without changing the pattern semantics. Merging is useful in practice for constructing patterns of a higher arity from two or more patterns of smaller arities (see the example in section 4).

3.1 Pattern Subsumption and Equivalence

Pattern subsumption refers to checking when the set of tuples extracted by a pattern is subsumed by the set of tuples extracted by a second (possibly simpler) pattern. Two patterns are equivalent when they subsume each other.

Definition 6. *(Pattern subsumption and equivalence) Let* p_1 *and* p_2 *be two patterns of arity* k. p_1 subsumes p_2*, written as* $p_1 \preceq p_2$*, if and only if for all trees* t*,* $Ans(p_1, t) \subseteq Ans(p_2, t)$*. If the two patterns mutually subsume each other, i.e.* $p_1 \preceq p_2$ *and* $p_2 \preceq p_1$*, then they are called* equivalent*, written as* $p_1 \simeq p_2$.

In practice, given a pattern p, we are interested in simplifying p to yield a new pattern p' equivalent to p.

Proposition 2. *(Pattern simplification) Let* $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ *be a pattern and let* $u, v, w \in V$ *be three distinct vertices of* p *such that* $(u, w) \in A$*,* $\lambda_a((u, w)) = 'c'$*,* $(u, v) \in A$*, and* $\lambda_a((u, v)) =' n'$*. Let* $p' = \langle V, A', U, D, \mu, \lambda'_a, \lambda_c \rangle$ *be a pattern defined as:*

i) $A' = (A \setminus \{(u, w)\}) \cup \{(v, w)\}$.
ii) If $x \in A \setminus \{(u, w)\}$ *then* $\lambda'_a(x) = \lambda_a(x)$*, and* $\lambda'_a((v, w)) = 'c'$.

Then $p' \simeq p$.

Basically, this proposition says that shifting one position right an arc labeled with $'c'$ in a pattern produces an equivalent pattern. The result follows from the property that for all nodes u, v, w of an ordered tree such that v is the next sibling of u then w is the parent of u if and only if w is the parent of v. Note

[1] The $\circ$ operator denotes function composition.

that if $(v, w) \in A$ then the consistency of p enforces $\lambda_a((v, w)) = 'c'$ and this results in no new arc being added to p'. In this case p gets simplified to p' by deleting arc (u, w).

A pattern p can be simplified to an equivalent pattern p' called normal form.

Definition 7. *(Pattern normal form) A pattern p is said to be in* normal form *if the out-degree of every pattern vertex is at most 1.*

A pattern can be brought to normal form by repeatedly applying the operation described in proposition 2. The existence of a normal form is captured by the following proposition.

Proposition 3. *(Existence of normal form) For every pattern p there exists a pattern p' in normal form such that $p' \simeq p$.*

Note that the application of pattern simplification operation from proposition 2 has the result of decrementing by 1 the number of pattern vertices with out-degree equal to 2. Because the number of pattern vertices is finite and the out-degree of each vertex is at most 2, it follows that after a finite number of steps the resulted pattern will be brought to normal form.

3.2 Pattern Merging

Merging looks at building more complex patterns by combining simpler patterns. In practice we found convenient to learn a set of simpler patterns that share attributes and then merge them into more complex patterns, that are capable to extract tuples of higher arity.

Merging two patterns first assumes performing a pairing of their pattern vertices. Two vertices are paired is they are meant to match identical nodes of the target document. Paired vertices will be fusioned in the resulting pattern.

Definition 8. *(Pattern vertex pairings) Let $p_i = \langle V_i, A_i, U_i, D_i, \mu_i, \lambda_{a_i}, \lambda_{c_i} \rangle$, $i = 1, 2$, be two patterns such that $V_1 \cap V_2 = \emptyset$. The set of vertex pairings of p_1 and p_2 is the maximal set $P \subseteq V_1 \times V_2$ such that:*

