Visual Web Information Extraction with Lixto*

Robert Baumgartner
DBAI, TU Wien
Favoritenstr. 9
1040 Vienna
Austria
baumgart@dbai.tuwien.ac.at

Sergio Flesca
DEIS, Università della Calabria
Via Pietro Bucci, 41C-42C
87030 Rende (CS)
Italy
flesca@deis.unical.it

Georg Gottlob
DBAI, TU Wien
Favoritenstr. 9
1040 Vienna
Austria
gottlob@dbai.tuwien.ac.at

Abstract

We present new techniques for supervised wrapper generation and automated web information extraction, and a system called *Lixto* implementing these techniques. Our system can generate wrappers which translate relevant pieces of HTML pages into XML. *Lixto*, of which a working prototype has been implemented, assists the user to semi-automatically create wrapper programs by providing a fully visual and interactive user interface. In this convenient user-interface very expressive extraction programs can be created. Internally, this functionality is reflected by the new logic-based declarative language *Elog*. Users never have to deal with *Elog* and even familiarity with HTML is not required. *Lixto* can be used to create an "XML-Companion" for an HTML web page with changing content, containing the continually updated XML translation of the relevant information.

1 Introduction and Motivation

Nowadays web content is mainly formatted in HTML. This is not expected to change soon, even if more flexible languages such as XML are attracting a lot of attention. While both HTML and XML are languages for representing semistructured data, the first is mainly presentation-oriented and is not really suited for database applications. XML, on the other hand, separates data structure from layout and provides a much more suitable data representation (cf. e.g. [1, 16]). A set of XML documents can be regarded as a database and can be directly processed by a database application or queried via one of the new query languages for XML, such as XML-GL [7], XML-QL [10] and XQuery [8]. As the following example shows, the lack of accessibility of HTML data for querying has dramatic consequences on the time and cost spent to retrieve relevant information from web pages.

Imagine you would like to monitor interesting *eBay* offers (`www.ebay.com`) of notebooks, where an interesting offer is, for example, defined by an auction item which contains the word "notebook", has current value between GBP 1500 and 3000 and which has received at least three bids so far. The *eBay* site does not offer the possibility to formulate such complex queries. Similar sites do not even give restricted query possibilities and leave you with a large number of result records organised in a huge table split over many web pages. You have to wade through all these records manually, because of no possibility to further restrict the result. Another drawback is that you cannot directly collect information of different auction sites (e.g. *onetwosold* and *ebay* items together) into a single structured file, a difficult task of web information integration due to very different presentation on each site.

The solution is thus to use *wrapper* technology to extract the relevant information from HTML documents and translate it into XML which can be easily queried or further processed. Based on a new method of identifying and extracting relevant parts of HTML documents and translating them to XML format, we designed and implemented the efficient wrapper generation tool *Lixto*, which is particularly well-suited for building HTML/XML wrappers and introduces new ideas and programming language concepts for wrapper generation. Once a wrapper is built, it can be applied automatically to continually extract relevant information from a permanently changing web page.

The *Lixto* method and system fulfills the require-

*All new methods and algorithms presented in this paper are covered by a pending patent. Future developments of Lixto will be reported at `www.lixto.com`.

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

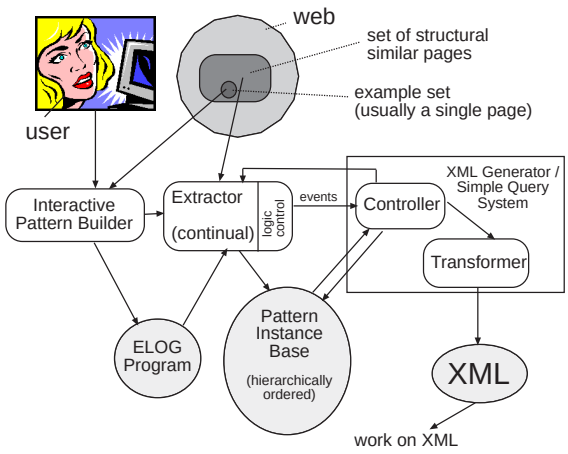

Figure 1: Overview of the implemented *Lixto* System

ments specified in a very recent paper on e-commerce tools [23]: "These tools must be targeted at typical, non-technical content managers. In order to be usable, the tools must be graphical and interactive, so that content managers see data as it is being mapped." *Lixto*'s distinctive features are summarised in the following. *Lixto* is easy to learn and use because a fully visual and interactive user interface is provided. Neither manual fine-tuning nor knowledge of the internal language is necessary. *Lixto* uses straightforward region marking and selection procedures that allow even those users not familiar with HTML to work with the wrapper generator. *Lixto* lets a wrapper designer work directly and solely on browser-displayed example pages, unlike other tools (see Section 6), that force the designer to work with other document views such as, e.g., table-views of the document or displayed HTML parse trees, or even HTML sources. After selecting example targets in the browser display, *Lixto* responds with highlighted targets in the same display (see Section 3). With *Lixto*, very expressive visual wrapper generation is possible: It allows for extraction of target patterns based on surrounding landmarks, on the contents itself, on HTML attributes, on the order of appearance and on semantic and syntactic concepts. Extraction is not limited to tokens of some document object model, but also possible from flat strings. Multiple and single targets are treated in a uniform way. *Lixto* even allows for more advanced features such as disjunctive pattern definitions, crawling to other pages during extraction, recursive wrapping. Moreover, the extracted data structures do not have to strictly obey the input HTML structure. Preliminary results on representative web pages with using the current *Lixto* prototype show a good performance (see Section 5).

