Soft Comput (2007) 11:753–772
DOI 10.1007/s00500-006-0118-y

FOCUS

L-wrappers: concepts, properties and construction

A declarative approach to data extraction from web sources

Costin Bădică · Amelia Bădică · Elvira Popescu · Ajith Abraham

Published online: 6 July 2006

Abstract In this paper, we propose a novel class of wrappers (*logic wrappers*) inspired by the logic programming paradigm. The developed Logic wrappers (L-wrapper) have declarative semantics, and therefore: (i) their specification is decoupled from their implementation and (ii) they can be generated using inductive logic programming. We also define a convenient way for mapping L-wrappers to XSLT for efficient processing using available XSLT processing engines.

Keywords Logic wrappers · Information retrieval · Web intelligence · Inductive logic programming

C. Bădică (✉)
Software Engineering Department,
University of Craiova,
Bvd.Decebal 107, Craiova 200440, Romania
e-mail: badica_costin@software.ucv.ro

A. Bădică
Business Information Systems Department,
University of Craiova,
A.I.Cuza 13, Craiova 200585, Romania

E. Popescu
Software Engineering Department,
University of Craiova,
Bvd.Decebal, Craiova 200440, Romania
e-mail: popescu_elvira@software.ucv.ro

A. Abraham
IITA Professorship Program,
School of Computer Science and Engineering,
Chung-Ang University,
221, Heukseok-dong, Dongjak-gu Seoul 156-756,
Republic of Korea
e-mail: ajith.abraham@ieee.org

1 Introduction

The Web is now a huge information repository that is characterized by (i) *high diversity*, i.e. the Web information covers almost any application area, (ii) *disparity*, i.e. the Web information comes in many formats ranging from plain and structured text to multimedia documents and (iii) *rapid growth*, i.e. old information is continuously being updated in form and content and new information is constantly being produced. The HTML markup language is the *lingua franca* for publishing information on the Web.

The Web was designed for human consumption rather than machine processing. Web pages are designed by humans and are targeted to human consumers that seek specialized information in various areas of interest. That information can be reused for different problem solving purposes; in particular it can be searched, filtered out, processed, analyzed, and reasoned about.

For example, in the e-tourism domain one can note an increasing number of travel agencies offering online services through online transaction brokers [21]. They provide to human users useful information about hotels, flights, trains or restaurants, in order to help them planning their business or holiday trips. Travel information, like most of the information published on the Web, is heterogeneous and distributed, and there is a need to gather, search, integrate and filter it efficiently [17] and ultimately to enable its reuse for multiple purposes. The high diversity and disparity of Web content hinder its automatic processing. In our opinion, this explains the significant growth of the interest in the field of automatizing the tasks of data extraction from the Web and this interest is expected to grow, as the Web is continuously growing in both size and complexity.

Another interesting use of data harvested from semi-structured Web sources that has been recently proposed [3] is to feed business intelligence tasks, in areas like competitive analysis and intelligence. According to [3], business intelligence can be broadly understood as an insight into a company and its chain of actions. Therefore, developing tools that help automate the process of data extraction and integration about competitors from public information sources on the Web, is of primary importance.

Two emergent technologies that have been put forward to enable automated processing of information published on the Web are semantic markup and Web services. However, most of the current practices in Web publishing are still being based on the combination of traditional HTML – *lingua franca* for Web publishing, with server-side dynamic content generation from databases. Moreover, many Web pages are using HTML elements that were originally intended for use to structure content (e.g. those elements related to tables), for layout and presentation effects, even if this practice is not encouraged in theory. Therefore, techniques developed in areas like information extraction, machine learning and wrapper induction are still expected to play a significant role in tackling the problem of Web data extraction.

Many Web sources can be nicely abstracted as providing relational data as sets of tuples, including: search engines result pages, product catalogues, news sites, product information sheets, travel resources, multimedia repositories, Web directories, etc. *Data extraction* is related to the more general problem of information extraction that is traditionally associated with artificial intelligence and natural language processing. *Information extraction* was developed as part of DARPA's MUC program and originally was concerned with locating specific pieces of information in text documents written in natural language [22] and then using them to populate a database or structured document. The field then expanded rapidly to cover extraction tasks from networked documents including HTML and XML and attracted other communities including databases, electronic documents and Web technologies. Usually, the content of these data sources can be characterized as neither natural language, nor structured, and therefore usually the term semi-structured data is used. For these cases we consider that the term *data extraction* is more appropriate than information extraction and consequently, we shall use it in the rest of this paper.

One area that can be described as a success story for the application of traditional machine learning technologies (like inductive logic programming) is wrapper induction for data extraction from the Web. A *wrapper* is a program that is actually used for performing the data extraction task. Manual creation of wrappers is a tedious, error-prone and quite difficult task, and therefore, a lot of techniques for (semi-)automatic wrapper construction have been proposed. Some of these approaches [18] propose a generic wrapper written in a procedural programming language that is then specialized for a particular data extraction task by instantiating its set of parameters. The values of the parameters are determined using machine learning.

In this paper we propose a novel class of wrappers inspired by the logic programming paradigm – *L-wrappers* (i.e. *logic wrappers*). Our wrappers (i) have a declarative semantics, and therefore their specification is decoupled from their implementation and (ii) can be generated using inductive logic programming. In particular, we outline an approach for the efficient implementation of L-wrappers using XSLT transformation language [12] – a standard language for processing XML documents.

This study serves at least the following purposes: (i) to give a concise specification of L-wrappers; this enables a theoretical investigation of their properties and operations and allows comparisons with related works; (ii) to devise an approach for semi-automatic construction of L-wrappers within the framework of inductive logic programming; (iii) to define 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). In our opinion, one advantage of our proposal is the use of the right tool for tackling a given task, i.e. for learning extraction rules it employs inductive logic programming systems [26], and for performing the extraction it employs XSLT technology [12].

The paper is structured as follows. First, 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 ([3,11,14,16,18,20,28,29,23]; see also the survey in [19]). Therefore, we start with a summary of these relevant works, briefly comparing them with our own work. Second, in Sect. 3 we follow with a concise description of 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. Third, in Sect. 4 we show how the task of semi-automatic construction of

L-wrappers can be mapped to inductive logic programming. In particular, we show how FOIL system [26] can be used for achieving this task. This section contains also a brief description of the software tools that we have used in our experiments together with experimental results in various settings. In Sect. 5 we outline an algorithm for mapping L-wrappers to a subset of XSLT. Then we show how L-wrappers technology was successfully applied in the e-travel domain, on a non-trivial example – data extraction from Travelocity[1] Web site. The last section of the paper contains some concluding remarks and points to future research directions.

2 Wrappers for Web sources: related research

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.

Freitag [14] 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, etc. 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 [11] is described a generalization of the notion of string delimiters developed for information extraction from string documents [18] 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 [29] a technique is proposed 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 [2] Elog is described, 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 paper [28] 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 may be 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 paper [20] 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

[1] http://www.travelocity.com

example where tuples representing features of the same class share the feature class attribute).

In paper [27] tree wrappers for tuples extraction are introduced. A tree wrapper is a sequence of tree extraction paths. There is an extraction path for each extracted attribute. A tree extraction path is a sequence of triples that contain a tag, a position and a set of tag attributes. A triple matches a node based on the node tag, its position among its siblings with a similar tag and its attributes. Extracted items are assembled into tuples by analyzing their relative document order. The algorithm for learning a tree extraction path is based on the composition operation of two tree extraction paths. Note also that L-wrappers use a different and richer representation of node proximity and therefore, we have reasons to believe that they could be more accurate (this claim needs, of course, further support with experimental evidence). Finally, note that L-wrappers are fully declarative, while tree wrappers combine declarative extraction paths with a procedural algorithm for grouping extracted nodes into tuples.