- *i) For all $d \in D_1 \cap D_2$, $(\mu_1(d), \mu_2(d)) \in P$.*
- *ii) If $(u_1, u_2) \in P$, $(u_1, v_1) \in A_1$, $(u_2, v_2) \in A_2$, and $\lambda_{a_1}((u_1, v_1)) = \lambda_{a_2}((u_2, v_2)) =' n'$ then $(v_1, v_2) \in P$.*
- *iii) If $(u_1, u_2) \in P$, $w_0 = u_1, w_1, \ldots, w_n = v_1$ is a path in (V_1, A_1) such that $\lambda_{a_1}((w_i, w_{i+1})) = 'n'$ for all $1 \leq i < n - 1$, $\lambda_{a_1}((w_{n-1}, w_n)) = 'c'$, and $w'_0 = u_2, w'_1, \ldots, w'_m = v_2$ is a path in (V_2, A_2) such that $\lambda_{a_2}((w'_i, w'_{i+1})) = 'n'$ for all $1 \leq i < m - 1$, $\lambda_{a_2}((w'_{m-1}, w'_m)) = 'c'$ then $(v_1, v_2) \in P$.*
- *iv) If $(u_1, u_2) \in P$, $(v_1, u_1) \in A_1$, $(v_2, u_2) \in A_2$, and $\lambda_{a_1}((u_1, v_1)) = \lambda_{a_2}((u_2, v_2)) =' n'$ then $(v_1, v_2) \in P$.*
- *v) If $(u_1, u_2) \in P$, $(v_1, u_1) \in A_1$, $(v_2, u_2) \in A_2$, $\lambda_{a_1}((u_1, v_1)) = \lambda_{a_2}((u_2, v_2)) = 'c'$, and $('f' \in \lambda_{c_1}(v_1) \cap \lambda_{c_2}(v_2)$ or $'l' \in \lambda_{c_1}(v_1) \cap \lambda_{c_2}(v_2))$, then $(v_1, v_2) \in P$.*

Defining vertex pairings according to definition 8 deserves some explanations. Point i) states that if two extraction vertices denote identical attributes then they must be paired. Points ii), iii), iv) and v) identify additional pairings based on properties of ordered trees. Points ii) and iii) state that next-siblings or parents of paired vertices must be paired as well. Point iv) states that previous siblings of paired vertices must be paired as well. Point v) state that first children and respectively last children of paired vertices must be paired as well.

For all pairings (u, v), the paired vertices u and v are fusioned into a single vertex that is labeled with the union of the conditions of the original vertices, assuming that these conditions are not mutually inconsistent.

First, we must define the fusioning of two vertices of a directed graph.

Definition 9. *(Vertex fusioning) Let $G = (V, A)$ be a directed graph and let $u, v \in V$ be two vertices such that $u \neq v$, $(u, v) \notin A$, and $(v, u) \notin A$. The graph $G' = (V', A')$ obtained by fusioning vertex u with vertex v is defined as:*

i) $V' = V \setminus \{v\}$;
ii) *A' is obtained by replacing each arc $(x, v) \in A$ with (x, u) and each arc $(v, x) \in A$ with (u, x).*

Pattern merging involves the repeated fusioning of vertices of the pattern vertex pairings. For each paired vertices, their conditions must be checked for mutual consistency.

Definition 10. *(Pattern merging) Let $p_i = \langle V_i, A_i, U_i, D_i, \mu_i, \lambda_{a_i}, \lambda_{c_i} \rangle$, $i = 1, 2$, be two patterns such that $V_1 \cap V_2 = \emptyset$ and let P be the set of vertex pairings of p_1 and p_2. If for all $(u, v) \in P$ and for all $\sigma_1, \sigma_2 \in \Sigma$, if $\sigma_1 \in \lambda_{c_1}(u)$ and $\sigma_2 \in \lambda_{c_2}(v)$ then $\sigma_1 = \sigma_2$, then the pattern $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ resulted from* merging *patterns p_1 and p_2 is defined as follows:*