The above mentioned features are internally reflected by a declarative extraction language called *Elog* (see Section 4), which uses a datalog-like logical syntax and semantics. *Elog* is invisible to the user. It is ideally suited for representing and successively incrementing the knowledge about patterns described by users. This knowledge is generated in an interactive process consisting of successive narrowing (logical *and*) and broadening (logical *or*) steps. An *Elog* program is a collection of datalog-like rules containing special extraction conditions in their bodies. *Elog* is flexible, intuitive and easily extensible.

This paper is structured as follows. In the next section the system architecture is described, in Section 3 we give an overview of the the interactive pattern generation and visual UI, whereas Section 4 is devoted to the theory of the *Elog* extraction language. Section 5 presents empirical results of using the *Lixto* wrapper generator, Section 6 discusses related approaches and Section 7 highlights future research directions.

2 Architecture and Implementation

A working prototype of *Lixto* already has been implemented with Java using Swing, OroMatcher [22] and JDOM [19]. The *Lixto* Toolkit (Figure 1) consists of the following modules:

The *Interactive Pattern Builder* provides the visual UI that allows a user to specify the desired extraction patterns and the basic algorithm for creating a corresponding *Elog* wrapper as output.

The *Extractor* is the *Elog* program interpreter that performs the actual extraction based on a given *Elog* program. The extractor, provided with an HTML document and a previously constructed program, generates as its output a *pattern instance base*, a data structure encoding the extracted instances as hierarchically ordered trees and strings. One program as input of the extractor can be used for continual extraction on changing pages, or to extract from several HTML pages of similar structure.

With the controller of the *XML Generator*, the user chooses how to map extracted information to XML. Its transformer module performs the actual translation from the extracted pattern instance base to XML.

3 Interactive Wrapper Generation

3.1 Creating Wrappers

A *Lixto* wrapper is created interactively by creating *patterns* in a hierarchical order. For example, one can first define a pattern `<item>` and then define a subpattern `<price>`. The subpattern relationship in this case expresses that each extracted instance of `<price>` must occur within one instance of `<item>`. Pattern names act as default XML element names. Each pattern characterises one kind of information. The set of extracted instances of a pattern, which are either HTML elements, list of elements, or strings, depends on the current page. Each pattern is defined by one or more filters. A filter e.g. allows the system to identify a set of similar nodes of the HTML parse tree, for instance a set of items internally represented as `<td>`.

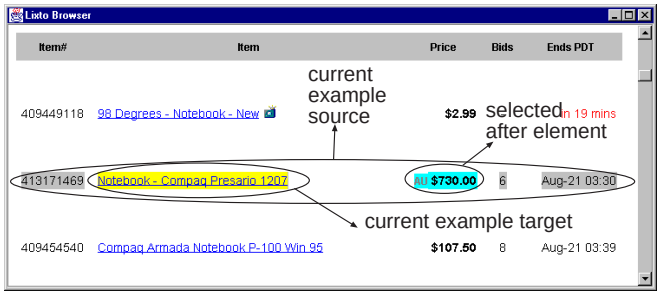

Figure 2: The *Lixto* Browser

A filter is created as follows: First, the user highlights with the mouse a representative instance of the desired target pattern directly on the example page. Internally, the system associates to this instance a generalised tree path in the HTML parse tree identifying similar instances and incorporates this as main goal of a *Elog* rule representing the filter (see Section 4). Second, the user adds restrictive conditions to the filter. These are reflected by the system as additional goals in the rule body describing this filter. The possible conditions, which will be explained in more detail, include: (a) **before/after conditions** that express that the target pattern instance must appear before or after some specific element. (b) **notbefore/notafter conditions** that express that some specific element must not be close to the target pattern. (c) **internal conditions** that express that some specific element must (not) appear inside the target pattern. (d) **range conditions** which, in case of multiple matchings, restrict the set of matched instances to a subinterval.

Adding a filter to a pattern extends the set of extracted targets, whereas imposing a condition to a filter restricts the set of targets. Alternately imposing conditions and adding new filters can perfectly characterise the desired information. The system creates *Elog* rules based on user-defined filters. The user is never concerned with the internal language *Elog*. The user interface is extremely simple and the entire wrapper construction process can be learned by an average user in very short time. The user is guided through a supervised pattern generation, and by simply marking relevant information items on-screen and visually setting constraints, filters and patterns are created. In [5], we describe an example program construction.

3.2 Pattern Creation Algorithm

The generation of a pattern is described in Fig. 3. The user can hierarchically define and refine patterns. She enters a pattern name, specifies the parent pattern S, selects by mouse clicks one example instance s of the parent pattern and marks (with the mouse) an element (e.g. one price) inside this instance on the sample page.

