Intelligence Integration in Distributed Knowledge Management

Dariusz Król
Wroclaw University of Technology, Poland

Ngoc Thanh Nguyen
Wroclaw University of Technology, Poland
Chapter II

Using Logic Programming and XML Technologies for Data Extraction from Web Pages

Amelia Bădică
University of Craiova, Romania

Costin Bădică
University of Craiova, Romania

Elvira Popescu
University of Craiova, Romania

ABSTRACT

The Web is designed as a major information provider for the human consumer. However, information published on the Web is difficult to understand and reuse by a machine. In this chapter, we show how well established intelligent techniques based on logic programming and inductive learning combined with more recent XML technologies might help to improve the efficiency of the task of data extraction from Web pages. Our work can be seen as a necessary step of the more general problem of Web data management and integration.

INTRODUCTION

The Web is extensively used for information dissemination to humans and businesses. For this purpose, Web technologies are used to convert data from internal formats, usually specific to database management systems, to suitable presentations for attracting human users. However, the interest has rapidly shifted to make that information available for machine consumption by realizing that Web data can be reused for various problem solving purposes, including common tasks like
searching and filtering, and also more complex tasks like analysis, decision making, reasoning and integration.

For example, in the e-tourism domain one can note an increasing number of travel agencies offering online services through online transaction brokers (Laudon & Traver, 2004). They provide useful information to human users about hotels, flights, trains or restaurants, in order to help them plan 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 (Staab et al., 2002) and ultimately to enable its reuse for multiple purposes. In particular, for example, personal assistant agents can integrate travel and weather information to assist and advise humans in planning their weekends and holidays. Another interesting use of data harvested from the Web that has been recently proposed (Gottlob, 2005) is to feed business intelligence tasks, in areas like competitive analysis and intelligence.

Two emergent technologies that have been put forward to enable automated processing of information published on the Web are semantic markup (W3C Semantic Web Activity, 2007) and Web services (Web Services Activity, 2007). However, most of the current practices in Web publishing are still being based on the combination of traditional HTML-lingua franca for Web publishing (W3C HTML, 2007) with server-side dynamic content generation from databases. Moreover, many Web pages are using HTML elements that were originally intended for use in structure content (e.g., those elements related to tables), or 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.

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 originally concerned with locating specific pieces of information in text documents written in natural language (Lenhert & Sundheim, 1991) and then using them to populate a database or structured document. The field then expanded to cover extraction tasks from Web documents represented in HTML and attracted other communities including databases, electronic documents, digital libraries 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 chapter.

A wrapper is a program that is used for performing the data extraction task. On one hand, manual creation of Web wrappers is a tedious, error-prone and difficult task because of Web heterogeneity in both structure and content. On the other hand, construction of Web wrappers is a necessary step to allow more complex tasks like decision making and integration. Therefore, a lot of techniques for (semi-)automatic wrapper construction have been proposed. One application area that can be described as a success story for machine learning technologies is wrapper induction for Web data extraction. For a recent overview of state-of-the-art approaches in the field see Chang, Kayed, Girgis, and Shaalan (2006).

In this chapter, we propose a novel class of wrappers, L-wrappers (i.e., logic wrappers), that fruitfully combine logic programming paradigm with efficient XML processing technologies (W3C Extensible Markup Language (XML), 2007). Our wrappers have certain advantages over existing proposals: i) they have a declarative semantics, and therefore their specification is decoupled from their implementation; ii) they can be generated...
using techniques and algorithms inspired by inductive logic programming (ILP hereafter); iii) they are implemented using XSLT – the “native” language for processing XML documents (W3C Extensible Stylesheet Language Family (XSL), 2007); and iv) they have also a visual notation making them easier to read and understand than their equivalent XSLT coding.

The chapter is structured as follows. We start with a brief review of logic programming, XML technologies and related approaches to Web data extraction. Then, we discuss flat relational and hierarchical approaches to Web pages conceptualization for data extraction. We follow with a concise definition of L-wrappers covering both their textual and visual representations. Both flat and hierarchical cases are considered. Next, we discuss efficient algorithms for semi-automatic construction of L-wrappers. Then, we present an approach for implementing L-wrappers using XSLT transformation language. The last two sections of this chapter contain some pointers to future works, as well as a list of concluding remarks.

BACKGROUND

The goal of this section is to briefly review the main ingredients of our approach to Web data extraction, that is, XML technologies and logic programming. Finally, as the application of logic programming and XML to information extraction is not entirely new, we briefly provide an literature overview of related proposals.

XML Technologies for Data Extraction

The Web is now a huge information repository that is characterized by i) high diversity, that is, the Web information covers almost any application area, ii) disparity, that is, the Web information comes in many formats ranging from plain and structured text to multimedia documents and iii) rapid growth, that is, 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, so our core data sources are in fact HTML documents. HTML was initially devised for modeling the structure and content of Web documents, rather than their presentation layout. However, with the advent of graphic Web browsers, software providers like Microsoft or Netscape added many features to HTML that were mainly addressing the visual representation and interactivity of Web documents, rather than their structure and content. The effects of this process were that initially HTML was developed (and consequently used) in a rather unsystematic way. However, starting with HTML 4.01, W3C consortium enforced a rigorous standardization process of HTML that ultimately resulted in a complete redefinition of HTML as an XML application, known as XHTML.

In our work we make the assumption that Web documents already are or can be converted through a preprocessing stage to well-formed XML before being actually processed for extraction of interesting data. While clearly, data extraction from HTML can benefit from existing approaches for information extraction from unstructured texts, we state that preprocessing and conversion of HTML to a structured (i.e., tree-like or well-formed XML) form has certain obvious advantages: i) an extracted item can depend on its structural context in a document, while this information is lost in the event the tree document is flattened as a string; ii) data extraction from XML documents can benefit from the plethora of XML query and transformation languages and tools.

A Web 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. The content part of a document
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

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

We model semistructured Web documents as labeled ordered trees. The node labels of a labeled ordered tree correspond to HTML tags. In particular, a text element will be considered to have a special tag text. Let \( \Sigma \) 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 chapter we are using some basic graph terminology as introduced in Cormen, Leiserson, and Rivest (1990).

Figure 1 shows a labeled ordered tree with 25 nodes and tags in the set \( \Sigma = \{a, b, c\} \).

Intuitively, a wrapper takes a labeled ordered tree and returns a subset of extracted nodes. An extracted node can be viewed as representing the whole subtree rooted at that node. The structural context of an extracted node is a complex condition that specifies i) 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 \( ? \)) and ii) certain conditions on node labels and their position (e.g., is the tag label \( td \ ? \), is it the first child \( ? \)). This conditions are nicely captured as conjunctive queries represented using logic programming (see next section).

Logic Programming for Representation and Querying of Web Documents

The rapid growth of the Web gave a boost to research on techniques to cope with the information flood. At the core of the various applications that include tasks like data retrieval, data extraction, and text categorization there are suitable representations of Web documents to allow their efficient structured querying and processing. In this subsection we show how logic programming can be used to achieve this desiderate.

Logic programming (Sterling & Shapiro, 1994) was originally developed within the artificial intelligence community to help with the implementation of natural language processing tools. However, its attractive features including declarative semantics, compact syntax, built-in reasoning capabilities, and so forth, together with efficient compilation techniques, made logic programming a suitable paradigm for the development of high-level general-purpose programming languages; see, for example, the Prolog language. Moreover, during the last decade applications of logic programming spread also to the areas of the Web and the Semantic Web (Alferes, Damasio, & Pereira, 2003).