An interesting application of data extraction and annotation in the framework of logic programming is discussed in paper [8]. This work extends Lixto [2,3] to a platform that allows manipulation of Web data using logic-based inference.

A new wrapper induction algorithm inspired by inductive logic programming is introduced in paper [1]. The algorithm exploits traversal graphs of documents trees that are mapped to XPath expressions for data extraction. However, that paper does not define a declarative semantics of the resulting wrappers. Moreover, the wrappers discussed in [1] aim to extract only single items, and there is no discussion of how to extend the work to tuples extraction.

As concerning theoretical work, [16] 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 [15].

Finally, in [19] 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.

3 L-wrappers and their patterns

3.1 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, an L-wrapper can be regarded as a set of data extraction patterns. A pattern is 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 [13].

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 [13].*
- (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* $1 \leq i < k$.

In what follows we take a graph-based perspective in defining patterns. Within this framework, a pattern is a directed graph with labeled arcs and vertices. Arc labels denote conditions that specify the tree delimiters of the extracted information, according to the parent-child and next-sibling relationships (e.g. is there a parent node?, is there a left sibling?, etc.). Vertex labels specify conditions on nodes (e.g. is the tag label *td*?, is it the first child?, etc.). A subset of graph vertices is used for selecting the items for extraction.

Intuitively, an arc labeled 'n' denotes the "next-sibling" relation while an arc labeled $'c'$ denotes the "parent-child" relation. As concerning vertex labels, label $'f'$ denotes "first child" condition, label $'l'$ denotes "last child" condition and label $\sigma \in \Sigma$ denotes "equality with tag σ" condition.

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:* $'f'$ *requires the corresponding vertex to indicate a first child;* $'l'$ *requires the corresponding vertex to indicate a last child;* σ *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 [16].

Note also that we use the term 'node' when referring to document trees and the term 'vertex' when referring to the graph of an extraction pattern.

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* $I : V \to T$ *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* $I : V \to T$ *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 [13]. 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 $I \circ \mu : D \rightarrow T^2$*, 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.

3.2 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 Sect. 3.3).

3.2.1 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* $p_1 \simeq p_2$.

Examples of pattern subsumption and equivalence are shown in Fig. 1. Referring to that figure, one can easily note that $p_1 \preceq p_2$ and $p_3 \simeq p_4$.

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 $p' \simeq p$.

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

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 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).

Figure 2 illustrates the operation of pattern simplification. On that figure, p is simplified to p' by deleting arc (U,W), so $p \simeq p'$. Intuitively, the fact that U is a child of W follows from the facts that U is a left sibling of V and V is a child of W, so it does not need to be represented explicitly.

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.

3.2.2 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 if 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:*

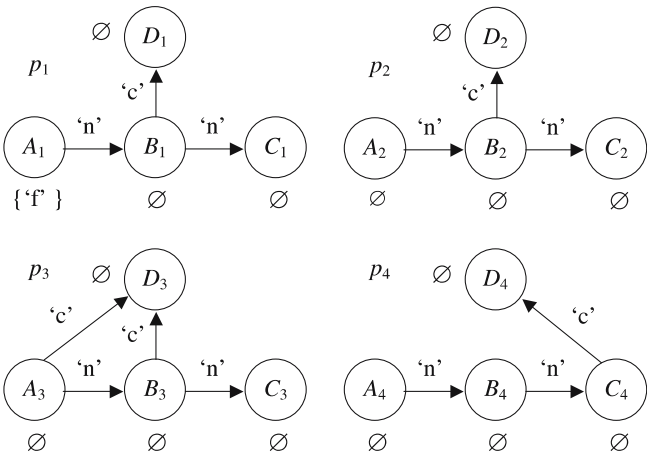


Fig. 1 Pattern subsumption and equivalence

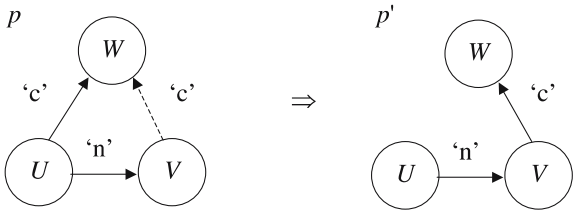


Fig. 2 Pattern simplification

(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.

Figure 3 illustrates graphically how vertex pairing is performed.

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:*

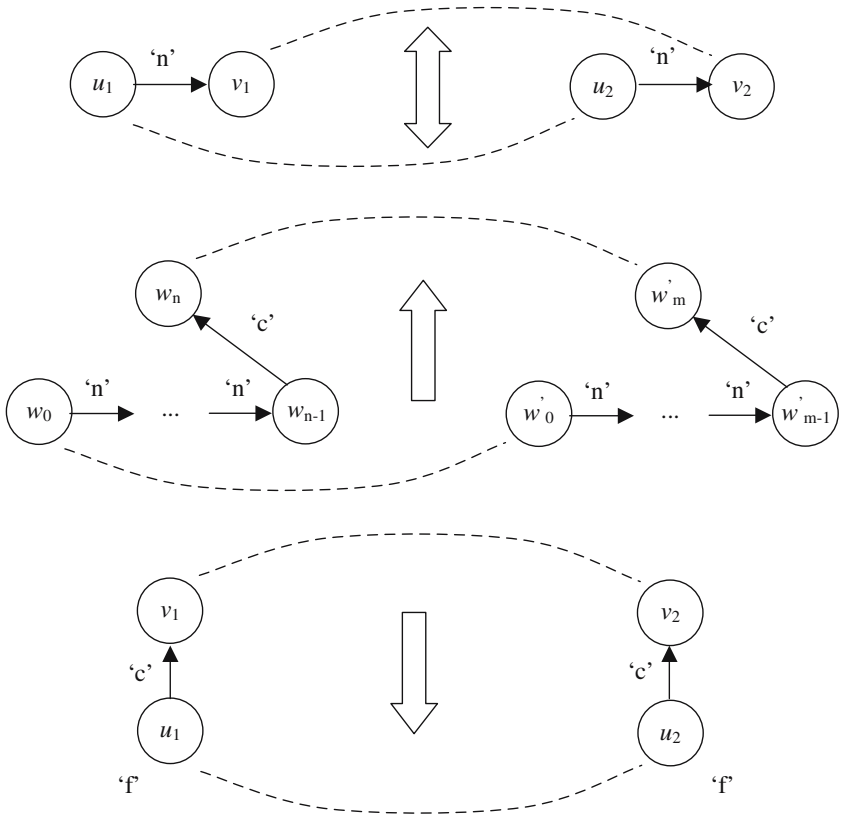
(i) $V' = V \setminus \{v\}$;
(ii) A' *is obtained by replacing each arc* $(x, v) \in A$ *with* (x, u) *and each arc* $(v, x) \in A$ *with* (u, x).

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* $\lambda_a((u, v)) = \lambda_{a_2}((u, v'))$.

Fig. 3 Vertex pairing



(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 Fig. 5).

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) $tag_\sigma \subseteq \mathcal{N}$, $(tag_\sigma(N) = true) \Leftrightarrow (\sigma$ 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.

3.3 Formal definition of L-wrappers

An *L-wrapper* can be defined formally as a set of extraction patterns that share the set of attribute names.

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.* $Ans(W, t) = \cup_{i=1}^{n} Ans(p_i, t)$.