At the beginning, i.e. when facing a new HTML document (which is loaded into an internal browser; see Fig. 2) and having created a new program, the

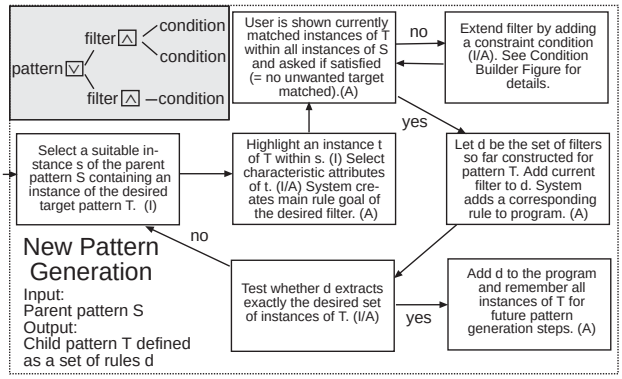

Figure 3: Generation of a new Pattern

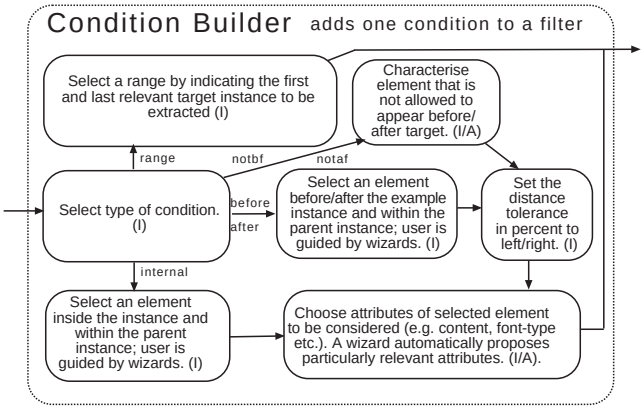

Figure 4: Adding Conditions

only pattern is `<document>` with a unique instance, the current example document. Fig. 3 distinguishes interactive (I) and automatic (A) steps and gives the logical pattern structure in its top-left corner. A pattern may consist of multiple filters. Each filter contains a number of conditions. An extracted instance must satisfy all conditions of at least one filter. Two consecutive mouse clicks on different parts of the current parent instance are interpreted in the best possible way to mark an HTML element of the document parse tree (cf. Fig. 7) or if not possible a list of elements.

The system generates a basic filter without conditions, but the user can already state some attribute requirements (the system constructs a suitable element path definition, see Section 4). Then it highlights all objects on the current example page that match these initial filter criteria (not only in the current pattern instance, but in all pattern instances). Sometimes a user wants a single match within one source, sometimes multiple matches – this makes no difference in the algorithm – it just depends on the definition of filters and conditions. E.g., if the user marks a table row, the system recognises the entity `<tr>` and highlights all table rows occurring at a comparable level in the document. At the same time the system constructs a general *Elog* rule for extracting table rows.

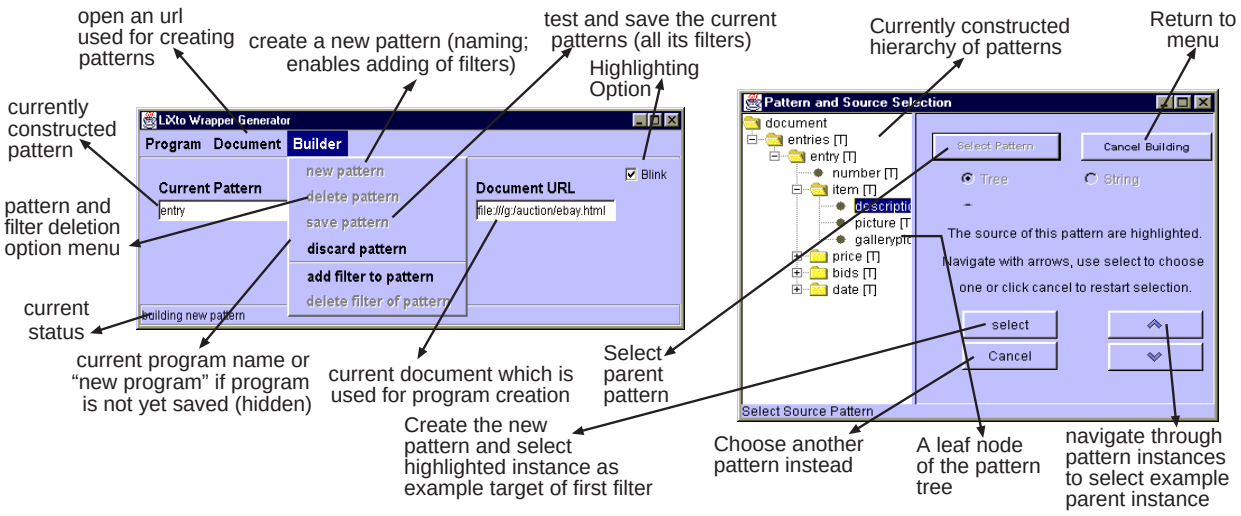

Figure 5: Main Menu and New Pattern Generation Menu of current prototype

If the user is satisfied with the elements identified by the system, she can confirm the pattern definition. Satisfaction, in this context, means that only desired targets are matched. Otherwise, if the concept is too general, then she can add restricting conditions (which are reflected by *Elog* condition predicates); cf. Fig. 4. For each such restriction, the system adds the corresponding condition atom to the *Elog* rule defining the filter at hand. Each filter is intended to extract a subset of the desired target set. If the current pattern is less general than intended by the user, another filter can be added, internally reflected by an additional *Elog* rule for the same pattern (several rules for the same pattern are interpreted disjunctively, as usual in Datalog). Different filters may be created based on labelling in different example parent pattern instances. By iterating restricting and generalising steps, it is usually possible to describe a desired pattern perfectly. Once a pattern has been defined, the user may use this pattern as parent for a new pattern. Recall that a detailed creation of a simple example wrapper is given in [5].