A logic program is a set of logic statements that are classified as facts, rules and queries. Facts and rules are used to describe the problem

---

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

ii. \( l : T \to \Sigma \) is a node labeling function.

iii. \( c \subseteq T \times T \) is the parent-child relation between tree nodes, that is, \( c = \{(v, u) \mid \text{node } u \text{ is the parent of node } v\} \).

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, that is, \( (v_i, v_{i+1}) \in n \) for all \( 1 \leq i < k \).
domain, while queries are used to pose specific problem instances and to retrieve the corresponding solutions as query answers. Intuitively, the computation associated to a logic program can be described as the reasoning process for determining suitable bindings for the query variables such that the resulting instance of the query is entailed by the facts and rules that comprise the logic program.

Consider, for example, the logic programming representation of a Web document tree. We assign a unique identifier (an integer value) to each node of the tree. Let $N$ be the set of all node identifiers.

The structural component of a Web document can be represented as a set of facts that use the following relations Box 1.

In order to represent the content component of a Web document, we introduce two sets: the set $S$ of content elements that denote strings attached to text nodes and values assigned to HTML attributes, and the set $A$ of HTML attributes. With these notations, the content part of a Web document tree can be represented using two relations (Box 2).

Consider the Hewlett Packard’s Web site of electronic products and the task of data extraction from a product information sheet for Hewlett Packard printers. The printer information is displayed

**Box 1.**

i. $\text{child} \subseteq N \times N$ defined as $\text{child}(P, C) \iff P$ is the parent of $C$.

ii. $\text{next} \subseteq N \times N$ defined as $\text{next}(L, R) \iff L$ is the left sibling of $R$.

iii. $\text{tag}_\sigma \subseteq N, \sigma \in \Sigma$ defined as $\text{tag}_\sigma(N) \iff$ the tag of node $N$ is $\sigma$.

iv. $\text{first} \subseteq N$ defined as $\text{first}(X) \iff X$ is the first child of its parent node.

v. $\text{last} \subseteq N$ defined as $\text{last}(X) \iff X$ is the last child of its parent node.
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

in a two-column table as a set of feature-value pairs (see Figure 2a). Our task is to extract the names or the values of the printer features. This information is stored in the leaf elements of the page. Figure 2b displays the tree representation of a fragment of this document and Figure 3c displays the logic programming representation of this fragment as a set of facts.

Considering the example from Figure 2 and assuming that we want to extract all the text nodes of this Web document that have a grand-grand-parent of type table that has a parent that has a right sibling, we can use the following query. Note that for expressing logic programs we are using the standard Prolog notation (Sterling, 1994):

\[
? \text{tag(A, text), child(B, A), child(C, B), child(D, C), tag(D, table), child(E, D), next(E, F).}
\]

The query can be more conveniently packed as a rule as follows:

\[
\text{extract(A) :- tag(A, text), child(B, A), child(C, B), child(D, C), tag(D, table), child(E, D), next(E, F).}
\]

The rule representation has at least three obvious advantages: i) modularity: the knowledge embodied in the query is encapsulated inside the body of the predicate extract; ii) reusability: the query can be more easily reused rather than having to fully copy the conjunction of conditions and iii) information hiding: the variables occurring in the right-hand side of the rule are hidden to the user, that is, running the initial version of the query would produce a tuple of variable bindings as solution \((A, B, C, D, E, \text{ and } F)\), while running the rule version would produce the single variable binding \(A\) as solution.

Related Works

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 have focused our 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 (1998) is one of the first papers describing a “relational learning program” called SRV. It uses an ILP 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, and so forth. 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 an ILP algorithm. The difference is that our approach has been explicitly devised to cope with tree structured documents, rather than string documents.
Chidlovskii (2003) describes a generalization of the notion of string delimiters developed for information extraction from string documents (Kushmerick, 2000) 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 tree data structures, which is very different from our 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 $\uparrow$, $\leftarrow$ and $\rightarrow$ can be easily simulated by our rules using the relations $\text{child}$ and $\text{next}$ in our approach.

Xiao, Wissmann, Brown, and Jablonski (2001) proposes 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 that XSLT wrappers are learned indirectly via L-wrappers, and our wrappers are capable of extracting tuples in a straightforward way, and therefore TE-agents are not needed. Additionally, our approach covers the hierarchical case, which is not addressed in Xiao et al. (2001).

Lixto (Baumgartner, Flesca, & Gottlob, 2001) is a visual wrapper generator that uses an internal logic programming-based extraction language called Elog. In Elog, a document is abstracted as a tree (similar to our work), rather than a string. 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. The differences between Elog and L-wrappers are at least two fold: i) L-wrappers are only devised for the extraction task and they use a classic logic programming approach, for example, an L-wraper can be executed without any modification by a standard Prolog engine. Elog was devised for both crawling and extraction and has a customized logic programming-like semantics, that is more difficult to understand; and ii) L-wrappers are efficiently implemented by translation to XSLT, a standard language for transforming XML documents, while for Elog the implementation approach is different (a custom interpreter has been devised from scratch).

Thomas (2000) introduces 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 type feature. 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.

Laender, Ribeiro-Neto, and Silva (2002) describes the DEByE (i.e., Data Extraction By Example) environment for Web data management. DEByE contains a tool that is capable of extracting 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 the example collection stage by means of a GUI metaphor called nested tables. DEByE also addresses 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).

Sakamaoto (2002) introduces tree wrappers for tuples extraction. 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 reason 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.

A new wrapper induction algorithm inspired by ILP is introduced in Anton (2005). 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 Anton (2005) aim to extract only single items, and there is no discussion of how to extend the work to tuples extraction.

Stalker (Muslea, Minton, & Knoblock, 2001) uses a hierarchical schema of the extracted data called embedded catalog formalism that is similar to our approach. However, the main difference is that Stalker abstracts the document as a string rather than a tree and therefore their approach is not able to benefit from existing XML processing technologies. Extraction rules of Stalker are based on a special type of finite automata called landmark automata, rather than logic programming, as our L-wrappers.

Concerning theoretical work, Gottlob and Koch (2004) 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 Gottlob, Koch, and Schultz (2004).

### CONCEPTUALIZING WEB PAGES FOR DATA EXTRACTION

Many Web pages are dynamically generated by filling in HTML templates with data obtained from relational data bases. We have noticed that most often such Web documents can be successfully abstracted as providing relational data as sets of tuples or records. Examples include search engines’ answer pages, product catalogues, news sites, product information sheets, travel resources, multimedia repositories, Web directories, and so forth.

Sometimes, however, Web pages contain hierarchically structured presentations of data for usability and readability reasons. Moreover, it is generally appreciated that hierarchies are very helpful for focusing human attention and management of complexity. Therefore, as most Web pages are developed by knowledgeable specialists in human-computer interaction design, we expect to find this approach in many designs of Web interfaces to data-intensive applications.