Let us consider for example the problem of extracting printer information from Hewlett Packard's Web site. The printer information is represented in multi-section two column HTML tables (see Fig. 4). 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.

Moreover, let us consider also two single-pattern L-wrappers for this example inspired from [6,7] that were learnt using FOIL program [26]: (i) for pairs (feature class, feature name); (ii) for pairs (feature name, feature value). The wrappers are shown in Fig. 5 (*FC* = feature class, *FN* = feature name, *FV* = feature value). That figure illustrates the two patterns and also the pattern resulted from their merging. One can easily notice that these patterns are already in normal form.

4 Semi-automatic construction of L-wrappers

Manual construction of L-wrappers is a difficult and time consuming process that would require a very careful examination of target documents to identify correct extraction patterns. Fortunately it is possible to employ standard machine learning techniques, like inductive logic programming, to semi-automatic construction of L-wrappers. This problem is the focus of the current section of the paper.

4.1 Overview of the extraction process

We consider a Web data extraction scenario which assumes the manual execution of a few extraction tasks by the human user. An inductive learning engine could then use the extracted examples to learn a general extraction rule that can be further applied to the current or other similar Web pages.

We propose a generic data extraction process that is structured into the following sequence of stages:

(i) *Crawl or browse the Web and download HTML pages of interest.* This step is quite straightforward. Either a human user is browsing the Web to download interesting pages or the task of Web navigation and page download is automatized by means of a crawler component [10]. The result of this step is a collection of pages that were fetched from the Web and stored on the user machine into a local repository.

(ii) *Preprocess and convert the document from HTML to XHTML.* In this step the input HTML document is cleaned and converted to a well-formed XML document written in XHTML and structured as a tree.

(iii) *Manually extract a few examples.* In this step the user loads a few XHTML pages from the local repository and performs a few extraction tasks. The result is a set of annotated XHTML documents with special markups for the extracted items.

(iv) *Parse the annotated XHTML documents and generate a suitable representation as input for the learner.* In this step the annotated XHTML documents are parsed and converted into an appropriate input for the learning program.

(v) *Apply the learning algorithm and obtain the wrapper.* In this step the learning program is executed on the input generated in the previous step. The result is a specification of the wrapper.

(vi) *Compile the wrapper into a suitable implementation language.* In this step the wrapper specification is mapped to a declarative or procedural implementation language thus enabling the efficient execution of the wrapper.

(vii) *Execute the wrapper to extract new data.* In this step the user applies the wrapper to new XHTML pages in order to automatically extract new data.

4.2 Relational representation of document trees

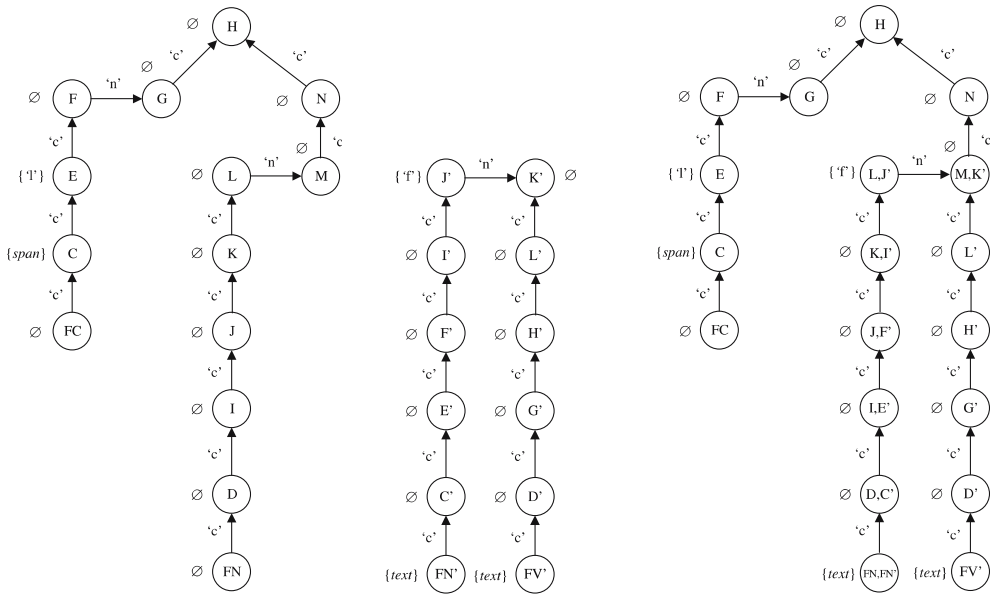
We propose the use of general-purpose inductive logic programming for semi-automatic generation of L-wrappers. In particular, in the next section we shall describe the use of FOIL program for performing this task. One prerequisite for this is to devise a relational representation of XHTML documents.

An XHTML document is composed of a structural part and a content part.

The structural part consists of the set of document nodes or elements. The document elements are nested into a tree like structure. Each document element has assigned a specific tag from a given finite set of tags Σ. There is a special tag *text* $\in \Sigma$ that designates a text element.

Fig. 4 An XHTML document fragment and its graphic view

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



$D_1 = \{Class, Name\}$
$U_1 = \{FC, FN\}$
$\mu_1(Class) = FC$
$\mu_1(Name) = FN$

Pattern p_1

$D_2 = \{Name, Value\}$
$U_2 = \{FN', FV'\}$
$\mu_2(Name) = FN'$
$\mu_2(Value) = FV'$

Pattern p_2

$D = \{Class, Name, Value\}$
$U = \{FC, (FN, FN'), FV'\}$
$\mu(Class) = FC$
$\mu(Name) = (FN.FN')$
$\mu(Value) = FV'$

p_1 merged with p_2

Fig. 5 Patterns and pattern merging

The content part of a document consists of the actual text in the text elements and the attribute-value pairs attached to the other document elements.

The structure of XHTML documents can be represented as unranked ordered trees, also known as Σ-trees. The set $\mathcal{T}_\Sigma$ of Σ-trees is defined inductively as follows:

(i) if $\sigma \in \Sigma$ then $\sigma \in \mathcal{T}_\Sigma$;
(ii) if $\sigma \in \Sigma$ and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$, $n \geq 1$ then $\sigma(t_1, \ldots, t_n) \in \mathcal{T}_\Sigma$.

Note that there is no a priori bound on the number of children of a node in a Σ-tree, i.e. the tree is *unranked*, and note also that the set of children of a given node in a Σ-tree is ordered, i.e. the tree is *ordered*.

Recall that our task in this section was to devise a relational representation for an XHTML document tree. We assign a unique identifier (an integer value) to each node of the tree. Let $\mathcal{N}$ be the set of all node identifiers. In what follows a node is identified by its unique identifier.

The structural component of an XHTML document tree can be represented using the following five relations:

(i) $child \subseteq \mathcal{N} \times \mathcal{N}$ defined as:
($child(P, C) = true$) $\Leftrightarrow$ (P is the parent of C).
(ii) $next \subseteq \mathcal{N} \times \mathcal{N}$ defined as:
($next(L, N) = true$) $\Leftrightarrow$ (L is the left sibling of N).
(iii) $tag_\sigma \subseteq \mathcal{N}, \sigma \in \Sigma$ defined as:
($tag_\sigma(N) = true$) $\Leftrightarrow$ the tag of node N) is σ.
(iv) $first \subseteq \mathcal{N}$ defined as:
($first(X) = true$) $\Leftrightarrow$ (X is the first child of its parent node).
(v) $last \subseteq \mathcal{N}$ defined as:
($last(X) = true$) $\Leftrightarrow$ (X is the last child of its parent node).