3.3 The Visual Interface

The current implementation includes visual tree pattern construction and the use of string patterns. All filter conditions discussed in this paper are supported. Moreover, the visual interface is assisted by an XML visualisation tool which at each instant shows the user the so far extracted XML code. A concept atom generator to create predefined concepts (such as "isCity", "isDate") based both on regular expressions and on reading some database tables is currently being added. Such concepts are especially useful to allow users to create string patterns without knowledge of regular expressions. Fig. 5 shows the main menu of *Lixto* (left-hand side). There, a new program can be created or an existing one loaded, new patterns can be added, the document for labelling can be chosen, etc. The same figure shows on its right hand side the source selection dialogue which enables the user to select at which node to create a new pattern. Fig. 2 shows the internal *Lixto* browser when selecting an after element. For each condition, an own interface is provided which uses the user-labelled information.

3.4 Translation into XML

The output by the extractor is well-suited for translation into XML. The interactive XML generator exploits the hierarchical structure of the pattern instance base and uses pattern names as default XML element names. The user can interactively choose the HTML attributes that appear in the XML output. Even more important is the possibility to decide which patterns are written to XML, possibly using auxiliary patterns. Fig. 9 displays the result of applying a (not illustrated) wrapper program onto the web page of Fig. 6.

4 Extraction Language/Mechanisms

4.1 A first glance at *Elog*

Elog is the system-internal datalog-like rule based language specifically designed for hierarchical and modular data extraction. A user of *Lixto* does not have to learn *Elog* and *never sees* the *Elog* program. *Elog* rules are the implementations of the visually defined filters and define elements to be extracted from web pages. Before we discuss the features of the language in detail, have a look at Fig. 8, in particular at the rule with head predicate $record(S, X)$. Observe that we use as in Prolog the same variables for each rule, and denote with "_" a variable in whose instantiations we are not interested. This predicate identifies records on an *eBay* page (each one is an own table). The first atom in the rule body specifies that the context S of the extraction, i.e. the so-called *parent pattern*, is an instance of `<tableseq>`. The second atom in the rule

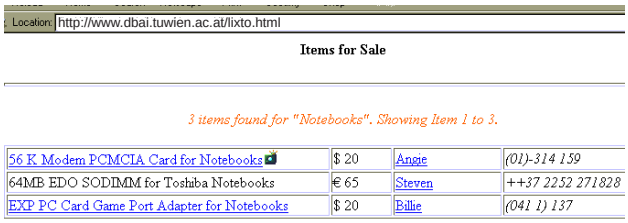

Figure 6: HTML Example Page

body looks for subelements that qualify as tables inside the unique `<tableseq>` instance and instantiates X with them. Given that the same *Elog* program can be applied to different web pages, the actual elements that an *Elog* program defines and extracts depend on the current web page. For this reason, we refer to the head predicates defined by an *Elog* program as *patterns*. Moreover, we denote a set of rules with the same head as pattern, too. The syntax and semantics of *Elog* and its predicates is explained below (only informally due to space constraints).

4.2 Document Model

Consider the example web page `lixto.html` of Fig. 6 and its parse tree as displayed in Fig. 7 based on the Java Swing parser. The values in brackets are the start and end-offsets (in characters) of the corresponding elements in the actual document. Additionally, we number nodes in a depth-first left-to-right fashion. Nodes of the HTML tree refer to elements which are represented as sets. The set contains pairs describing the association between attribute names and corresponding attribute values. E.g., the `<body>` element node of Fig. 7 is associated with {*(name,body), (bgcolor,FFFFFF), (elementtext,Items for ... ... 137)*} (whole document text). Fig. 7 highlights two other such attribute sets.

Observe that in our chosen document object model, several leaf elements are `<content>` elements – this parser treats tags such as `<b>` (bold-face) as attributes of an imaginary `<content>` element. We introduced a special attribute called "elementtext" for each element. This attribute reflects the contents of the element, which is in case of an internal node the left-to-right concatenation of the leaf elements below the internal node. In the following, we distinguish *tree regions* and *subtrees* of the HTML tree. A tree region is a region rooted at an internal node of the HTML tree where only the i-th up to the j-th child and their descendants are considered. Observe that a tree region is contiguous. A subtree is the tree rooted at one node of the HTML tree, i.e. *all* descendants are considered.