i) *(V, A) is obtained by fusioning u with v for all $(u, v) \in P$ in graph $(V_1 \cup V_2, A_1 \cup A_2)$.*
ii) *$U = U_1 \cup U_2$. $D = D_1 \cup D_2$. If $d \in D_1$ then $\mu(d) = \mu_1(d)$, else $\mu(d) = \mu_2(d)$.*
iii) *For all $(u, v) \in V$ if $u, v \in V_1$ then $\lambda_a((u, v)) = \lambda_{a_1}((u, v))$ else if $u, v \in V_2$ then $\lambda_a((u, v)) = \lambda_{a_2}((u, v))$ else if $u \in V_1$, $v \in V_2$ and $(u, u') \in P$ then $\lambda_a((u, v)) = \lambda_{a_2}((u', v))$ else if $u \in V_2$, $v \in V_1$ and $(v, v') \in P$ then $\lambda_a((u, v)) = \lambda_{a_2}((u, v'))$.*
iv) *If a vertex in $x \in V$ resulted from fusioning u with v then $\lambda_c(x) = \lambda_{c_1}(u) \cup \lambda_{c_2}(v)$, else if $x \in V_1$ then $\lambda_c(x) = \lambda_{c_1}(x)$, else $\lambda_c(x) = \lambda_{c_2}(x)$.*

Essentially this definition says that pattern merging involves performing a pattern vertex pairing (point i)), then defining of the attributes attached to pattern extraction vertices (point ii)) and of the labels attached to vertices (point iv)) and arcs (point iii)) in the directed graph of the resulting pattern.

An example of pattern merging is given in the next section of the paper. Despite these somehow cumbersome but rigorous definitions, pattern merging is a quite simple operation that can be grasped more easily using a graphical representation of patterns (see figure 2).

The next proposition states that the set of tuples extracted by a pattern resulted from merging two or more patterns is equal to the relational natural join of the sets of tuples extracted by the original patterns.

Proposition 4. *(Tuples extracted by a pattern resulted from merging) Let p_1 and p_2 be two patterns and let p be their merging. For all labeled ordered trees t, $Ans(p, t) = Ans(p_1, t) \bowtie Ans(p_2, t)$. $\bowtie$ is the relational natural join operator.*

This result follows by observing that a pattern can be mapped to a conjunctive query over the signature $(child, next, first, last, (tag_\sigma)_{\sigma \in \Sigma})$. Relations $child$, $next$, $first$, $last$ and tag_σ are defined as follows (here $\mathcal{N}$ is the set of tree nodes):
i) $child \subseteq \mathcal{N} \times \mathcal{N}$, $(child(P, C) = true) \Leftrightarrow$ (P is the parent of C).
ii) $next \subseteq \mathcal{N} \times \mathcal{N}$, $(next(L, N) = true) \Leftrightarrow$ (L is the left sibling of N).
iii) $first \subseteq \mathcal{N}$, $(first(X) = true) \Leftrightarrow$ (X is the first child of its parent node).
iv) $last \subseteq \mathcal{N}$, $(last(X) = true) \Leftrightarrow$ (X is the last child of its parent node).
v) $tag_\sigma \subseteq \mathcal{N}$, $(tag(N) = true) \Leftrightarrow$ (σ is the tag of node N).
A pattern vertex is mapped to a logic variable. The query defines a predicate with variables derived from the pattern extraction vertices, one variable per pattern vertex. Merging involves renaming with identical names the variables corresponding to paired pattern vertices and then taking the conjunction of queries corresponding to merged patterns. Now, by simple relational manipulation, it is easy to see that the result stated by proposition 4 holds.

4 An Example

Here we show how the theory developed in the previous sections can be applied to obtain practical wrappers for HTML, implemented as XSLT stylesheets.

First, using the observation stated at the end of the previous section, we redefine L-wrappers ([5, 6]) as sets of patterns. Second, we consider the two single-clause wrappers from [6] and describe them as single-pattern wrappers. Third, we consider the pattern resulted from their merging. Finally, we map the resulting wrapper to XSLT and give arguments for its correctness. Note that in this mapping we are using a subset of XSLT that has a formal semantics ([7]).

4.1 L-wrappers as Sets of Patterns

L-wrappers (introduced in [5, 6]) can be redefined using patterns as follows.