### Flat Relational Conceptualization

We adopt a standard relational model by associating to a Web data source a set of distinct attributes. Let $\mathcal{A}$ be the set of all attribute names and let $D \subseteq \mathcal{A}$ be the set of relational attributes associated to a given Web data source. An extracted tuple can be defined as a function $\text{tuple} : D \to \mathbb{N}$, such that for each attribute $a \in D$, $\text{tuple}(a)$ represents the document node extracted from the Web source as an instance of attribute $a$. Note that in practice, instead of an extracted node, a user is rather interested to get the HTML content of the node.
Let us consider, for example, the problem of extracting printer information from Hewlett Packard’s Web site. The printer information is represented in multisection, two column HTML tables (as shown in Figure 2a). Each row contains a pair consisting of a feature name and a feature value. Consecutive rows represent related features that are grouped into feature classes. For example, there is a row with the feature name ‘Processor speed’ and the feature value ‘300 Mhz.’ This row has the feature class ‘Speed/monthly volume.’ So, actually, this table contains triples consisting of a feature class, a feature name, and a feature value. The set of relational attributes is \( D = \{ \text{feature-class}, \text{feature-name}, \text{feature-value} \} \). The document fragment shown in Figure 2a contains three tuples:

\[
\begin{align*}
\text{feature-class:} & \ 'Speed/monthly volume,' \quad \text{feature-name:} \ 'Print speed, black (pages per minute),' \\
& \ 'Up to 50 ppm' \\
\text{feature-class:} & \ 'Speed/monthly volume,' \quad \text{feature-name:} \ 'First page out, black,' \\
& \ '8 secs'
\end{align*}
\]

Note that in this example some tuples may have identical feature classes. More generally, for some documents, distinct tuples might have identical attribute instances. Clearly, this happens when the document has a hierarchical structure. For such cases, a hierarchical conceptualization of the Web data source is more appropriate (see the next section).

Let us now show how logic programming can be employed to conveniently define wrappers for data extraction from Web pages that have been conceptualized as flat relational data sources. Anticipating (see next section on logic wrappers), we shall call such programs logic wrappers or L-wrappers.

A L-wrapper for extracting relational data operates on a target Web document represented as a labeled ordered tree and returns a set of relational tuples of nodes of this document. A L-wrapper for the printers example shown in Figure 2b is \((FN = \text{feature name}, FV = \text{feature value})\) (Box 3).

**Definition 2 (L-wrapper as set of logic rules)** A L-wrapper can be defined formally as a set of patterns represented as logic rules. Assuming that \( \mathcal{N} \) is the set of document tree nodes and \( \Sigma \) is the set of HTML tags, a L-wrapper is a logic program defining a relation \( \text{extract}(N_1, \ldots, N_k) \subseteq \mathcal{N} \times \cdots \times \mathcal{N} \). For each clause, the head is \( \text{extract}(N_1, \ldots, N_k) \) and the body is a conjunction of literals in the set \{child, next, first, last, \( (\text{tag}_\sigma)_{\sigma \in \Sigma} \)\}. Number \( k \) of extracted attributes is called wrapper arity and is equal to the number of elements of set \( D \).

**Box 3.**

\[
\text{extract}(FN, FV) \ :- \\
\text{tag}(FN, \text{text}), \text{text}(FV), \text{child}(C, FN), \text{child}(D, FV), \text{child}(E, C), \text{child}(H, G), \text{child}(I, F), \text{child}(J, I), \text{next}(J, K), \text{child}(F, E), \text{child}(G, D), \text{first}(J), \text{child}(K, L), \text{child}(L, H).
\]
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

Figure 2. Web document fragment

b. Tree view of a Web document c

Figure 3. A hierarchical HTML document and its schema

a. Sample HTML document containing hierarchically structured data

<table>
<thead>
<tr>
<th>Speed/monthly volume</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Print speed, black</td>
<td>Up to 50 ppm</td>
</tr>
<tr>
<td>First page out, black</td>
<td>8 secs</td>
</tr>
<tr>
<td>Processor speed</td>
<td>300 MHz</td>
</tr>
</tbody>
</table>

a. Graphic view of a Web document

tag(0, html)
ntag(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)
content(111, 'Print speed, black')
attribute value(108, border, '0')
next(101, 102)
next(102, 103)
next(103, 104)

b. Hierarchical schema

Red apple

| weight | 120 |
| color  | red |
| diameter | 8 |

Limited stock, order today!

Lemon

| weight | 70 |
| color  | yellow |
| height | 7 |
| width  | 4 |

Limited stock, order today!
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 also the left sibling of the grand-grand-grand-grandparent of the second node \((K)\).

**Hierarchical Conceptualization**

In this subsection, we propose an approach for utilizing L-wrappers to extract hierarchical data. The advantage would be that extracted data will be suitably annotated to preserve its hierarchical structure, as found in the Web page. Further processing of this data would benefit from this additional metadata to allow for more complex tasks, rather than simple searching and filtering by populating a relational database. For example, one can imagine the application of this technique to the task of ontology extraction, as ontologies are assumed to be natively equipped with the facility of capturing taxonomically structured knowledge.

Let us consider a very simple HTML document that contains hierarchical data about fruits (see Figure 3a). A fruit has a name and a sequence of features. Additionally, a feature has a name and a value. This is captured by the schema shown in Figure 3b. Note that this representation allows features to be fruit-dependent; for example, while an apple has an average diameter, a lemon has both an average width and an average height.

Abstracting the hierarchical structure of data, we can assume that the document shown in Figure 3a contains triples consisting of a fruit name, a feature-name and a feature-value. However, this approach has at least two drawbacks: i) redundancy, because distinct tuples might contain identical attribute instances, and ii) the intrinsic hierarchical structure of the data is lost, while it might convey useful information.

Following the hierarchical structure of this data, the design of a L-wrapper of arity 3 for this example can be done in two stages: i) derive a wrapper \(W_1\) for binary tuples \((fruit-name, list-of-features)\); and ii) derive a wrapper \(W_2\) for binary tuples \((feature-name, feature-value)\). Note that wrapper \(W_1\) is assumed to work on documents containing a list of tuples of the first type (i.e., the original target document), while wrapper \(W_2\) is assumed to work on document fragments containing the list of features of a given fruit (i.e., a single table from the original target document). For example, wrappers \(W_1\) and \(W_2\) can be defined as logic programs as shown in Box 4.

Note that for the combination of \(W_1\) and \(W_2\) into a single L-wrapper of arity 3, we need to extend the definition of a L-wrapper by adding a new argument to relation \(extract\) for representing the root node of the document fragment to which the wrapper is applied, that is, instead of \(extract(N_1, \ldots, N_k)\) we shall now have \(extract(R, N_1, \ldots, N_k)\). \(R\) is the new argument. Moreover, it is required that for all \(1 \leq i \leq k\), \(N_i\) is a descendant of \(R\) in the document tree. The resulted solution is shown in Box 5.

The final wrapper (assuming the index of document root node is 0) is shown in Box 6.

---

**Box 4.**

```prolog
extr_fruits(FrN,FrFs) :-
  tag(FrN,text),child(A,FrN),child(B,A),next(B,FrFs),child(C,FrFs),tag(C,p).
extr_features(FN,FV) :-
  tag(FN,text),tag(FV,text),child(A,FN),child(B,FV),next(A,B),
  child(C,B),tag(C,tr).
```

---
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

Box 5

% ancestor(Ancestor,Node).
ancestor(N,N).
ancestor(A,N) :-
    child(A,B), ancestor(B,N).
extr_fruits(R,FrN,FrFs) :-
    ancestor(R,FrN), ancestor(R,FrFs), extr_fruits(FrN,FrFs).
extr_features(R,FN,FV) :-
    ancestor(R,FN), ancestor(R,FV), extr_features(FN,FV).

Box 6.

extract(FrN,FN,FV) :-
extr_fruits(0,FrN,FrFs), extr_features(FrFs,FN,FV).

Figure 4. Wrapping hierarchically structured data

While simple, this solution has the drawback that, even if it was devised with the idea of hierarchy in mind, it is easy to observe that the hierarchical nature of the extracted data is lost.