Consider the Hewlett Packard's site of electronic products[3] and the task of IE from a product information sheet for printers. The printer information is displayed in a two-column table as a set of feature-value pairs. Our task is to extract the names and/or the values of the printer features. This information is stored in the leaf elements of the page. Figure 6 displays in the left panel the XHTML tree of a fragment of this document and in the right panel the graphical view of this fragment as a two-column table. Figure 7 displays this XHTML document fragment as a Σ-tree.

The right panel of Fig. 7 displays a part of the relational representation of the XHTML document tree shown on the left panel.

The relational representation of document trees introduced in this section makes convenient the definition of first-order data extraction rules. A rule for extracting a data item with k attributes defines a relation $extract \subseteq \mathcal{N}^k$ such that ($extract(N_1, \ldots, N_k) = true$) $\Leftrightarrow$ (($N_1, \ldots, N_k$) is an extracted tuple of k nodes).

The definition of a data extraction rule consists of a set of clauses. The head of a clause is $extract(N_1, \ldots, N_k)$ and the body of a clause is a conjunction of positive and negative literals made of the relations *child*, *next*, and tag_σ. Following the observation that an extraction pattern can be mapped to a conjunctive query over the signature $(child, next, first, last, (tag_\sigma)_{\sigma \in \Sigma})$, it is not difficult to see that a data extraction rule corresponds to an extraction pattern and each argument of relation *extract* corresponds to an extraction vertex of this pattern.

For example, assuming that we want to extract all the text nodes of the XHTML document from Fig. 4 that have a grand–grand–parent of type *table* that has a parent that has a right sibling, we can use the following extraction rule:

$$\begin{aligned} &extract(A) \leftarrow \\ &\quad text(A) \wedge child(B, A) \wedge child(C, B) \wedge \\ &\quad child(D, C) \wedge table(D) \wedge child(E, D) \wedge \\ &\quad next(E, F) \end{aligned}$$

Manual writing of data extraction rules is a slow and difficult process requiring a careful analysis of the structure of the target XHTML document. We propose to learn these rules with a general purpose relational learning program.

4.3 Using FOIL to learn extraction rules

First order inductive learner (FOIL) is a general purpose first order (also known as relational) inductive learning program developed by John Ross Quinlan and his team at the beginning of the 1990s [26]. Even if it is not in the scope of the paper to give a detailed description of how FOIL works, we decided to give a brief overview of first order inductive learning in order to make the paper self contained. For more details see [26] and [24] (Chapter 10).

In first order inductive learning the training data comprises the following two parts:

(i) The *target relation*, i.e. the relation that is learnt. It is defined extensionally as a set of tuples. Usually the tuples are partitioned into a set of positive tuples and a set of negative tuples.
(ii) The *background relations*, usually defined extensionally as sets of tuples.

The goal of first order inductive learning is to construct a logic program that represents an appropriate intensional definition of the target relation in terms of the background relations and, optionally itself, if recursive definitions are allowed. The learnt definition must cover all the positive tuples (or a large fraction of them) and no negative tuples (or a small fraction of them) from its extensional definition.

The FOIL requires as input the information about the target and the background relations, as mentioned above, and the definition of the relations arguments types, as sets of constants. FOIL's output is a set of clauses that represents a disjunctive definition of the target relation. The set of clauses is constructed iteratively, one clause at a time, removing all the covered positive tuples until an empty set remains. Every clause is constructed using an heuristic hill climbing search strategy that is guided by the information gain, determinate literals and the clause complexity (see [26] for details).

[3] http://www.hp.com

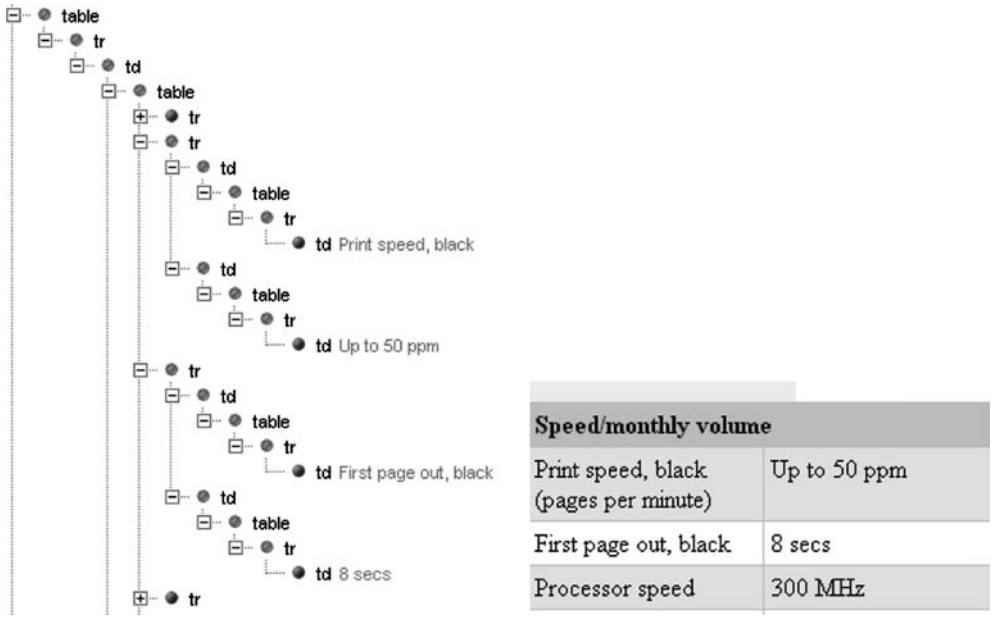


Fig. 6 An XHTML document fragment and its graphic view

Fig. 7 The structure of the XHTML fragment from Fig. 4 represented as a Σ-tree and a part of its relational representation

$$
\begin{array}{l}
html(\ \ldots \\
\quad table(\\
\quad\quad tr(\ldots), \\
\quad\quad tr(\\
\quad\quad\quad td(table(tr(td(text)))), \\
\quad\quad\quad td(table(tr(td(text)))), \\
\quad\quad), \\
\quad\quad tr(\\
\quad\quad\quad td(table(tr(td(text)))), \\
\quad\quad\quad td(table(tr(td(text)))), \\
\quad\quad), \\
\quad\quad tr(\ldots), \\
\quad\quad \ldots \\
\quad) \\
\quad \ldots \\
)
\end{array}
$$

$$
\begin{array}{l}
tag(0.html) \\
tag(100, table) \\
tag(101, tr) \\
tag(102, tr) \\
tag(103, tr) \\
tag(107, td) \\
tag(108, table) \\
tag(109, tr) \\
tag(110, td) \\
tag(111, text)
\end{array}
$$

$$
\begin{array}{l}
child(100, 101) \\
child(100, 101) \\
child(100, 102) \\
child(100, 103) \\
child(101, 107) \\
child(107, 108) \\
child(108, 109) \\
child(109, 110) \\
child(110, 111)
\end{array}
$$

$$
\begin{array}{l}
next(101, 102) \\
next(102, 103) \\
next(103, 104)
\end{array}
$$

An important step is to prepare the input for FOIL. In what follows we assume that during the training stage the user selects one or more XHTML documents, and performs a few extraction tasks on them. These documents are assembled into a single XML document – the training document, by concatenating them under a new root element. The problem is to map these training data into input for FOIL.

First, we must define the relations arguments types. We used a single type that represents the set $\mathcal{N}$ of all the nodes found in the training document. Second, we must define the target relation *extract*. We used only the nodes extracted during the training stage from the training document, as the set of positive examples. FOIL uses the closed world assumption (CWA) by considering all the other nodes in the training document as negative examples. Third, we must define the background relations *next* and tag_σ.