Definition 11. *(L-wrapper) An* L-wrapper *of arity k is a set of $n \geq 1$ patterns $W = \{p_i | p_i = \langle V_i, A_i, U_i, D, \mu_i, \lambda_{a_i}, \lambda_{c_i} \rangle$, p_i has arity k, for all $1 \leq i \leq n\}$. The set of tuples extracted by W from a labeled ordered tree t is the union of the sets of tuples extracted by each pattern p_i, $1 \leq i \leq n$, i.e. $Ans(W, t) = \cup_{i=1}^{n} Ans(p_i, t)$.*

In [6] we considered learning L-wrappers for extracting printer information from Hewlett Packard's Web site, using general-purpose ILP. There, we also proposed a generic process for information extraction from the Web that consists of

the following stages: page collection, pre-processing, manual information extraction, conversion to the input format of the learning program, learning, wrapper compilation, wrapper execution. In this section we are focusing on the last two stages of this process: wrapping compilation, i.e. the mapping of L-wrappers to XSLT and wrapper execution.

The printer information is represented in multi-section two column HTML tables (see figure 1). Each row contains a pair (feature name, feature value). Consecutive rows represent related features that are grouped into feature classes. For example, there is a row with the feature name 'Print technology' and the feature value 'HP Thermal Inkjet'. This row has the feature class 'Print quality/technology'. So actually this table contains triples (feature class, feature name, feature value). Some triples may have identical feature classes.

Speed/monthly volume	
Print speed, black (pages per minute)	Up to 15 ppm
Print speed, color (pages per minute)	Up to 11 ppm
Recommended monthly volume, maximum	12,000 pages
Print quality / technology	
Print technology	HP Thermal Inkjet
Print quality, black	up to 1200 x 600 dpi
Print quality, color	up to 1200 x 600 dpi on photo paper
Resolution technology	HP PhotoREt III
Paper handling / media	
Paper trays, std.	2
Paper trays, max.	2

Fig. 1. An XHTML document fragment and its graphic view

In [6] we presented two single-clause L-wrappers for this example that were learnt using FOIL program ([18]): i) for pairs (feature class, feature name); ii) for pairs (feature name, feature value), together with figures of the precision and recall performance measures. The wrappers are ($FC =$ feature class, $FN =$ feature name, $FV =$ feature value):

$$\begin{aligned} & extract(FC, FN) \leftarrow child(C, FC) \wedge child(D, FN) \wedge tag(C, span) \wedge child(E, C) \wedge \\ & \quad child(F, E) \wedge next(F, G) \wedge child(H, G) \wedge last(E) \wedge child(I, D) \wedge child(J, I) \wedge \\ & \quad child(K, J) \wedge child(L, K) \wedge next(L, M) \wedge child(N, M) \wedge child(H, N). \\ & extract(FN, FV) \leftarrow tag(FN, text) \wedge tag(FV, text) \wedge child(C, FN) \wedge child(D, FV) \wedge \\ & \quad child(E, C) \wedge child(F, E) \wedge child(G, D) \wedge child(H, G) \wedge child(I, F) \wedge \\ & \quad child(J, I) \wedge next(J, K) \wedge first(J) \wedge child(K, L) \wedge child(L, H). \end{aligned}$$

Figure 2 illustrates the two patterns corresponding to the clauses shown above and the pattern resulted from their merging. One can easily notice that these patterns are already in normal form.

The experimental analysis performed in [6] also revealed some difficulties of learning triples (feature class, feature name, feature value) in a straightforward way. The problems were caused by the exponential growth of the number of negative examples, with the tuples arity. The idea of pattern merging presented here

is an approach of learning extraction patterns of a higher arity that overcomes these difficulties, and thus supporting the scalability of our approach.