Assuming a Prolog execution engine of L-wrappers, we can solve the drawback using the findall predicate. findall($X$, $G$, $X$s) returns the list $X$s of all terms $X$ such that goal $G$ is true (it is assumed that $X$ occurs in $G$). The solution and the result are shown in Figure 4. Note that we assume that i) the root node of the document has index 0, and ii) predicate content(TextNode, Content) is used to determine the content of a text node.
**Definition 3** (Pattern graph) Let \( \mathcal{W} \) be a set denoting all vertices. A pattern graph \( G \) is a quadruple \( \langle A, V, L, \lambda \rangle \) such that \( V \subseteq \mathcal{W} \), \( A \subseteq V \times V \), \( L \subseteq V \) and \( \lambda : A \rightarrow \{ 'c', 'n' \} \). The set \( G \) of pattern graphs is defined inductively as follows:

1. If \( v \in \mathcal{W} \) then \( \{ \emptyset, \{ v \} \} \) is a graph.
2. If \( G = \langle A, V, L, \lambda \rangle \) is a graph, then for \( w, v, u \in V \) and \( L = \{ 'c', 'n' \} \),

   - a) \( G_1 = \langle A \cup \{ (w, v) \}, V \cup \{ w \}, \lambda \cup \{ ((w, v), 'n') \} \rangle \)
   - b) \( G_2 = \langle A \cup \{ (u_i, v) \}, V \cup \{ u_i \}, \lambda \cup \{ ((u_i, v), 'c') \} \rangle \)
   - c) \( G_3 = \langle A \cup \{ (w, v), (u_i, v) \}, V \cup \{ w, u_i \}, \lambda \cup \{ ((w, v), 'n'), ((u_i, v), 'c') \} \rangle \)

Then \( G \) is also a graph.

**Logic Wrappers as Directed Graphs**

In this section, we take a graph-based perspective in defining L-wrappers as sets of patterns. Within this framework, a pattern is a directed graph with labeled arcs and vertices that correspond to a rule in the logic representation. Arc labels denote conditions that specify the tree delimiters of the extracted data, 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 the "parent-child" 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 \( \sigma \)” condition.

Patterns are matched against parts of a target document modeled as a 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. The result of applying a pattern to a labeled ordered tree is a set of tuples of extracted nodes.

Patterns can be concisely defined in two steps: i) define the pattern graph together with arc labels that model parent-child and next-sibling relations, and ii) extend this definition with vertex labels that model conditions on vertices, extraction vertices and assignment of extraction vertices to attributes.

Intuitively, if \( \langle A, V, L, \lambda \rangle \) is a pattern graph, then \( V \) denotes its set of vertices, \( A \) denotes its set of arcs, \( L \subseteq V \) are its leaves (vertices with in-degree 0) and \( \lambda \) distinguishes between parent-child (labeled with 'c') and next-sibling (labeled with 'n') arcs. Note also that a pattern graph is tree shaped with arcs pointing up.

Note that according to definition 4, we assume that extraction vertices are among the leaves of the pattern graph, that is, an extraction pattern does not state any condition about the descendants of an extracted node. This is not restrictive in the context of patterns for information extraction from Web documents.

**Definition 4** (L-wraper pattern) Let \( \mathcal{A} \) be the set of attribute names. An **L-wraper pattern** is a tuple \( p = \langle A, U, D, \mu, \lambda \rangle \) such that \( \langle A, V, L, \lambda \rangle \) is a pattern graph, \( U = \{ u_i \} \subseteq L \) is the set of pattern extraction vertices, \( D \subseteq \mathcal{A} \) is the set of attribute names, \( \mu : D \rightarrow U \) is a one-to-one function that assigns a pattern extraction vertex to each attribute name, and \( \lambda : V \rightarrow C \) is the labeling function for vertices. \( C = \{ \emptyset, \{ 'f' \}, \{ 'l' \}, \{ \sigma \}, \{ 'f', 'l' \}, \{ 'f', \sigma \}, \{ 'l', \sigma \}, \{ 'f', 'l', \sigma \} \} \) is the set of conditions, where \( \sigma \) is a label in the set \( \Sigma \) of tag symbols.
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

Figure 5. L-wrapper both as directed graph and as logic program

\[ 
\begin{align*}
\text{extract}(F,D) &::= \\
\text{next}(F,G), \text{child}(G,D), \text{tag}(G,c), \\
\text{child}(H,G), &\text{tag}(H,b)
\end{align*} \]

\( a. \) L-wrapper as directed graph       \( b. \) Same L-wrapper as logic program

**Definition 5** (L-wrapper) An *L-wrapper* of arity \( k \) is a set of \( n \geq 1 \) patterns \( W = \{p_i|p_i = (V_i, A_i, U_i, D_i, \mu_i, \lambda_a^i, \lambda_c^i)\} \), such that each \( p_i \) 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. \( \text{Ans}(W, t) = \cup_{1 \leq i \leq n} \text{Ans}(p_i, t) \).

**Definition 6** (Schema tree) Let \( \mathcal{W} \) be a set denoting all vertices. A schema tree \( S \) is a directed graph defined as a quadruple \( (A, V, L, \lambda_a) \) s.t. \( V \subseteq \mathcal{W}, A \subseteq V \times V, L \subseteq V \) and \( \lambda_a : A \rightarrow \{\ast, \ast \} \).

The set of schema trees is defined inductively as follows:

i. For all \( n \geq 1 \), if \( u, v, w_i \in \mathcal{W} \) for all \( 1 \leq i \leq n \) then \( S = (A, V, L, \lambda_a) \) such that \( V = \{u, v, w_1, \ldots, w_n\}, A = \{(u,v), (v,w_1), \ldots, (v,w_n)\}, L = \{w_1, \ldots, w_n\} \), \( \lambda_a((u,v)) = \ast \) and \( \lambda_a((v,w_i)) = \ast \) for all \( 1 \leq i \leq n \).

ii. If \( S = (A, V, L, \lambda_a) \) is a schema tree, \( n \geq 1 \), \( u \in L \) and \( v, w_i \in \mathcal{W} \setminus V \) for all \( 1 \leq i \leq n \) then \( S' = (A', V', L', \lambda_a') \) defined as \( V' = V \cup \{w_i, \ldots, w_n\}, A' = A \cup \{(u, v), (v,w_i), \ldots, (v,w_n)\}, L' = (L \setminus \{u\}) \cup \{w_i, \ldots, w_n\} \), \( \lambda_a'((u,v)) = \ast \) and \( \lambda_a'((v,w_i)) = \ast \) for all \( 1 \leq i \leq n \) is also a schema tree.

Figure 5 shows a single-pattern L-wrapper of arity 2 represented both as a directed graph and a logic program. The extraction vertices are marked with small arrows (vertices \( F \) and \( D \) on that figure). Note also that the figure shows the attributes extracted by the extraction vertices (attribute \( x \) extracted by vertex \( F \) and attribute \( y \) extracted by vertex \( D \)).

If \( p \) is a pattern and \( t \) is a Web document represented as a labeled ordered tree, we denote with \( \text{Ans}(p,t) \) the set of tuples extracted by \( p \) from \( t \). For a formal definition of function \( \text{Ans} \) see Bădică (2006).

A *L-wrapper* can comprise more patterns so it can be defined formally as a set of extraction...
patterns that share the set of attribute names. This idea is captured by definition 5.

Let us now formally introduce the concept of **hierarchical logic wrapper** or HL-wrapper. We generalize the data source schema from flat relational to hierarchical and we attach to this schema a set of L-wrappers.

If $\Sigma$ is a set of tag symbols denoting schema concepts and $S$ is a schema tree then a pair consisting of a schema tree and a mapping of schema tree vertices to $\Sigma$ is called a **schema**. For example, for the schema shown in Figure 3b, $\Sigma = \{fruits, fruit, features, feature, feature-name, feature-value\}$ (note that on that figure labels ‘l’ are not explicitly shown). For a L-wrapper corresponding to the relational case if $D$ is the set of attribute names, then $\Sigma = D \cup \{result, tuple\}$. Here, result denotes the tag of the root element of the output XML document containing the extracted tuples and tuple is a tag used to demarcate each extracted tuple; see example in Bădică (2006). Also, it is not difficult to see that in an XML setting a schema nicely corresponds to the document type definition of the output document that contains the extracted data.