4.3 Extraction Mechanisms

Lixto offers two basic mechanisms of data extraction – tree and string extraction. For tree extraction, we identify elements with their corresponding tree paths

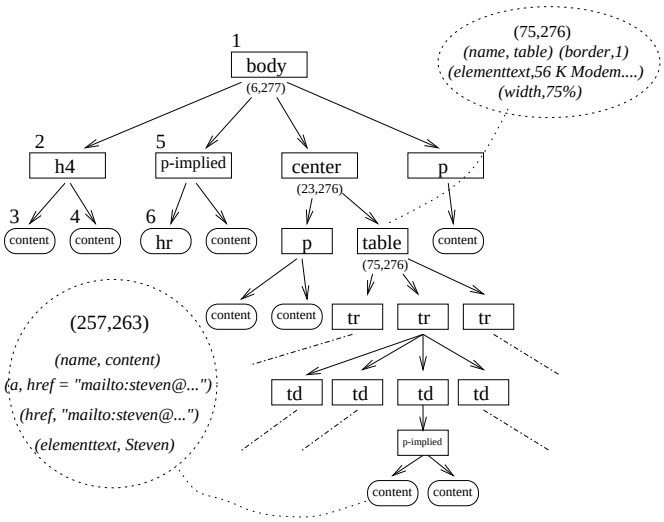

Figure 7: HTML Parse Tree of Example in Figure 6

and possibly some properties of the elements themselves. This does not necessarily identify a single element. As an example, $\star.table.\star.tr$ is a valid tree path. In the sample page page of Fig. 6, three elements are matched. The star acts as wildcard. The expression $.\star.x$ matches all paths to x which contain x as last element only. A plain tree path is a sequence of consecutive nodes in a subtree of an HTML tree. In an incompletely specified tree path stars may be used instead of element names. For simplicity, incompletely specified tree paths are referred to as *tree paths*. The semantics of a tree path applied to a tree region of an HTML page is defined as the *set of matched elements*.

Attribute Conditions are constraints reducing the number of matched elements. They pose requirements on occurring attributes and their values. An attribute condition is a triple specifying a required name, a required value (a string, or in case the third parameter is *regvar*, a regular expression possibly containing some variables indicated by $\backslash var$), and a special parameter *exact*, *substr* or *regvar*, indicating that the attribute value is exactly the required string, is a superstring of it, or matches the given regular expression, respectively. Instead of giving a formal definition, we illustrate this with an example: $(\star.hr, [(size, [3-4]*, regvar), (width, \%, substr)])$ identifies horizontal rules of size 3 or 4 with a width specified in percent. Each output variable, which is included in the second parameter must be used as input for a concept of the same rule (cf. Section 4.4).

An *element path definition epd* consists of a tree path and a set of attribute conditions. It is called simple if it consists of one element name only. The semantics of applying an element path definition to a tree region of an HTML tree is given as the *set of matched elements* of the corresponding tree path which moreover satisfy all of the attribute conditions. Instead of element path definitions, equivalently, XPath expres-

sions can be used (with some extensions, such as the possibility to express that an attribute value is a concept). To simplify presentation, however, we stick to our introduced notation.

The second extraction method relies on strings. In the HTML parse tree, strings are represented by the text of *content* leaves. However, we associate a string to *every* node of the parse tree available as the value of the attribute *elementtext*. For instance when extracting access codes of the phone-numbers of `lixto.html`, string extraction has to be used. A substring of the elementtext of an HTML tree is denoted as *string source*. One can express that a string source must match a given regular expression. A *string path definition spd* is a regular expression possibly containing some variables (variable Y indicated by $\backslash var[Y]$) which appear in some concept predicate of the corresponding rule. Regular expressions are powerful tools for text processing and matching. Refer to [22] for a Java regular expression library. Extraction generates minimal non-overlapping substrings. The final two patterns of Fig. 8 give an example of string extraction. An *attribute path definition apd* helps to extract values of attributes. It is simply a string (expressing the attribute name).

4.4 Language Definition

Elog atoms correspond to special predicates with a well-defined semantics. They operate on source objects (tree regions and string sources), path definition objects and numerical arguments and obey binding conventions. In a datalog-like language, the function mapping a given source S to a set of elements matching an *epd* is treated as relation $subelem(S, epd, X)$. $subelem(s, epd, x)$ evaluates to true iff s is a tree region, *epd* is an element path definition and x is a tree region contained in s where the root of x matches *epd*. Note that the tree path specified in a tree extraction definition predicate is always relative to the parent-pattern instance.

Extraction definition predicates specify a set of extraction instances. One of these is *subelem*. As far as string extraction is concerned, the predicate $subtext(S, spd, X)$ is used. There, S is either a tree region or a string source, and X a string source. Two more extraction definition predicates are built-in. (1) $subsq(S, epd, fpd, lpd, X)$: If s and x are tree regions, *epd* is an element path definition, and *fpd* and *lpd* are simple element path definitions, $subsq(s, epd, fpd, lpd, x)$ evaluates to true iff the root of x satisfies *epd*, its first child satisfies *fpd* and its last one *lpd*. (2) $subatt(S, apd, X)$: If s is a tree region, x a string source and *att* is an attribute path definition of the root element of s, then $subatt(s, apd, x)$ evaluates to true iff x is the value of *apd*. *subatt* gives the possibility to extract the values of attributes.