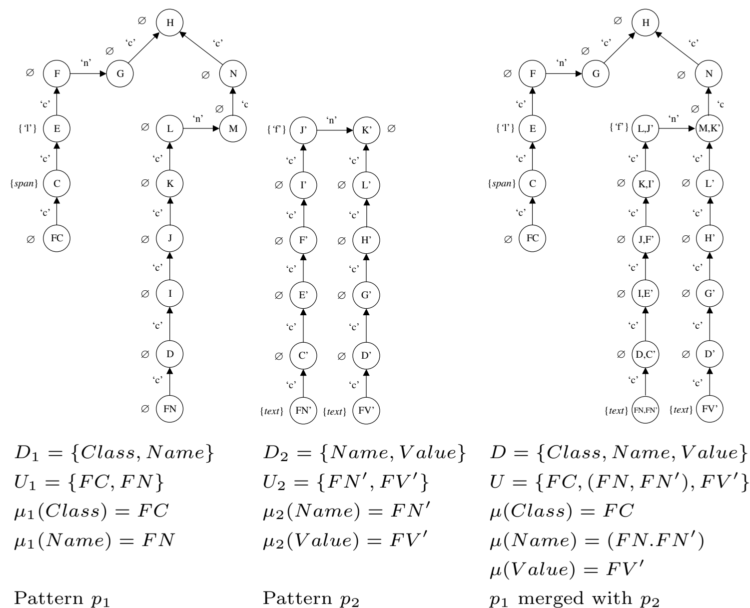

Fig. 2. Patterns and pattern merging

4.2 Mapping Wrappers to XSLT

Paper [7] describes a subset of XSLT, called XSLT_0, that has a Plotkin-style formal semantics. The reader is invited to consult reference [7], for details on XSLT_0, its pseudocode notation and the formal semantics.

The XSLT_0 description of the single-pattern wrapper resulted from merging patterns p_1 and p_2 from figure 2 is shown in table 1[2]. The XSLT wrapper is shown in the appendix. XPath expressions xp_1, xp_2 and xp_3 are defined as follows:

```
xp1 = //*/*/preceding-sibling::*[1]/*[last()]/span/node()
xp2 = parent::*/parent::*/parent::*/following-sibling::*[1]/parent::*/*/*/
      preceding-sibling::*[1][last()]/*/*/*/*/text()
xp3 = parent::*/parent::*/parent::*/parent::*/parent::*/following-sibling::*[1]/
      */*/*/*/text()
```

The idea is simple: referring to figure 2, we start from the document root, labeled with *html*, then match node H and move downwards to FC, then move back upwards from FC to H and downwards to (FN, FN'), and finally move back upwards from (FN, FN') to (M, K') and downwards from (M, K') to FV'. The wrapper actually extracts the node contents rather than the nodes them-

[2] Note that our version of XSLT_0 is slightly different from the one presented in [7].

selves, using the *content*(.) expression. The extracted information is passed between templates in template variables and parameters $varClass$ and $varName$.

```
template start(html)                template selclass(*)
    return                              vardef
        result(selclass(xp1))               varClass := content(.)
end                                     return
                                            selname(xp2,varClass)
                                    end

template selname(*,varClass)        template display(*,varClass,varName)
    vardef                              vardef
        varName := content(.)               varValue := content(.)
    return                              return
        display(xp3,varClass,varName)       triple[class→ varClass;
end                                                name→ varName;
                                                   value→ varValue]
                                    end
```

Table 1. Description of the sample wrapper in $XSLT_0$ pseudocode

Note that this technique works for the general case of mapping a pattern to XSLT. As the pattern is a tree, it is always possible to move from a pattern extraction vertex to another via their common descendant in the pattern graph.

Below we give an informal argument of why this transformation works.

The $XSLT_0$ program contains four templates: i) the constructing templates $t_1 = start(html)$, $t_4 = display(*, varClass, varName)$, and ii) the selecting templates $t_2 = selclass(*)$, $t_3 = selname(*, varClass)$.

Initially, t_1 is applied. xp_1 selects the nodes that match the feature class. Then, for each feature class, t_2 is applied. xp_2 selects the feature names that are members of a feature class. Then, for each pair (feature class, feature name), t_3 is applied. xp_3 selects the feature values that are related to a pair (feature class, feature name). Then, for each triple (feature class, feature name, feature value), t_4 is applied. It constructs a triple and adds it to the output XML document.

For wrapper execution we can use any of the available XSLT transformation engines. In our experiments we have used Oxygen XML editor ([17], a tool that incorporates some of these engines. Figure 3 illustrates our wrappers in action.

5 Related Work

With the rapid expansion of the Internet and the Web, the field of information extraction from HTML attracted a lot of researchers during the last decade. Clearly, it is impossible to mention all of their work here. However, at least we can try to classify these works along several axes and select some representatives for discussion.