An HL-wrapper for the example document considered in this paper consists of: i) schema shown in Figure 3b, ii) L-wrapper $W_1$ assigned to the vertex labeled with symbol fruit, and iii) L-wrapper $W_2$ assigned to vertex labeled with symbol feature.

### EFFICIENT ALGORITHMS FOR AUTOMATED CONSTRUCTION OF LOGIC WRAPPERS

Inductive logic programming is one of the success stories in the application area of wrapper induction for information extraction. However, this approach suffers from two problems: high computational complexity with respect to the

---

**Definition 7** (HL-wrapper) A HL-wrapper consists of a schema and an assignment of L-wrappers to split vertices of the schema tree. A vertex $v$ of the schema tree is called **split vertex** if it has exactly one incoming arc labeled ‘*’ and $n \geq 1$ outgoing arcs labeled ‘l.’ An L-wrapper assigned to $v$ must have arity $n$ to be able to extract tuples with $n$ attributes corresponding to outgoing neighbors of vertex $v$.

**Definition 8** (Extraction path) An **extraction path** is a labeled directed graph that is described as a list $[t_0, t_1, \ldots, t_k]$ with the following properties:

i. Each element $t_i$, $0 \leq i \leq k$ is a list $[v_{i,0}, \ldots, v_{i,l}, v_{i+1}, \ldots, v_i]$, $l \geq 0$, $r \geq 0$ such that: i) $v_i$, $-l < i < l$ are vertices; ii) $(v_i, v_{i+1})$, $-l < i < r$ are arcs labeled with ‘n’, and iii) for each pair of adjacent lists $t_i$ and $t_{i+1}$, $1 \leq i < k$ in the extraction path, $(v_0, v_0)$ is an arc labeled with ‘c’.

ii. Vertex labels are defined as: i) if $l > 0$ then $v_{i,j}$ is labeled with a subset of $\{\text{'f'}, \sigma\}$, $\sigma \in \Sigma$; ii) $v_i$, $-l < i < 0$ is labeled with a subset of $\{\sigma\}$, $\sigma \in \Sigma$; iii) if $r > 0$ then $v_i$ is labeled with a subset of $\{\text{'l'}, \sigma\}$, $\sigma \in \Sigma$; iv) $v_i$, $1 < i < r$ is labeled with a subset of $\{\sigma\}$, $\sigma \in \Sigma$; v) if $l > r > 0$ then $v_0$ is labeled with a subset of $\{\sigma\}$, $\sigma \in \Sigma$; if $l = r = 0$ then $v_0$ is labeled with a subset of $\{\text{'l'}, \text{'f'}, \sigma\}$, $\sigma \in \Sigma$.

Vertex $v_0^k$ (i.e. $v_0$ in list $t_k$) is matched against the extraction node and consequently is called **extraction vertex**.
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

Figure 6. Extraction paths for example nodes in Figure 1

\[ a. \text{Extraction path for example node } n_{12} \]

\[ b. \text{Extraction path for example node } n_{22} \]

**Definition 9** (Height and widths of an extraction path) Let \( p = [t_0, t_1, \ldots, t_k] \) be an extraction path.

i. The value \( \text{height}(p) = k \) is called the **height** of \( p \).

ii. The value \( \text{left}(p) = \max_{0 \leq i \leq k} \text{left}(t_i) \) is called the **left width** of \( p \). The value \( \text{right}(p) = \max_{0 \leq i \leq k} \text{right}(t_i) \) is called the **right width** of \( p \).

**Definition 10** (Bounded extraction path) Let \( H, L, R \) be three positive integers. An extraction path \( p \) is called \((H, L, R)\)-bounded if \( \text{height}(p) \leq H \), \( \text{left}(p) \leq L \) and \( \text{right}(p) \leq R \).

number of nodes of the target document and to the arity of the extracted tuples. In this chapter, we address the first problem by proposing a path generalization algorithm for learning rules to extract single information items. The algorithm produces a pattern (called extraction path) from positive examples and is proven to have good computational properties. The idea of this algorithm can also be used to devise an algorithm for learning tuples extraction paths from positive examples. Finally, note this algorithm can also be adapted to generate multiple pattern L-wrappers from both positive and negative examples.

**Extraction Paths**

Basically, an extraction path is a L-wraper pattern for extracting single items, that is, a pattern of arity equal to 1. See examples in Figure 6.

Consider an extraction path \( p = [t_0, t_1, \ldots, t_k] \). For a list \( t = [v_0, v_1, v_2, \ldots] \) in \( p \) let \( \text{left}(t) = l \) and \( \text{right}(t) = r \). The following definition introduces height, together with left and right widths of an extraction path.

In practice it is useful to limit the height and the widths of an extraction path, yielding a bounded extraction path.

Note that the extraction path shown in Figure 6a is a \((3, 2, 1)\)-bounded extraction path. Moreover, if we restrict \( H = 2 \) and \( L = 1 \), then nodes \( J \) and
A subpath will be pruned, resulting a (2, 1, 1)-bounded extraction path, that obviously is less constrained than the initial path.

**Learning Extraction Paths**

The practice of Web publishing assumes dynamically filling in HTML templates with structured data taken from relational databases. Thus, we can safely assume that a lot of Web data is contained in sets of documents that share similar structures. Examples of such documents are: search engines’ result pages, product catalogues, news sites, product information sheets, travel resources, and so forth.

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.

Usually the extraction task is focused on extracting similar items (like book titles in a library catalogue or product features in a product information sheet). One approach to generate an extraction rule from a set of examples is to discover a common pattern of their neighboring nodes in the tree of the target document.

In what follows, we discuss an algorithm that takes: i) an XML document (possibly assembled from many Web pages, previously preprocessed and converted to well-formed XML) that is modeled as a labeled ordered tree \( t \); ii) a set of example nodes \( \{e_1, e_2, \ldots, e_n\} \); and iii) three positive integers \( H, L, R \), and that produces an \((H, L, R)\)-bounded extraction path \( p \) that generalizes the set of input examples. Intuitively, this technique is guaranteed to work if we assume that semantically similar items will exhibit structural similarities in the target Web document. This is a feasible assumption for the case of Web documents that are generated on-the-fly by filling in HTML templates with data taken from databases. Moreover, based on experimental results recorded in our work (Bădică, 2006), we have noticed that in practice an extraction rule only needs to check the proximity of nodes. This explains why we focused on the task of learning bounded extraction paths.

The basic operation of the learning algorithm is the generalization operator of two extraction paths. This operator takes two extraction paths \( p_1 \) and \( p_2 \) and produces an appropriate extraction path \( p \) that generalizes \( p_1 \) and \( p_2 \).

---

**Figure 7. Extraction path generalization algorithm**

---
The idea of the learning algorithm is as follows. For each example node we generate a bounded extraction path (of given input parameters $H$, $L$, $R$) by following sibling and parent links in the document tree. We initialize the output path with the first extraction path and then we proceed by iterative application of the generalization operator to the current output path and the next example extraction path, yielding a new output path. The result is a bounded extraction path that represents an appropriate generalization of the input examples.

The generalization of two paths assumes the generalization of their elements, starting with the elements containing the extraction vertices and moving upper level by level. The generalization of two levels assumes the generalization of each pair of corresponding vertices, starting with vertices with index 0 and moving to the left and respectively to the right in the lists of vertices. Generalization of two vertices is as simple as taking the intersection of their labels. The algorithm is shown in Figure 7.