Additionally, FOIL can be parameterized from the command line. All the parameters have default values, but we found useful to control explicitly the following options:

(i) The use or not of negative literals in the body of a clause.
(ii) The maximum variable depth in literals. Variables occurring in the clause head have depth 0 and a new variable in the literal has depth one greater than the maximum depth of its existing variables.
(iii) And finally, to set the minimum clause accuracy to an appropriate value. The clause accuracy represents the percentage of the positive tuples from the set of all tuples covered by the relation. FOIL will not accept any clause with an accuracy lower than this value.

We consider a FOIL example taken from our experiments. The input is split into the following sections: argument types (*N*), target relation (*extract*, only positive

examples), and background relations (*child*, *next*, *table*, ..., *span*). In this example we assume the extraction of single items (i.e. tuples of arity $k = 1$).

```
#N: 0,1,2,3,4, ..., 1096,1097.

extract(N)
1004
1006
...
1056
1058
.
*child(N,N)
0,1
0,2
1,3
1,4
...
1094,1097
.
*next(N,N)
1,2
3,4
...
1091,1092
.
*table(N)
26
27
...
991
.
...
*span(N)
250
251
...
980
.
```

For example, a FOIL command-line that we used in our experiments, is:

```
foil -d8 -a40 -v3 <ex10.d >ex10.o
```

The parameter `d8` is used for setting the maximum variable depth to 8. The parameter `a40` is used for setting the minimum clause accuracy to 40%. The parameter `v3` is used to set the level of verbosity of FOIL's output to 3 (minimum is 0 and maximum is 4), i.e. the output will include quite detailed information about the FOIL's search process. The parameters `ex10.d` and `ex10.o` are the input and output files.

4.4 Experiments and results

4.4.1 Software tools

We have developed a set of prototype software tools to support our experiments with learning L-wrappers. The tool set implements the scenario outlined at the beginning of this section.The tool set incorporates the following on-the-shelf software components:

(i) The machine learning program FOIL. This component is called from the main program through a command line.
(ii) The Xerces[4] library for XML processing. This component is used to read an XHTML document into a DOM tree. The DOM tree is then mapped onto the format required as input for FOIL using a breadth-first traversal.
(iii) The Tidy[5] library for HTML document cleaning and pre-processing. This component is used in the pre-processing stage of HTML documents for converting them to XHTML. We have also developed a custom XSLT stylesheet to remove redundant HTML elements like *script* and *style*.

4.4.2 Single-item extraction using CWA

We ran a series of experiments on the Hewlett Packard's Web site. The task was to extract the printer's feature values from their information sheets.

We selected an experimental set of 30 documents representing information sheets for HP printers. The printer information is represented in two column HTML tables (see Fig. 4). The feature values are stored in the second column. Each of the selected documents contains between 28 and 38 features.

We used training examples from a single document representing the information sheet of the HP Business Inkjet 2600 (C8109A) printer. This document contains 28 positive examples and a total of 1,098 nodes. We performed learning experiments with 1, 2, 3, 4, 12, 24 and 28 positive examples selected from this set. The results of the learning stage were applied to all of the 30 documents, measuring the precision and recall indicators. These values were computed by averaging the precision and recall for each individual document.

[4] http://xml.apache.org/xerces2-j/index.html.

[5] http://www.w3c.org/People/Ragget/tidy/.

The experiments were divided in two classes:

(i) In the first class we have ignored the *next* relation as a background relation of FOIL's input.
(ii) In the second class we have added the *next* relation to the set of background relations of FOIL's input.

For each class of experiments we have varied explicitly the following parameters: the use or not of negated literals in rule bodies (FOIL parameter), the minimum clause accuracy (FOIL parameter), and the number of positive examples. We also set the maximum variable depth to 8 and used the closed world assumption to let FOIL generate the set of negative examples in all the experiments.

The results of the experiments in the first and second class are summarized in Tables 1, 2 respectively.

This example shows the rule learnt for our printer data case in Experiment 16 from Table 2.

$$extract(A) \leftarrow child(B,A) \wedge text(A) \wedge td(B) \wedge \neg next(A,\_1) \wedge \neg next(\_1,A) \wedge child(C,B) \wedge child(D,C) \wedge \neg next(\_1,C) \wedge child(E,D) \wedge child(F,E) \wedge \neg next(F,\_1) \wedge \neg next(E,\_1)$$

Note that if X is an unbound variable and Y is a bound variable then $\neg next(X,Y) = true$ means that Y is instantiated with the first child node of its parent node. Therefore we can replace $\neg next(X,Y)$ with $first(Y)$. Similarly, if X is a bound variable and Y is an unbound variable then $\neg next(X,Y) = true$ means that Y is instantiated with the last child node of its parent node. Therefore we can replace $\neg next(X,Y)$ with $last(X)$. According to these observations, the rule shown before can be rewritten without negations:

$$extract(A) \leftarrow child(B,A) \wedge text(A) \wedge td(B) \wedge last(A) \wedge first(A) \wedge child(C,B) \wedge child(D,C) \wedge first(C) \wedge child(E,D) \wedge child(F,E) \wedge last(F) \wedge last(E)$$

After a careful analysis of the results of our experiments, the following conclusions were drawn:

(i) In all the experiments with the precision lower than 0.500 the learnt rule also extracted the feature name of every extracted feature value. This explains the low values of the precision in these experiments.
(ii) The low precision values in the experiments of the first class clearly shows the importance of exploiting the information conveyed by the *next* relation for data extraction from tree structured documents.
(iii) The use of negated literals in rule bodies improves significantly the precision of the rules in the case of using the *next* relation. But note that in all the experiments the use of ¬ could be replaced with predicates *last* and *first*.
(iv) The learnt rules were able to extract all the feature values (i.r. recall was 1) even when for training was used a single positive example. This indicates a minimum user effort to manually extract a single item from the training document.
(v) In the case when we used for training as positive examples all the fields that must be extracted from the training document and we allowed the presence of negated literals in rule bodies, the precision and recall were 1.

Table 1 Experiments without the use of the background relation *next*

Experiment no.	Neg. lit.	No. of pos. examples	Clause acc.(%)	Precision	Recall
1	No	12	10	0.334	0.980
2	Yes	12	0	0.355	1
3	No	24	20	0.334	0.980
4	Yes	24	0	0.409	0.970
5	No	28	20, 30	0.334	0.980
6	Yes	28	0, 20	0.409	0.970

Table 2 Experiments with the use of the background relation *next*

Experiment no.	Neg. lit.	No. of pos. examples	Clause acc. (%)	Precision	Recall
1	No	1	0	0.332	1
2	Yes	1	0	0.411	1
3	No	2	0	0.730	0.800
4	Yes	2	0	0.440	1
5	No	3	0, 5	0.698	1
6	Yes	3	0	0.440	1
7	No	4	0	0.332	1
8	No	4	10	0.730	0.800
9	Yes	4	0	0.448	1
10	No	12	20, 40	0.855	0.800
11	Yes	12	20	1	0.800
12	No	24	20	0.698	1
13	No	24	60	0.855	0.800
14	Yes	24	80 (default)	1	0.800
15	No	28	80 (default)	0.855	0.800
16	Yes	28	80 (default)	1	1

4.4.3 Explicit enumeration of negative examples

The use of CWA has the advantage that the negative examples do not have to be explicitly enumerated. But this has also a serious drawback: if only a part of the relevant items are annotated and used as positive examples (i.e. the user has manually extracted some, but not all the useful items) then, by applying CWA, the set of negative examples will also include the rest of the items to be further extracted.