First, we are interested in research on information extraction from HTML using logic representations of tree (rather than string) wrappers that are generated automatically using techniques inspired by ILP. Second, both theoretical and experimental works are considered.

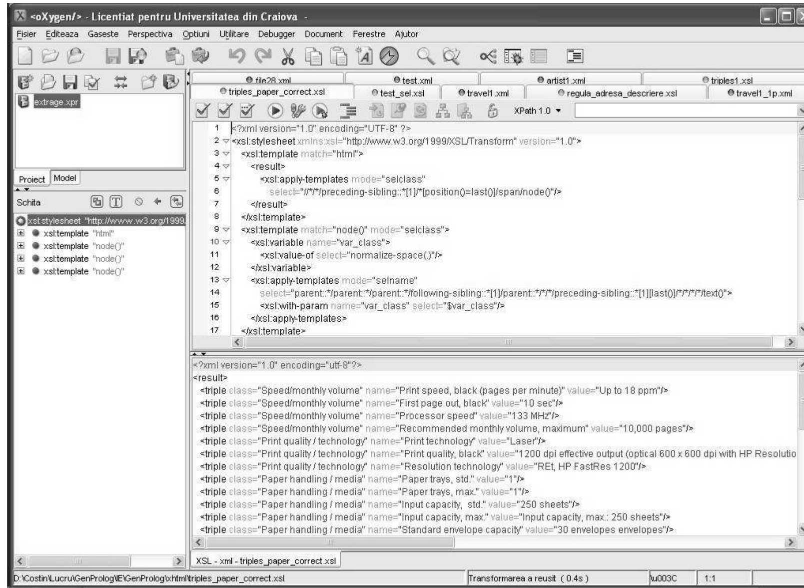

Fig. 3. Wrapper execution inside Oxygen XML editor

[11] is one of the first papers describing a "relational learning program" called SRV. It uses a FOIL-like algorithm for learning first order information extraction rules from a text document represented as a sequence of lexical tokens. Rule bodies check various token features like: length, position in the text fragment, if they are numeric or capitalized, a.o. SRV has been adapted to learn information extraction rules from HTML. For this purpose new token features have been added to check the HTML context in which a token occurs. The most important similarity between SRV and our approach is the use of relational learning and a FOIL-like algorithm. The difference is that our approach has been explicitly devised to cope with tree structured documents, rather than string documents.

In [8] is described a generalization of the notion of string delimiters developed for information extraction from string documents ([14]) to subtree delimiters for information extraction from tree documents. The paper describes a special purpose learner that constructs a structure called candidate index based on trie data structures, which is very different from FOIL's approach. Note however, that the tree leaf delimiters described in that paper are very similar to our information extraction rules. Moreover, the representation of reverse paths using the symbols $Up(\uparrow)$, $Left(\leftarrow)$ and $Right(\rightarrow)$ can be easily simulated by our rules using the relations *child* and *next*.

In [20] is proposed a technique for generating XSLT-patterns from positive examples via a GUI tool and using an ILP-like algorithm. The result is a NE-agent (i.e. name extraction agent) that is capable of extracting individual items. A TE-agent (i.e. term extraction agent) then uses the items extracted by NE-agents and global constraints to fill-in template slots (tuple elements according

to our terminology). The differences in our work are: XSLT wrappers are learnt indirectly via L-wrappers; our wrappers are capable of extracting tuples in a straightforward way, therefore TE-agents are not needed.

In [3] is described Elog, a logic Web extraction language. Elog is employed by a visual wrapper generator tool called Lixto. Elog uses a tree representation of HTML documents (similar to our work) and defines Datalog-like rules with patterns for information extraction. Elog is very versatile by allowing the refinement of the extracted information with the help of regular expressions and the integration between wrapping and crawling via links in Web pages. Elog uses a dedicated extraction engine that is incorporated into Lixto tool.