Function LEARN generalizes the extraction paths of the example nodes. We assume that paths $p_1, \ldots, p_n$ are generated as bounded extraction paths before function LEARN is called. Function GEN-PATH takes two extraction paths $p_1$ and $p_2$ and computes their generalization $p$. Function GEN-LEVEL takes two lists of vertices $t_1$ and $t_2$ that are members of the extractions paths and computes a generalized list $t$ that is member of the generalization path. Function GEN-VERTEX takes two vertices $v_1$, $v_2$ and computes a generalized vertex $v$.

It is not difficult to see that the execution of algorithm LEARN takes time $O(n \times H \times (L + R))$ because GEN-VERTEX takes time $O(1)$, GEN-LEVEL takes time $O(L + R)$ and GEN-PATH takes time $O(H \times (L + R))$. Note also that if we set $H = L = R = 1$, then the complexity of the algorithm is $O(n \times H' \times W')$ where $H'$ and $W'$ are the height and the width of the target document tree.

Consider again the labeled ordered tree shown in Figure 1 and the example nodes marked with dashed rectangles ($n_{12}$ and $n_{22}$). The extraction paths corresponding to these nodes are shown in Figure 6. The result of applying the generalization algorithm on these paths is shown in Figure 8a.
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

In this section we describe an approach for the efficient implementation of L-wrappers using XSLT transformation language—a standard language for transforming XML documents. We start with introducing XSLT \( 0 \)—an expressive subset of XSLT that has a formal operational semantics. Then we describe the algorithm for translating L-wrappers into XSLT \( 0 \) programs.

Translating Extraction Paths to XPath

An extraction path can be translated to an XPath query. The XPath query can be embedded into an XSLT stylesheet to finally extract the information and store it into a database or another structured document (W3C Extensible Stylesheet Language Family (XSL), 2007).

Figure 9 shows an algorithm for translating an extraction path into XPath. Actually the algorithm takes following route of vertices: \( v_0^0 \rightarrow \cdots \rightarrow v_i^0 \rightarrow v_i^1 \rightarrow \cdots \rightarrow v_j^1 \rightarrow \cdots \rightarrow v_k^1 \rightarrow \cdots \rightarrow v_{\ell-1}^1 \rightarrow v_{\ell}^1 \rightarrow \cdots \rightarrow v_k^\ell \rightarrow \cdots \rightarrow v_0^\ell \). Note that when moving from element \( i \) to element \( i+1 \), \( 0 \leq i < k \), the algorithm takes the route \( v_0^i \rightarrow v_i^i \) (opposite direction of dotted arrows in figure 8a) rather than the route \( v_0^i \rightarrow v_i^i \). For each vertex \( v_0^i \), \( 0 \leq i \leq k \), the algorithm also generates a condition that accounts for their left siblings by taking the route \( v_0^i \rightarrow v_i^i \rightarrow v_j^i \). It is easy to see that if \( p \) is an \( (H, L, R) \)-bounded extraction path then the time complexity of the translation algorithm PATH-TO.XPATH is \( O(H \times (L + R)) \).

Figure 8b shows the result of applying this algorithm to the extraction path from figure 8a. The algorithm will explore the following route of vertices: \( I \rightarrow H \rightarrow G \rightarrow F \rightarrow D \rightarrow C \). For each vertex the algorithm generates a location step comprising an axis specifier, a node test and a sequence of predicates written between \([\[ \) and \( ]\). The node test is always \(*\). The axis specifier is determined by the relation of the current vertex with its preceding vertex on the route explored by the translation algorithm. For example, the axis specifier that is generated for vertex \( F \) is \( \text{preceding-sibling}\::\). In this later case, an additional predicate \([1]\) that constrains...
the selection of exactly the preceding node, is added. The algorithm generates also a predicate for each element of the label of a vertex. For example, predicate \( [\text{local-name()} = 'a'] \) is generated for vertex \( F_c \) that checks the node tag, and predicate \( [\text{not (preceding-sibling::*]} \) is generated for vertex \( G_c \), that checks if the matched node is the last child of its parent node. Moreover, for vertices \( F \) and \( C \) the algorithm generates an additional predicate that accounts for their left siblings \( E \) (of \( F \)) and respectively \( B \rightarrow A \) (of \( C \)). For example, additional predicate \( [\text{preceding-sibling::*}  \) is generated for vertex \( F \). This predicate checks if the document node matched by vertex \( F \) has a predecessor and if the predecessor is the first child of its parent node.

Note that running the XPath query from figure 4b on the document represented by the labeled ordered tree from figure 1 produces the following two answers \( /a[1]/a[1]/a[1]/b[3] \) and \( /a[1]/c[1]/a[1]/c[4] \) that correspond to nodes \( n_{12} \) and \( n_{22} \).

**XSLT\(_0\) Transformation Language**

\( \text{XSLT}_0 \) is a subset of XSLT that retains most of its features and additionally has a formal operational semantics, and a cleaner and more readable syntax. In what follows, we just briefly review \( \text{XSLT}_0 \) and its pseudocode notation. For more details on its abstract model and formal semantics, the reader is invited to consult Bex, Maneth, and Neven (2002).

An \( \text{XSLT}_0 \) program is a set of transformation rules. A rule can be either selecting or constructing. In what follows, we focus only on constructing rules, as we are only using constructing rules in translating L-wrappers into \( \text{XSLT}_0 \).

A \( (q, \sigma) \) constructing rule has the following general form:

**template** \( q(\sigma, x_1, \ldots, x_n) \)

**vardef**

\[ y_1 := r_1; \ldots; y_m := r_m \]

**return**

\[ \text{if } c_1 \text{ then } z_1; \ldots; \text{if } c_k \text{ then } z_k \]

**end**

Here:

i. \( q \) is an XSLT mode (actually a constructing mode) and \( \sigma \) is a symbol in \( \Sigma \cup \{\ast\} \);

ii. \( x_1, \ldots, x_n, y_1, \ldots, y_m \) are variables;

iii. Each \( r_i \) is an expression (possibly involving variables \( x_1, \ldots, x_n, y_1, \ldots, y_m \)) that evaluates to a data value (i.e., the value of an attribute or the content of an element);

iv. Each \( c_i \) is a test (possibly involving variables \( x_1, \ldots, x_n, y_1, \ldots, y_m \)) and thus it evaluates either to true or false; and

v. Each \( z_i \) is a forest (i.e., a (possibly empty) sequence of tree fragments) that is created by the constructing rule. The leaves of this forest are expressions of the form \( q'(p, z) \) such that \( q' \) is a constructing mode, \( p \) is an XPath pattern (possibly with variables), and \( z \) is a sequence (possibly empty) of variables from the set \( \{x_1, \ldots, x_n, y_1, \ldots, y_m\} \).

Additionally, we require the existence of a constructing mode \( \text{start} \) such that each \( z_i \) of a \( (\text{start}, \sigma) \) constructing rule is a tree (rather than a forest). This constraint ensures that the output of an \( \text{XSLT}_0 \) program is always a tree. Also, for each rule, if none of tests \( c_i \) succeeds, the empty forest will be output. Finally, to ensure that the model is deterministic, we require that for any two \( (q_1, \sigma_1) \) and \( (q_2, \sigma_2) \) rules either \( q_1 \neq q_2 \) or \( q_1 = q_2 \) and \( \sigma_1 \neq \sigma_2 \) and \( \sigma_1 \neq \ast \) and \( \sigma_2 \neq \ast \).