One approach to avoid this is to let the user explicitly indicate one or more fragments of the training document where the relevant information is located and then

explicitly generate the negative examples by excluding all the items inside these fragments. For example, for the HP printer information sheets, the relevant information is located in the table with the list of printer features.

We performed experiments with this approach and compared the results with the CWA approach, using the same training document and the same sets of items manually annotated. In all the experiments with 6 or more positive examples (allowing the use of negated literals), the values of precision and recall were 1.

4.4.4 Using first and last

Analyzing the wrappers learnt in the previous experiments we noticed that all the occurrences of $\neg next$ could be replaced with *last* and *first*. Taking also into account that the computation of the best literal during the top-down learning process is very expensive, we thought to improve a bit the learning time by combining *first* and *last* in the document representation with no negated literals in the learnt clauses. The results are reported in Table 3.

4.4.5 Tuples extraction

Learning rules for tuple extraction proceeds similarly with the single item case. The difference is that the *extract* relation is now described as a list of positive and negative tuples rather than a list of positive and negative items.

We performed experiments of learning L-wrappers for extracting printer information from Hewlett Packard's Web site. The information is represented in two column HTML tables (see Fig. 4). Each row contains a pair (feature name, feature value). Consecutive rows represent related features and are grouped in 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.

In what follows we consider two experiments: (i) tuples extraction from flat information resources, exemplifying with pairs (feature name, feature value); (ii) coping with hierarchical information by extracting pairs (feature class, feature name). We have used the same test data as before, only the learning tasks were changed. We used examples from a single training document with 28 tuples.

Tuples extraction from flat information resources We performed an experiment of learning to extract the tuples containing the feature name and feature value from HP printer information sheets. The experimental results are reported in Table 4, row 1.

Because the number of tuples exceeds the size of FOIL's default tuple space, we explicitly set this size to a higher value (300,000) and we used a random sample representing a fraction of the negative examples in the learning process.

The L-wrapper learnt consisted of a single clause (*FN* = feature name, *FV* = feature value):

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

This rule extracts all the pairs of text nodes such that the grand–grand–grand–grandparent of the first node (J) is the first child of its parent node and the left sibling of the grand–grand–grand–grandparent of the second node (K).

Tuples extraction from hierarchical information resources The idea of modeling nested documents using a relational approach and building wrappers according to this model is not entirely new; see [18].

In this section we present an experiment of learning to extract pairs (feature class, feature name) from printer information sheets. Note that because we may have many features in the same class, the information is hierarchically structured. The experimental results are reported in Table 4, row 2.

The L-wrapper learnt consisted of a single clause (*FC* = feature class, *FN* = feature name):

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

There is one difference from the flat case – how examples are collected. In this case, some examples will share the feature class. Moreover, in the general case,

Table 3 Experiment with *first* and *last* and no negated literals

Experiment no.	No. pos. ex	Cl. acc. (%)	Prec.	Rec.
1	1	0	0.436	1
2	3	0	0.448	1
3	4	0 , 10	1	1
4	12	20 , 40	1	0.800
5	24	20 , 60	1	1
6	28	80	1	1

Table 4 Experiments with tuples extraction

Experiment no.	No. pos. ex	Frac. neg. ex (%)	Prec.	Rec.	No.lit.
1	24, 28	20	0.959	1	14
2	24, 28	20	1	1	15

some fields will need to be selected repeatedly during the manual extraction process (like the feature class in the printer example). This can be avoided by designing the graphical user interface that guides the example selection such that the tuple previously selected is always saved and thus its fields may be reused in the next selection.

Discussion In our experiments we used CWA to generate the negative examples. This means that each tuple not given as positive example, automatically counts as negative example. Let d be the size of the training document (i.e. the number of nodes) and let k be the tuple arity. The number of negative examples is proportional with d^k, i.e. it is exponential in the tuple arity. For example, our documents had about 1,000 nodes. This means that for tuples of arity 3, the total number of negative examples is about 10^9.

For this reason we had problems with learning to extract tuples of arity greater than 2. Because the number of negative examples exceeded the memory available, we were forced to use for learning a random sample representing a very small fraction (less than 0.1%) of the total number of the negative examples. This had the effect of producing wrappers with a very low precision.

Finally, note that the two wrappers presented here for extracting pairs (feature class, feature name) and (feature name, feature value) correspond to patterns p_2 and p_1 from Fig. 5. However, applying the pattern merging operator, it is possible to obtain an L-wrapper for extracting triples (feature class, feature name, feature value). Generalizing, it follows that in order to learn a wrapper to extract tuples of arity $k \geq 3$, we can learn $k-1$ 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.

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.

5 Mapping L-wrappers to XSLT

In this section we are focusing on the last two stages of the data extraction process: wrapping compilation and wrapper execution. We have chosen to express L-wrappers into XSLT – a standard language for the transformation of XML documents [12].

5.1 Mapping algorithm

Paper [9] describes a subset of XSLT, called XSLT_0, that has a Plotkin-style formal semantics. The reader is invited to consult reference [9], for details on XSLT_0, its pseudocode notation and the formal semantics.

Let us consider an L-wrapper $W = \{p_1, p_2, \ldots, p_n\}$. Each pattern p_i is mapped to an XSLT stylesheet that is expressed in XSLT_0, so, in what follows, we shall focus only on the mapping of a single pattern to a stylesheet.

Let us consider a pattern $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ and let $L \subseteq V$ be the leaves of its graph (i.e. vertices with in-degree 0). The idea is to generate XSLT_0 templates for all the leaf and extraction vertices (i.e. elements of $L \cup U$) of the L-wrapper, moving upwards and downwards in the graph and generating appropriate XPath expressions [30]. The extracted information is passed between templates by means of template variables and parameters. Here is an informal description of this algorithm:

Step 1 Start from the document root and generate the start template, by moving downwards to one of the vertices in $L \cup U$.

Step 2 Move from the current vertex (say w_0) to another vertex in $L \cup U$ (say w_1). The path taken depends on the type of the first vertex: if $w_0 \in L$ then we move first upwards, to the common ascendent of w_0 and w_1 and then downwards to w_1; if $w_0 \notin L$ then we follow the direct descendent path to w_1.

Step 3 Generate a template that will select the content of w_0 in case $w_0 \in U$.

Step 4 Repeat Steps 2 and 3 until there are no more unvisited vertices in $L \cup U$.

Step 5 Generate the final template, which will display the extracted tuples.

The XSLT_0 description of the single-pattern wrapper resulted from merging patterns p_1 and p_2 from Figure 5 is shown in Table 5.[6] 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()
```

An informal argument why this mapping works correctly is in place. Referring to Fig. 5, 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 themselves, using the *content*(.) expression.

[6] Note that our version of XSLT_0 is slightly different from the one presented in [9].

Table 5 Description of the sample wrapper in $XSLT_0$ pseudocode

```
template start(html)
    return
        result(selclass(xp1))
end

template selclass(*)
    vardef
        varClass := content(.)
    return
        selname(xp2,varClass)
end

template selname(*,varClass)
    vardef
        varName := content(.)
    return
        display(xp3,varClass,varName)
end

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

The extracted information is passed between templates in template variables and parameters *varClass* and *varName*.

5.2 An example: mining travel resources

E-tourism is a leading area in e-commerce, with an increasing number of travel agencies offering their services through online transaction brokers [21]. They provide to human users information in areas like hotels, flights, trains or restaurants, in order to help them to plan their business or holiday trips. The travel information is heterogeneous and distributed, and there is a need to gather, search, integrate and filter it efficiently [17].