Context condition predicates specify that some other subtree or text must (not) appear before or after the desired extraction target. For example, on a page with several tables, the final table could be identified by an external condition stating that no table appears after the desired table. *Before* predicates are explained here, *after* predicates work analogously. (1) $before(S, X, epd, b, e, Y, P)$: If s and x are tree regions, then $before(s, x, epd, b, e, y, p)$ evaluates to true iff y is a subtree whose root node is matched by *epd* and the end offset of y precedes the start offset of x within relative distance p where $b \leq p \leq e$. (2) $notbefore(S, X, epd, d)$: If s and x are tree regions, then $notbefore(s, x, epd, d)$ evaluates to true iff no element satisfying *epd* precedes x within relative distance d. The same predicates are defined for string extraction: There, S is an arbitrary source, X is required to be a string source, *spd* is used instead of *epd* and instead of the root node simply the string itself is used. The percentual distance values b and e define the tolerance interval where the element is allowed to occur inside the current parent-pattern instance. Additionally, a condition predicate may contain new variables Y and P, which can be referred by other conditions. To express that an element occurs anywhere within the parent instance and before the target (or a condition output), the distance values are set to 0 and 100, respectively.

Internal conditions predicates impose conditions on the internal structure. Imagine, for instance, one wants to extract all tables containing somewhere a word typeset in italics. This can be obtained by adding a *contains* condition. $contains(X, epd, Y)$: $contains(x, epd, y)$ evaluates to true iff x is a tree region (string source) containing a subtree (string source) y where the root element of y matches *epd* (where y matches *spd*). The *firstsubtree* condition is a kind of "startswith" condition that states that the first subtree of a tree region should contain a particular element. $firstsubtree(X, Y)$: $firstsubtree(x, y)$ evaluates to true iff y is the subtree rooted at the first child of the tree region x. *lastsubtree* is defined analogously.

Concept condition predicates are semantic concepts like $isCountry(X)$ or $isCurrency(X)$ (see Fig. 8) or syntactic ones like $isDate(X)$ (or $isDate(X, Y)$ where the output Y returns a standard date format), stating that a string X represents a date, a country, or a currency, respectively. Some predicates are built-in to enrich the system, however more concepts can be interactively added. Syntactic predicates are created as regular expressions, whereas semantic ones refer to an ontological database. Moreover, **Comparison Conditions** such as $<(X, Y)$ allow comparison of concepts such as two standard format dates.

Pattern predicates indicate that a source belongs to a particular pattern and refers to a particular parent pattern-instance. They are used in the head, and in the rule body for referring to a parent pattern and for further pattern references. As an example, the `<price>` pattern can be constructed by using the ele-

```
tablesq(S,X)   ← document("www.ebay.com/",S), subsq(S,(.body,[]),(.table,[]),(.table,[]),X),
                 before(S,X,(.table,[(elementtext,item,substr])),0,0,_,_), after(S,X,.hr,0,0,_,_)
 record(S,X)   ← tableseq(_,S), subelem(S,.table,X)
itemnum(S,X)   ← record(_,S), subelem(S,⋆.td,X), notbefore(S,X,.td,100)
itemdes(S,X)   ← record(_,S), subelem(S,(⋆.td.⋆.content,[(a,,substr)],X)
  price(S,X)   ← record(_,S), subelem(S,(⋆.td,[(elementtext,\var[Y].∗,regvar)]),X), isCurrency(Y)
   bids(S,X)   ← record(_,S), subelem(S,⋆.td,X), before(S,X,.td,0,30,Y,_), price(_,Y)
currency(S,X)  ← price(_,S), subtext(S,\var[Y],X), isCurrency(Y)
 pricewc(S,X)  ← price(_,S), subtext(S,[0−9]+\.[0−9]+,X)
```

Figure 8: *Elog* Extraction Program for Information on eBay

ment path definition $.\star.td$, and imposing the constraint that immediately before, a target of pattern `<item>` needs to occur: $before(S, X, .\star.td, 0, 1, Y, \_), item(\_, Y)$.

Range Conditions restrict the matched targets depending on their order of appearance. To any rule, a range condition such as "[3,7]" can be added, indicating that only the third up to the seventh matched instance within each parent instance are matched.

4.5 Elog Extraction Programs

A standard extraction rule looks as follows: $New(S, X) \leftarrow Par(\_, S), Ex(S, X), Co(S, X, \dots)[a, b]$, where S is the parent instance variable, X is the pattern instance variable, $Ex(S, X)$ is an extraction definition atom, and the optional $Co(S, X)$ are further imposed conditions. A tree (string) extraction rule uses a tree (string) extraction definition atom and possibly some tree (string) conditions and general conditions. The numbers a and b are optional and serve as range parameters. *New* and *Par* are pattern predicates referring to the parent pattern and defining the new pattern, respectively. The above standard rule reflects the principle of aggregation. In an extended environment, we moreover allow *specialisation rules* such as: $greentable(S, X) \leftarrow table(S, X), contains(X, (.td, [color, green, exact]), \_)$. Additionally, an extended environment contains document filters, using a *getDocument(S,X)* atom, where S is a string source representing an URL, and X the web page the URL points to. With such filters, one can crawl to further documents. If document filters are used, each program has an initial filter using the *getDocument* atom with user-specified input.