In [19] is introduced a special wrapper language for Web pages called token-templates. Token-templates are constructed from tokens and token-patterns. A Web document is represented as a list of tokens. A token is a feature structure with exactly one feature having name *type*. Feature values maybe either constants or variables. Token-patterns use operators from the language of regular expressions. The operators are applied to tokens to extract relevant information. The only similarity between our approach and this approach is the use of logic programming to represent wrappers.

In [16] is described the DEByE (i.e. Data Extraction By Example) environment for Web data management. DEByE contains a tool that is capable to extract information from Web pages based on a set of examples provided by the user via a GUI. The novelty of DEByE is the possibility to structure the extracted data based on the user perception of the structure present in the Web pages. This structure is described at example collection stage by means of a GUI metaphor called nested tables. DEByE addresses also other issues needed in Web data management like automatic examples generation and wrapper management. Our L-wrappers are also capable of handling hierarchical information. However, in our approach, the hierarchical structure of information is lost by flattening during extraction (see the printer example where tuples representing features of the same class share the feature class attribute).

As concerning theoretical work, [13] is one of the first papers that analyzes seriously the expressivity required by tree languages for Web information extraction and its practical implications. Combined complexity and expressivity results of conjunctive queries over trees, that also apply to information extraction, are reported in [12].

Finally, in [15] is contained a survey of Web data extraction tools. That paper contains a section on wrapper languages including HTML-aware tools (tree wrappers) and a section on wrapper induction tools.

6 Concluding Remarks

In this paper we studied a class of patterns for information extraction from semi-structured data inspired by logic. This complements our experimental work reported in [4–6]. There, we described how to apply ILP to learn L-wrappers for information extraction from the Web. Here the main focus were properties and

operations of patterns used by L-wrappers and L-wrapper implementation using XSLT for information extraction from HTML. The results of this work provide a theoretical basis of L-wrappers and their patterns linking our work with related works in this field. They also show how the combination of general-purpose ILP, L-wrappers and XSLT transformations can be successfully applied to extract information from the Web. We plan to implement this in an information extraction tool. As future theoretical work, we would like to give a formal proof of the correctness of the mapping of L-wrappers to XSLT. As future experimental work we plan to investigate the generality of our approach by applying it on Web sites in other application areas.

References

1. Abiteoul, S., Buneman, P., Suciu, D.: *Data on the Web: From Relations to Semistructured data and XML*, Morgan Kauffman Publishers, (2000).
2. Aleph. `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.html`.
3. Baumgartner, R., Flesca, S., Gottlob, G.: The Elog Web Extraction Language. In: Nieuwenhuis, R., Voronkov, A. (eds.): *Proceedings of LPAR'2001*, LNAI 2250, Springer Verlag, (2001) 548–560.
4. Bădică, C., Bădică, A.: Rule Learning for Feature Values Extraction from HTML Product Information Sheets. In: Boley, H., Antoniou, G. (eds): *Proc. RuleML'04*, Hiroshima, Japan. LNCS 3323 Springer-Verlag (2004) 37–48.
5. Bădică, C., Popescu, E., Bădică, A.: Learning Logic Wrappers for Information Extraction from the Web. In: Papazoglou M., Yamazaki, K. (eds.) *Proc. SAINT'2005 Workshops. Computer Intelligence for Exabyte Scale Data Explosion*, Trento, Italy. IEEE Computer Society Press (2005) 336–339.
6. Bădică, C., Bădică, A., Popescu, E.: Tuples Extraction from HTML Using Logic Wrappers and Inductive Logic Programming. In: Szczepaniak, P.S., Kacprzyk, J., Niewiadomski, A. (eds.): *Proc.AWIC'05*, Lodz, Poland. LNAI 3528 Springer-Verlag (2005) 44–50.
7. Bex, G.J., Maneth, S., Neven, F.: A formal model for an expressive fragment of XSLT. *Information Systems*, No.27, Elsevier Science (2002) 21–39.
8. Chidlovskii, B.: Information Extraction from Tree Documents by Learning Subtree Delimiters. In: *Proc. IJCAI-03 Workshop on Information Integration on the Web* (IIWeb-03), Acapulco, Mexico (2003) 3–8.
9. Clark, J.: XSLT Transformation (XSLT) Version 1.0, W3C Recommendation, 16 November 1999, `http://www.w3.org/TR/xslt` (1999).
10. Cormen, T.H., Leiserson, C.E., Rivest, R.R.: *Introduction to Algorithms.* MIT Press (1990).
11. Freitag, D.: Information extraction from HTML: application of a general machine learning approach. In: *Proceedings of AAAI'98*, (1998) 517–523.
12. Gottlob, G., Koch, C., Schulz, K.U.: Conjunctive Queries over Trees. In: *Proc.PODS'2004*, Paris, France. ACM Press, (2004) 189–200.
13. Gottlob, G., Koch, C.: Monadic Datalog and the Expressive Power of Languages for Web Information Extraction. In: *Journal of the ACM*, Vol.51, No.1 (2004) 74–113
14. Kushmerick, N., Thomas, B.: Adaptive Information Extraction: Core Technologies for Information Agents, In: *Intelligent Information Agents R&D in Europe: An AgentLink perspective* (Klusch, Bergamaschi, Edwards & Petta, eds.). LNCS 2586, Springer-Verlag (2003).