We now demonstrate the use of L-wrappers to extract travel information from the Travelocity Web site[7] (see Fig. 8). That Web page displays hotel information comprising the hotel name, address and description, the check-in and check-out dates, the types of rooms offered and the corresponding average nightly rate. Adopting the relational model, we associate to this resource the following set of attribute names related to hotels: {*name*, *address*, *description*, *period*, *roomtype*, *price*}.

Because of the relatively large number of attributes, we used the pattern merging approach. The following pairs of attributes were chosen: {*name*, *address*}, {*address*, *description*}, {*name*, *period*}, {*period*, *roomtype*}, and {*roomtype*, *price*}.

Then we generated extraction rules for each pair of attributes by using the FOIL program. The following 5 rules were generated: ($NA = name$, $AD = address$, $DE = description$, $PE = period$, $RO = roomtype$, and $PR = price$):

$$
\begin{aligned}
&extract(NA, AD) \leftarrow first(AD) \wedge td(AD) \wedge child(C, NA) \wedge \\
&\quad child(D, AD) \wedge next(D, E) \wedge child(F, E) \wedge span(C) \wedge \\
&\quad first(D) \wedge child(G, C) \wedge child(H, G) \wedge next(I, H) \wedge child(J, I) \wedge \\
&\quad child(K, F) \wedge child(L, K) \wedge child(M, J) \wedge child(N, M) \wedge \\
&\quad child(O, L) \wedge child(P, O) \wedge child(Q, P) \wedge child(N, Q). \\
&extract(AD, DE) \leftarrow child(C, AD) \wedge child(D, DE) \wedge next(AD, E) \wedge \\
&\quad next(C, F) \wedge child(G, F) \wedge child(F, D) \wedge first(G) \wedge text(DE). \\
&extract(NA, PE) \leftarrow text(NA) \wedge child(C, NA) \wedge child(D, PE) \wedge \\
&\quad next(E, D) \wedge child(F, E) \wedge b(D) \wedge child(G, C) \wedge child(H, G) \wedge \\
&\quad next(I, H) \wedge child(J, I) \wedge child(K, J) \wedge next(K, L) \wedge \\
&\quad next(L, M) \wedge child(M, N) \wedge child(O, F) \wedge \\
&\quad child(P, O) \wedge child(N, P). \\
&extract(PE, RO) \leftarrow child(C, PE) \wedge child(D, RO) \wedge next(D, E) \wedge \\
&\quad next(F, C) \wedge child(G, E) \wedge child(H, F) \wedge next(I, G) \wedge \\
&\quad child(J, I) \wedge next(K, J) \wedge first(D) \wedge child(K, L) \wedge child(L, H). \\
&extract(RO, PR) \leftarrow child(C, RO) \wedge child(D, PR) \wedge next(C, E) \wedge \\
&\quad next(D, F) \wedge child(G, E) \wedge next(H, G) \wedge child(I, H) \wedge \\
&\quad next(J, I) \wedge child(G, D) \wedge first(C) \wedge last(F) \wedge text(PR).
\end{aligned}
$$

Next we merged these patterns into a single-pattern L-wrapper and then we translated it into $XSLT_0$ using the algorithm outlined before. A set of seven $XSLT_0$ templates was obtained (see Table 6).

XPath expressions xp_1, xp_2, xp_3, xp_4, xp_5 and xp_6 of the wrapper from Table 6 are defined as follows:

```
xp1 = //*/*/*/*/following-sibling::*[1]/span/text()
xp2 = parent::*/parent::*/parent::*/preceding-sibling::*[1]/
      parent::*/parent::*/following-sibling::*[2]/*/*/*/*/*/
      following-sibling::*[1][local-name()='b']/*
xp3 = parent::*/preceding-sibling::*[1]/parent::*/parent::*/
      parent::*/parent::*/parent::*/preceding-sibling::*[2]/
      parent::*/*/*/*/*/*/*[position()=1]/*/*/text()
xp4 = parent::*/parent::*/preceding-sibling::*[1]
      [position()=1]/*[position()=1 and local-name()='td'
      and following-sibling::*[1]]
xp5 = parent::*/following-sibling::*[1]/parent::*/parent::*/
      parent::*/parent::*/parent::*/parent::*/parent::*/*/
      following-sibling::*[2]/*/*/following-sibling::*[1]/
      */following-sibling::*[1]/*[position()=1 and
      following-sibling::*[1]]/*
xp6 = parent::*/following-sibling::*[1]/parent::*/
      *[position()=last()-1]/text()
```

For wrapper execution we can use any of the available XSLT transformation engines. In our experiments we have used Oxygen XML editor (see Fig. 9), a tool that incorporates some of these engines. The experimental results confirmed the efficacy of the approach: values 0.87 and 1 were recorded for precision and recall measures.

6 Conclusions and future work

In this paper we studied a new class of wrappers for data extraction from semi-structured sources inspired by logic programming – L-wrappers. This study is supplemented with a description of some guidelines and experimental results in the area of semi-automatic construction of L-wrappers. In particular, we described how to apply first-order inductive learning to generate L-wrappers and how to map the resulting extraction rules to XSLT for efficient data extraction from Web sources. The results of this work provide also some theoretical insight into L-wrappers and their patterns, thus enabling the link of our work with related works in this field. As future work we plan to implement these ideas into an information extraction tool and also to give a formal proof of the correctness of the mapping of L-wrappers to XSLT.

[7] http://www.travelocity.com

Fig. 8 An XHTML document fragment and its graphic view

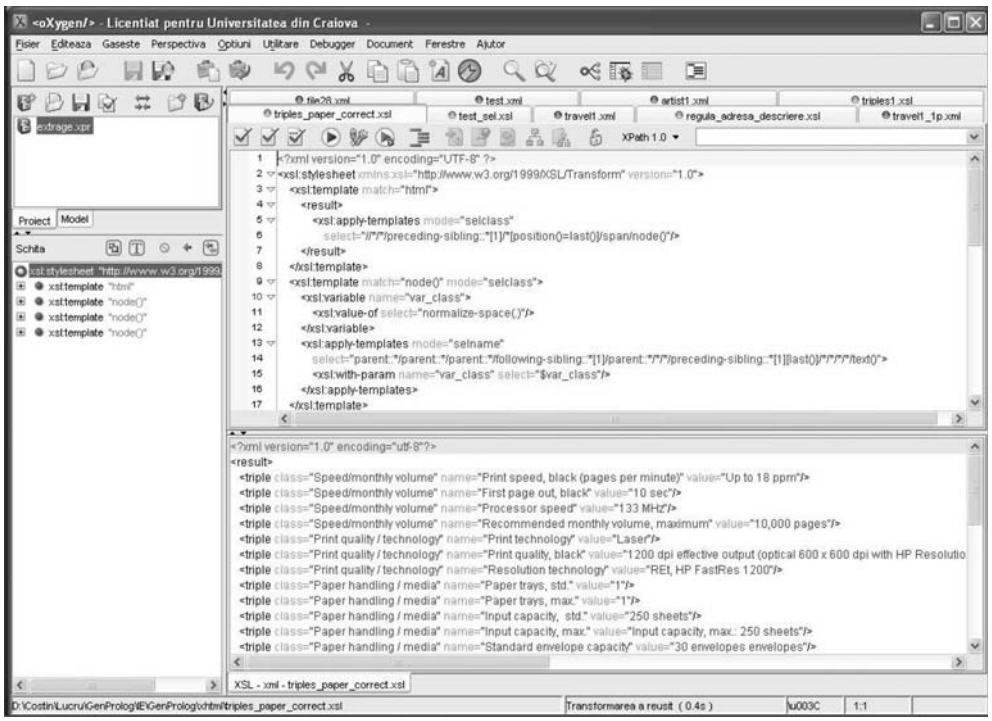


Fig. 9 Wrapper execution inside Oxygen XML editor

Appendix

AXSLT 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">
```

Table 6 Description of the sample wrapper in $XSLT_0$ pseudocode