The semantics of a rule is given as the set of matched targets x: A substitution s, x for S and X evaluates $New(s, x)$ to true if all atoms of the body are true for this substitution. Only those targets are extracted for which the head of the rule resolves to true. Moreover, if the extraction definition predicate is a subsequence predicate, only minimal rule outputs are matched (i.e. instances that do not contain any other instances). Observe that range criteria are applied after non-minimal targets have been sorted out.

A *pattern* is a set of extraction rules defining the same head and referring to the same parent pattern. In the visual pattern generation the user first enters a pattern name and to which parent pattern the pattern belongs. All rules created inside the pattern use this information. We distinguish tree and string patterns. To the first, only tree extraction rules can be asserted, to the second one only string extraction rules. The root pattern `<document>` is a special pattern without filters. If using document filters to crawl to further web pages, document patterns are used as third pattern type (and an initial document filter is used). Parents of tree patterns are either tree or document patterns, parent of string patterns are tree or string patterns, and parent of document patterns are string patterns. A pattern acts like a disjunction of rule bodies: To be an extracted instance of a pattern, a target needs to be in the solution set of at least one rule. The pattern output additionally obeys a minimality criterion. In patterns, even in those consisting of a single rule, overlapping targets may occur.

An extraction program P is a set of patterns. *Elog* program evaluation differs from Datalog evaluation in the following three aspects: built-in predicates, various kinds of minimisation, and use of range conditions. Moreover, the atoms are not evaluated over an extensional database of facts representing a web page, but directly over the parse tree of the web page. Applying a program to an HTML page creates a set of hierarchically ordered tree regions and string sources (called a *pattern instance base*) by applying all patterns of the program in their hierarchical order to this HTML document (and possibly to further HTML documents if document filters are used). Each pattern produces a set of instances. Each pattern instance contains a reference to its parent instance. As patterns are ordered in a strictly hierarchical way, the program is hierarchically stratified. In the final section we will relax the definition of patterns to create recursive programs.

As example program consider a wrapper for *eBay* pages (Fig. 8). On *eBay* pages, every offered item is stored in its own table extracted by `<record>`; further patterns are all defined within such a record. The pat-