15. Laender, A.H.F., Ribeiro-Neto, B., Silva, A.S., Teixeira., J.S.: A Brief Survey of Web Data Extraction Tools. In: *SIGMOD Record*, Vol.31, No.2, ACM Press (2002) 84–93.
16. Laender, A.H.F., Ribeiro-Neto, B., Silva, A.S.: DEByE - Data Extraction By Example. In: *Data & Knowledge Engineering* Vol.40, No.2, (2002) 121–154.
17. Oxygen XML Editor. `http://www.oxygenxml.com/`.
18. Quinlan, J. R., Cameron-Jones, R. M.: Induction of Logic Programs: FOIL and Related Systems, *New Generation Computing*, 13, (1995) 287–312.
19. Thomas, B.: Token-Templates and Logic Programs for Intelligent Web Search. *Intelligent Information Systems*. Special Issue: Methodologies for Intelligent Information Systems, 14(2/3) (2000) 241-261.
20. Xiao, L., Wissmann, D., Brown, M., Jablonski, S.: Information Extraction from HTML: Combining XML and Standard Techniques fro IE from the Web. In: Monostori, L., Vancza, J., Ali, M. (eds.): *Proc. IEA/AIE 2001*. LNAI 2070, Springer-Verlag (2001) 165–174.

A XSLT Code of the Sample Wrapper

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="html">
    <result>
      <xsl:apply-templates mode="selclass" select="//*/*/preceding-sibling::*[1]/
      *[last()]/span/node()"/>
    </result>
  </xsl:template>
  <xsl:template match="node()" mode="selclass">
    <xsl:variable name="var_class"> <xsl:value-of select="normalize-space(.)"/>
    </xsl:variable>
    <xsl:apply-templates mode="selname" select="parent::*/parent::*/parent::*/
      following-sibling::*[1]/parent::*/*/*/preceding-sibling::*[1][last()]/
      */*/*/*/text()">
      <xsl:with-param name="var_class" select="$var_class"/>
    </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="node()" mode="selname">
    <xsl:param name="var_class"/>
    <xsl:variable name="var_name"> <xsl:value-of select="normalize-space(.)"/>
    </xsl:variable>
    <xsl:apply-templates mode="display" select="parent::*/parent::*/parent::*/parent::*/
      parent::*/following-sibling::*[1]/*/*/*/*/text()">
      <xsl:with-param name="var_class" select="$var_class"/>
      <xsl:with-param name="var_name" select="$var_name"/>
    </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="node()" mode="display">
    <xsl:param name="var_class"/>
    <xsl:param name="var_name"/>
    <xsl:variable name="var_value"> <xsl:value-of select="normalize-space(.)"/>
    </xsl:variable>
    <triple>
      <xsl:attribute name="class"> <xsl:value-of select="$var_class"/>
      </xsl:attribute>
      <xsl:attribute name="name"> <xsl:value-of select="$var_name"/>
      </xsl:attribute>
      <xsl:attribute name="value"> <xsl:value-of select="$var_value"/>
      </xsl:attribute>
    </triple>
  </xsl:template>
</xsl:stylesheet>
```