```
template start(html)
    return
        result(selname(xp1))
end

template selname(*)
    vardef
        varName := content(.)
    return
        selperiod(xp2,varName)
end

template selperiod(*,varName)
    vardef
        varPeriod := content(.)
    return
        seldescription(xp3,varName,varPeriod)
end

template seldescription(*,varName,varPeriod)
    vardef
        varDescription := content(.)
    return
        seladdress(xp4,varName,varPeriod,
            varDescription)
end

template seladdress(*,varName,varPeriod,
    varDescription)
    vardef
        varAddress := content(.)
    return
        selroomtype(xp5,varName,varPeriod,
            varDescription,varAddress)
end

template selroomtype(*,varName,varPeriod,
    varDescription,varAddress)
    vardef
        varRoomtype := content(.)
    return
        display(xp6,varName,varPeriod,
            varDescription,varAddress,varRoomtype)
end

template display(*,varName,varPeriod,varDescription,varAddress,varRoomtype)
    vardef
        varPrice := content(.)
    return
        tuple[name → varName; period → varPeriod; description → varDescription;
              address → varAddress; roomtype → varRoomtype; price → varPrice]
end
```

```
      <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>
```

References

1. Anton T (2005) XPath-wrapper induction by generalizing tree traversal patterns. In: Mathias Bauer, Boris Brandherm, Johannes Fürnkranz, Gunter Grieser, Andreas Hotho, Andreas Jedlitschka, Alexander Krner (eds) Lernen, Wissensentdeckung und Adaptivitt (LWA) 2005, GI Workshops, Saarbrcken, pp 126–133
2. Baumgartner R, Flesca S, Gottlob G (2001) The Elog web extraction language. In: Nieuwenhuis R, Voronkov A (eds) Proceedings of LPAR'2001, LNAI 2250, Springer, Berlin Heidelberg New York, pp 548–560
3. Baumgartner R, Frolich O, Gottlob G, Harz P, Herzog M, Lehmann P (2005) Web data extraction for business intelligence: the Lixto approach. In: Gottfried Vossen, Frank Leymann, Peter C. Lockemann, Wolffried Stucky (eds) Datenbanksysteme in Business, Technologie und Web, 11. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Karslrhue, Germany, 2005. Lecture Notes in Informatics, vol 65, GI, pp 30–47
4. Bădică C, Bădică A (2004) Rule learning for feature values extraction from HTML product information sheets. In: Boley H, Antoniou G (eds) Proceedings RuleML'04, Hiroshima. LNCS, 3323 Springer, Berlin Heidelberg New York pp 37–48
5. Bădică C, Popescu E, Bădică A (2005a) Learning logic wrappers for information extraction from the Web. In: Papazoglou M, Yamazaki, K (eds) Proceedings of the SAINT'2005 Workshops. Computer Intelligence for Exabyte Scale Data Explosion. IEEE Computer Society Press, Trento pp 336–339
6. Bădică C, Bădică A, Popescu E (2005b) Tuples extraction from HTML using logic wrappers and inductive logic programming. In: Szczepaniak, PS, Kacprzyk J, Niewiadomski A (eds) Proceedings of the AWIC'05, Lodz, Poland. LNAI 3528 Springer, Berlin Heidelberg New York pp 44–50
7. Bădică C, Bădică A (2005) Logic wrappers and XSLT transformations for tuples extraction from HTML. In: Bressan S, Ceri S, Hunt E, Ives ZG, Bellahsene Z, Rys M, Unland R, (eds) Proceedings, 3rd international XML database symposium XSym'05, Trondheim LNCS 3671, Springer, Berlin Heidelberg New York pp 177–191
8. Bernardoni C, Fiumara G, Marchi M, Provetti A (2006) Declarative Web data extraction and annotation. 20th workshop on logic programming, WLP. Vienna, Austria
9. Bex GJ, Maneth S, Neven F (2002) A formal model for an expressive fragment of XSLT. Inf syst Elsevier 27: 21–39

10. Chakrabarti S (2003) Mining the Web. Discovering knowledge from hypertext data. Morgan Kaufmann Publishers
11. Chidlovskii B (2003) Information extraction from Tree documents by learning subtree delimiters. Proceedings of the IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03), Acapulco, Mexico pp 3–8
12. Clark J (1999) XSLT transformation (XSLT) version 1.0. W3C recommendation, 16 November 1999, http://www.w3.org/TR/xslt2
13. Cormen TH, Leiserson CE, Rivest RR (1990) Introduction to Algorithms. MIT Press, Cambridge
14. Freitag D (1998) Information extraction from HTML: application of a general machine learning approach. In: Proceedings of AAAI'98, pp 517–523
15. Gottlob G, Koch C, Schulz KU (2004) Conjunctive queries over trees. In: Proceedings of the PODS'2004, Paris, France. ACM Press, pp 189–200
16. Gottlob G, Koch C (2004) Monadic datalog and the expressive power of languages for Web information extraction. J ACM 51 (1):74–113
17. Knoblock C (2002) Agents for gathering, integrating, and monitoring information for travel planning. In: Intelligent systems for tourism. IEEE Intell Syst Nov./Dec.:53–66
18. Kushmerick N (2000) Wrapper induction: efficiency and expressiveness. Artif intell, Elsevier 118:15–68
19. Laender AHF, Ribeiro-Neto B, Silva AS, Teixeira, JS (2002) A brief survey of Web data extraction tools. In: SIGMOD record, ACM Press, 31(2): 84–93
20. Laender AHF, Ribeiro-Neto B, Silva AS (2002b) DEByE – data extraction by example. Data Knowl Eng 40 (2):121–154
21. Laudon KC, Traver CG (2004) E-commerce business technology society (2nd edn.). Pearson Addison-Wesley, location
22. Lenhert W, Sundheim B (1991) A performance evaluation of text-analysis technologies. AI Mag 12(3):81–94
23. Liu B, Grossman R, Zhai Y(2004) Mining web pages for data records. IEEE Intell Syst Nov./Dec.:49–55
24. Mitchell TM (1997) Machine learning, McGraw-Hill, location
25. Oxygen XML Editor. `http://www.oxygenxml.com/`
26. Quinlan JR, Cameron-Jones RM (1995) Induction of logic programs: FOIL and related systems. New Generation Comput 13:287–312
27. Sakamoto H, Arimura H, Arikawa S (2002) Knowledge discovery from semistructured texts. In: Arikawa S, Shinohara A (eds) Progress in discovery science. Lecture Notes in Computer Science, 2281, Springer, Berlin Heidelberg New York pp 586–599
28. Thomas B (2000) Token-templates and logic programs for intelligent web search. Intelligent Information Systems. Special Issue: Methodologies Intell Inf Syst 14(2/3):241–261
29. Xiao L, Wissmann D, Brown M, Jablonski S (2001) Information extraction from HTML: combining XML and standard techniques IE from the Web. In: Monostori L, Vancza J, Ali M (eds) Proceedings of IEA/AIE 2001. Lecture Notes in Artificial Intelligence, 2070, Springer, Berlin Heidelberg New York 165–174
30. XML Path Language (XPath) Version 1.0 http://www.w3.2.org/TR/xslt2