An \( \text{XSLT}_0 \) program defines a computation as a sequence of steps, that transforms an input labeled ordered tree \( t \) with root \( r \) into an output tree. At each step, the computation state is recorded as a tree such that some of its leaves are configura-
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

Figure 10. Algorithm for translating an L-wrapper into XSLT₀

<table>
<thead>
<tr>
<th>GEN-WRAPPER(W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. let w ∈ LW</td>
</tr>
<tr>
<td>2. L ← LW \ {w}</td>
</tr>
<tr>
<td>3. GEN-FIRST-TEMPLATE(w)</td>
</tr>
<tr>
<td>4. while L ≠ ∅ do</td>
</tr>
<tr>
<td>5. w₀ ← w</td>
</tr>
<tr>
<td>6. V ← ∅</td>
</tr>
<tr>
<td>7. let w’ be another vertex in L \ {w₀}</td>
</tr>
<tr>
<td>8. let w be the first common ancestor of w₀ and w’</td>
</tr>
<tr>
<td>9. if w₀ ∈ U₀ then</td>
</tr>
<tr>
<td>10. var ← VAR-GEN(μ₀(w₀))</td>
</tr>
<tr>
<td>11. GEN-TEMPLATE-WITH-VAR(w₀ ← w, w’ ← w, var, V)</td>
</tr>
<tr>
<td>12. V ← V ∪ {var}</td>
</tr>
<tr>
<td>13. else</td>
</tr>
<tr>
<td>14. GEN-TEMPLATE-NO-VAR(w₀ ← w, w’ ← w, V)</td>
</tr>
<tr>
<td>15. w ← w’</td>
</tr>
<tr>
<td>16. L ← L \ {w}</td>
</tr>
<tr>
<td>17. GEN-LAST-TEMPLATE(w, V)</td>
</tr>
</tbody>
</table>

The computation stops when the current state is a labeled ordered tree (i.e. it does not contain any configurations as leaves). The result of a computation is the tree representing its final state.

Mapping L-Wrappers to XSLT₀

Let W = <V, A, U, D, μ, λ> be a single-pattern L-wrapper and let L ⊆ V be the set of leaves of its pattern graph. Recall that we assumed that all extraction vertices are in L, that is U ⊆ L. Note that if u and v are vertices of the pattern graph then u ↪ v denotes the path from vertex u to vertex v in this graph. For example, referring to Figure 5a, F ↪ H = F, G, H.

Let L = {w₁, ..., wₙ} be the leaves and let w be the root of the pattern graph. The idea of the translation algorithm is as follows. We start from root w and move down in the graph to w₁, that is, w₁ ↪ w. Then, we move from w₁ to w₂ via their closest common ancestor w′₁ that is, w₁ ↪ w′₁ and w₂ ↪ w′₁, ..., and we move from wₙ₋₁ to wₙ via their closest common ancestor w′ₙ₋₁ that is, wₙ₋₁ ↪ wₙ₋₁' and wₙ ↪ wₙ₋₁'.

The intuition behind the application of a constructing rule is as follows. First, variables y_i and conditions c_j are evaluated. Let us assume that, as result of evaluating tests c_j-s, forest z’ with leaves of the form q’(p, z), is returned. Second, pattern p is applied to node u of t yielding a sequence of nodes u₁, ..., u_l in document order. Third, a new forest f is computed by substituting the variables in z with their values and leaves q’(p, z) of z with sequences of configurations (u₁, q’, d’), ..., (u_l, q’, d’). Here, d’ are the new values assigned to variables x_i and y_j. Fourth, next state is computed by substituting configuration (u, q, d) of the current state with f.

The computation stops when the current state is a labeled ordered tree (i.e. it does not contain any configurations as leaves). The result of a computation is the tree representing its final state.
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

For example, referring to Figure 5a, we start from the root $H$ and move downward to leaf $F$; that is, $F \leadsto H$. Then, we move from $F$ to $D$ via their closest common ancestor $G$, that is, $F \leadsto G$ and $D \leadsto G$.

Template rules are generated according to this traversal of the pattern graph. The first rule is generated according to path $w_1 \leadsto w$. The next $n-1$ rules are generated according to paths $w_i \leadsto w_i'$ and $w_{i+1} \leadsto w_{i+1}'$, $1 \leq i \leq n-1$. Finally, the last rule is generated for vertex $w_n$. Thus, a total of $n+1$ rules are generated.

The resulting GEN-WRAPPER translation algorithm is shown in Figure 10. Note that function VAR-GEN generates a new variable name based on the attribute associated to an extraction vertex.

GEN-FIRST-TEMPLATE algorithm generates the first template rule. Let $p_1$ be an XPath pattern that accounts for the conditions on the path $w_1 \leadsto w$. Then, the template rule that is firstly generated has the following form:

```
template \text{start}() 
return \text{result}(sel_w(p_1))
end 
```

GEN-TEMPLATE-WITH-VAR and GEN-TEMPLATE-NO-VAR algorithms generate a template rule for paths $w_i \leadsto w_i'$ and $w_{i+1} \leadsto w_{i+1}'$, $1 \leq i \leq n-1$, depending on whether vertex $w_i$ is an extraction vertex or not. Note that a new variable is generated only if $w_i$ is an extraction vertex, to store extracted data. Let $p_{i+1}$ be the XPath pattern that accounts for the conditions on the paths $w_i \leadsto w_i'$ and $w_{i+1} \leadsto w_{i+1}'$.

The structure of a template rule for the case when a new variable $var$ is generated is shown below:

```
template \text{sel}_{w_i}(*', V) 
vardef var := content(.) 
return sel_{w_{i+1}}(p_{i+1}, V \cup \{var\})
end
```

The structure of a template rule for the case when no new variable is generated is shown below:

```
template \text{sel}_{w_i}(*', V) 
return sel_{w_{i+1}}(p_{i+1}, V)
end
```

GEN-LAST-TEMPLATE algorithm generates the last template rule. The constructing part of this rule fully instantiates the returned tree fragment, thus stopping the transformation process of the input document tree. Depending on whether vertex $w_n$ is an extraction vertex or not, this template rule generates or does not generate a new variable. We assume that the set $D$ of attribute names is $\{d_1, ..., d_n\}$. Note that because a new variable is generated for each extraction vertex, it follows that the number of generated variables is $n$.

If a new variable $var$ is generated, then the last generated rule has the following form:

```
template \text{sel}_{w_n}(*', V) 
vardef var := content(.) 
return tuple((d_1(var_1), \ldots, d_n(var_n))
end
```

Here $V \cup \{var\} = \{var_1, ..., var_n\}$.

If no new variable is generated, then the last generated rule has the following form:
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

```
template sel_{\alpha}(\ast\ast, V)
return
tuple((d_1(var_1), \ldots, d_n(var_n))
end

Here V = \{var_1, \ldots, var_n\}.

Example L-Wrappers in XSLT

Let us first consider an example for the flat relational case. Applying algorithm GEN-WRAPPER to the L-wrapper shown in Figure 5a, we obtain an XSLT_0 program comprising three rules:

```
template start(/)
result ((selx(p_1))
end
template selx(\ast\ast)
vardef vx := content(.)
return
selx(p_2, vx)
end
template sely(\ast\ast, vx)
vardef vy := content(.)
return
tuple(x(vx), y(vy))
end
```

XPath pattern \(p_1 = //b/c/preceding-sibling::*[1]\) is determined by tracing the path \(H \sim G \sim F\) in the pattern graph (see Figure 5a). XPath pattern \(p_2 = following-sibling::*[1]/\ast\ast\) is determined by tracing the path \(F \sim G \sim D\) in the pattern graph (see Figure 5a).

The XSLT code for this wrapper is shown in Box 7.

Applying this XSLT transformation to the document shown Box 8.

Let us now consider the hierarchical example shown in Figure 3. The HTML code corresponding to the view shown Figure 3a is shown in Box 9.

An XSLT implementation for an HL-wrapper can be obtained by combining the idea of the hierarchical Prolog implementation with the translation of L-wrappers to XSLT outlined in the previous section.

A single-pattern L-wrapper for which the pattern graph has \(n\) leaves, can be mapped to an XSLT_0 stylesheet consisting on \(n+1\) constructing rules. In our example, applying this technique to each of the wrappers \(W_1\) and \(W_2\) (devised for the hierarchical source from Figure 3) we get three rules for \(W_1\) (start rule, rule for selecting \textit{fruit name} and rule for selecting \textit{features}) and three rules for \(W_2\) (start rule, rule for selecting \textit{feature name} and rule for selecting \textit{feature value}). Note that in addition to this separate translation of \(W_1\) and \(W_2\), we need to assure that \(W_2\) selects feature names and feature values from the document fragment corresponding to a given fruit, that is, the document fragment corresponding to the \textit{features} attribute of wrapper \(W_1\). This effect can be achieved by plugging in the body of the start rule corresponding to wrapper \(W_2\) into the body of the rule for selecting features, in-between tags \(<\textit{features}>\) and \(</\textit{features}>\) (see example below). Actually, this operation corresponds to realizing a join of the wrappers \(W_1\) and \(W_2\) on the attribute \textit{features} (assuming L-wrappers are extended with an argument representing the root of the document fragment to which they are applied.

The resulting HL-wrapper expressed in XSLT is shown in Box 10.

Note that this output faithfully corresponds to the hierarchical schema shown in Figure 3b.

For wrapper execution, we can use any of the available XSLT transformation engines. In our experiments we have used Oxygen XML editor (Oxygen XML Editor, 2007), a tool that incorporates some of these engines (see Figure 11).
FUTURE TRENDS AND CONCLUSION

In this chapter, we discussed a new class of wrappers for data extraction from semistructured sources inspired by logic programming, or L-wrappers. We described an inductive learning algorithm to generate extraction paths and also showed how to map the resulting extraction rules to XSLT stylesheets for efficient data extraction from Web sources. Our discussion covered two cases: i) extraction of relational tuples from flat relational Web data sources, and ii) extraction of hierarchical data from hierarchical Web data.
Box 8.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<a>
  <b>
    <t>x1</t>
    <c>
      <t>y1</t>
    </c>
  </b>
  <b>
    <t>x2</t>
    <d>
      <t>y2</t>
    </d>
  </b>
  <b>
    <t>x3</t>
    <c>
      <t>y3</t>
    </c>
  </b>
  <b>
    <t>x4</t>
    <c>
      <t>y4</t>
    </c>
  </b>
</a>
```

produces the following extracted data as output.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <tuple>
    <x>x1</x>
    <y>y1</y>
  </tuple>
  <tuple>
    <x>x3</x>
    <y>y3</y>
  </tuple>
</result>
```

sources. These ideas are currently being implemented in a tool for information extraction. As future work, we plan to finalize and evaluate the implementation and also to give a formal proof of the correctness of the mapping of L-wrappers to XSLT.

Currently, our approach is semi-automated rather than fully automated. There are two tasks that must be performed manually by the user: i) definition of the schema of the Web data source, either flat relational or hierarchical; and ii) extraction of a few examples. We plan to address these issues in our future research work.

REFERENCES


Box 9.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<html>
    <head>
        <title>Fruits.</title>
    </head>
    <body>
        <p>
            <h2>
                <th>Red apple</th>
            </h2>
            <table border="2">
                <thead>
                    <tr>
                        <td>weight</td>
                        <td>120</td>
                    </tr>
                    <tr>
                        <td>color</td>
                        <td>red</td>
                    </tr>
                    <tr>
                        <td>diameter</td>
                        <td>8</td>
                    </tr>
                </thead>
                <tbody>
                    <tr colspan="2">Limited stock, order today !</tr>
                </tbody>
            </table>
        </p>
        <p>
            <h2>
                <th>Lemon</th>
            </h2>
            <table border="2">
                <tbody>
                    <tr>
                        <td>weight</td>
                        <td>70</td>
                    </tr>
                </tbody>
            </table>
        </p>
    </body>
</html>
```
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

Box 9. continued

```html
<tr>
    <td>color</td>
    <td>yellow</td>
</tr>
<tr>
    <td>height</td>
    <td>7</td>
</tr>
<tr>
    <td>width</td>
    <td>4</td>
</tr>
</tbody>

Limited stock, order today!
```

Figure 11. Wrapper execution inside Oxygen XML editor

```
<html>
<body>
    <table>
        <tr>
            <td colspan="2">Limited stock, order today!</td>
        </tr>
    </table>
</body>
</html>
```
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

Box 10.

```xml
<?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">
    <fruits>
      <xsl:apply-templates select="//p/*/preceding-sibling::*[1]/*/text()"
        mode="select-fruit-name"/>
    </fruits>
  </xsl:template>
  <xsl:template match="node()" mode="select-fruit-name">
    <xsl:variable name="var-fruit-name" select="."/>
    <xsl:apply-templates mode="select-features"
      select="parent::*/parent::*/following-sibling::*[position()=1]"
      with-param name="var-fruit-name" select="$var-fruit-name"/>
  </xsl:template>
  <xsl:template match="node()" mode="select-features">
    <xsl:param name="var-fruit-name"/>
    <xsl:variable name="var-features" select="."/>
    <fruit>
      <name>
        <xsl:value-of select="normalize-space($var-fruit-name)"/>
      </name>
      <features>
        <xsl:apply-templates select="$var-features//tr/*/preceding-sibling::*[1]/text()"
          mode="select-feature-name"/>
      </features>
    </fruit>
  </xsl:template>
  <xsl:template match="node()" mode="select-feature-name">
    <xsl:variable name="var-feature-name" select="."/>
    <xsl:apply-templates mode="select-feature-value"
      select="parent::*/following-sibling::*[position()=1]/text()"
      with-param name="var-feature-name" select="$var-feature-name"/>
  </xsl:template>
  <xsl:template match="node()" mode="select-feature-value">
    <xsl:param name="var-feature-name"/>
    <xsl:variable name="var-feature-value" select="."/>
    <feature>
      <xsl:value-of select="$var-feature-value"/>
    </feature>
  </xsl:template>
</xsl:stylesheet>
```

continued on following page
Using Logic Programming and XML Technologies for Data Extraction from Web Pages

Box 10. continued

```
<name>
  <xsl:value-of select="normalize-space($var-feature-name)"/>
</name>
[value>
  <xsl:value-of select="normalize-space($var-feature-value)"/>
</value>
</feature>
</xsl:template>
</xsl:stylesheet>

and it produces the following output when applied to the fruits example.

```xml
<?xml version="1.0" encoding="utf-8"?>
<fruits>
  <fruit>
    <name>Red apple</name>
    <features>
      <feature>
        <name>weight</name>
        <value>120</value>
      </feature>
      <feature>
        <name>color</name>
        <value>red</value>
      </feature>
      <feature>
        <name>diameter</name>
        <value>8</value>
      </feature>
    </features>
  </fruit>
  <fruit>
    <name>Lemon</name>
    <features>
      <feature>
        <name>weight</name>
        <value>70</value>
      </feature>
      <feature>
        <name>color</name>
        <value>yellow</value>
      </feature>
    </features>
  </fruit>
</fruits>
```

continued on following page