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
 <heading>Items for Sale</heading>
 <description>3 items found for "Notebooks".
        Showing Item 1 to 3.</description>
 <entry>
    <article>56 K Modem PCMCIA Card for
        Notebooks</article>
    <price>$ 20</price>
    <person href="mailto:itsme@bestseller.org">
        Angie</person>
    <phone>(01)-314 159</phone>
    <picture/>
 </entry>
[...]
```

Figure 9: XML translation of lixto.html

tern `<price>` uses a concept attribute, namely *isCurrency* – which matches strings like $, DM, Euro, etc. The `<bids>` pattern uses a reference to the `<price>` pattern. The final two patterns are string patterns.

5 Testing the *Lixto* Tool

We chose twelve example sites (Table 1), some of which were already used for testing purposes by other wrapper generators. Several users of whom not all are familiar with details of HTML contributed to our test results. Initially, we asked them to create a wrapper based on a single example page. Table 2 summarises answers to the following questions: (1) Is it possible to wrap this page with *Lixto*? (2) How "complex" is the constructed program for this site? (ratio of required predicates to used output patterns) (3) What is the percentage of correctly wrapped pattern instances of a number of randomly chosen similarly structured testpages with a wrapper written on one example page only. (4) How many example pages are necessary (due to structural deviations) to get 100 percent of correctly matched pattern instances? (5) Moreover, we specify the time needed for constructing the initial wrapper based on one example page. Additionally, the time for constructing one output pattern is computed to gain a measure how much "thinking time" was required for each output pattern. (6) In the last row the depth of the pattern tree is specified.

Let us describe some more details: On *eBay*, the initial wrapper worked well on almost all test pages like queries on cars, football, etc. However, one filter rule of `<date>` required that dates must contain a colon and a dash. This matched one item description, too, which used both. Hence, the pattern had to be refined based upon the knowledge of this second page to match 100% of the patterns of all example pages. For the *CIA Factbook*, the user chose a bad example page with only one bordering country. Even after improving the wrapper to deal with comma-separated counries, Albania had to be treated in a special way. The wrapper for *DBLP* relies on a number of intermediate auxiliary patterns, indicated by the high nesting depth of the document. For the *CNN* pages of the US election results per state, a wrapper just extracting names of president candidates and the received votes was written in a few minutes; due to a very homogeneous structure, one example page was sufficient to extract these data for all states. The *Jobs Jobs Jobs* site is the only example where the number of needed sample pages depends on the number of testpages due to a wide variety of structures for job offers. For the *Perl Module List* we are merely interested in writing a wrapper for a single web page. This list uses mainly preformatted text, hence the program heavily relies on string extraction. In the current implementation some auxiliary patterns are needed, and some clever constructions to obtain a 100% match for the five chosen patterns (module group, leaf patterns name, DSLI, description, info). We conclude that almost all web pages can be visually wrapped with *Lixto*. For none of the test pages the user had to modify the *Elog* program manually. Wrapper construction is usually very fast. The program length measured in used predicates is never unreasonably large compared to the output patterns (ranging from 1.78 to 4.4). The user never had to consider more than three example pages to get a 100% match for all testpages.

6 Related Work

First, we give an overview of approaches less related to *Lixto* because they do not provide visual support. *Stand-alone wrapper programming languages* include *Florid* [18] (using a logic-programming formalism), *Pillow* [6] (an HTML/XML programming library for logic programming systems), *Jedi* [13] (using attributed grammars), *Tsimmis* and *Araneus*. In *Tsimmis* [11], the extraction process is based on a procedural program which skips to the required information, allows temporary storages, split and case statements, and to follow links. However, the wrapper output has to obey the document structure. In *Araneus* [3], a user can create relational views from web pages by computationally fast and advanced text extracting and restructuring formalisms, in particular using procedural "Cut and Paste" exception handling inside regular grammars. In general, all manual wrapper generation languages are difficult to use by laypersons.

Machine learning approaches rely on learning from examples and counterexamples of a large number of web pages. *Stalker* [20] specialises general *SkipTo* sequence patterns based on labelled HTML pages. An approach to maximise specific patterns is introduced by Davulcu et al. [9]. Other examples include *Softmealy* [12] (using finite-state transducers) and *MIA* [24] (prolog-based wrappers using anti-unification; neural networks to generalise and learn texts). *NoDoSe* ([2]) extracts information from plain string sources and provides a user interface for example labelling. It has restricted capabilities to deal with

Name	*Website*	*Used Example Page*	*Testpages*
Amazon	http://www.amazon.com/	Lord of the Rings	10
CIA Factbook	www.odci.gov/cia/publications/factbook/	United Kingdom	12
Cinemachine	www.cinemachine.com/	The World is not enough	15
DBLP	www.informatik.uni-trier.de/~ley/db/	Michael Ley	10
Election Res. / State	www.cnn.com/ELECTION/2000/results/	Alabama	50
eBay	www.ebay.com/	query on Notebooks	20
Excite Weather	www.excite.com/weather/forecast	London (UK)	12
Jobs-Jobs-Jobs	www.jobsjobsjobs.com/	23370	10
Perl Module List	www.cpan.org/modules/00modlist.long.html	single huge page	ex.pg.
Travelnotes	www.travelnotes.org/	query on Istanbul	10
Yahoo People Email	people.yahoo.com/	query on Mayer	15
Yahoo Weather	weather.yahoo.com/	Paris	15

Table 1: Some of the test-sites used for *Lixto*

Name	*wrapable?*	*Complexity*	*Correct*	*for 100%*	*Time/Pattern (mins)*	*Depth*
Amazon	yes	16/9 = 1.78	95%	3	22/9 = 2.44	4
CIA Factbook	yes	17/5 = 3.4	80%	3	18/5 = 3.6	3
Cinemachine	yes	6/4 = 1.5	100%	1	16/4 = 4	2
DBLP	yes	27/9 = 3	90%	2	54/9 = 6	8
Election Results / State	yes	4/2 = 2	100%	1	6/2 = 3	2
eBay	yes	19/8 = 2.38	99.9%	2	21/8 = 2.63	4
Excite Weather	yes	22/7 = 3.14	100%	1	30/7 = 4.3	3
Jobs Jobs Jobs	yes	21/12 = 1.75	90%	3	40/12 = 3.3	2
Perl Module List	yes	22/5 = 4.4	(100 %)	(1)	60/5 = 14	6
Travelnotes	yes	11/4 = 2.75	95%	2	20/4 = 5	2
Yahoo People Email	yes	10/3 = 3.3	100%	1	24/3 = 8	3
Yahoo Weather	yes	22/10 = 2.2	100%	1	12/10 = 1.2	3

Table 2: Evaluation of wrapper generation

HTML. Kushmerick et al. [15] create robust wrappers based on predefined extractors; their visual support tool *WIEN* receives a set of training pages, where the user can label relevant information and the system tries to learn a wrapper. Their approach does not use HTML parse trees. Kushmerick also contributed to the wrapper verification problem [14], an issue worth to explore w.r.t. *Elog*, too. In general, drawbacks of machine-learning approaches are limited expressive power and the large number of required example pages.

Supervised interactive wrapper generation tools include *W4F* [21] and *XWrap* [17]. *W4F* uses an SQL-like query language called HEL. Parts of the query can be generated using a visual extraction wizard which is limited to returning the full DOM tree path of an element. However, the full query must be programmed by the user manually. Hence, *W4F* requires expertise with both HEL and HTML. HEL requires tricky use of index variables and fork constructs to correctly describe a complex pattern structure. *XWrap* uses a procedural rule system and provides limited expressive power for pattern definition. The user cannot label regions in documents as flexible as in *Lixto*. *XWrap* lacks visual facilities for imposing external or internal conditions to a pattern, but instead is rather template-based. The division into two description levels and the automatic hierarchical structure extractor severely limit the ways to define extraction patterns (e.g. it is impossible to describe pattern disjunctions). Hence, in general, other supervised wrapper generation tools require manual postprocessing and do not offer the browser-displayed document for labelling.

7 Current/Future Work

It is currently already possible to write and execute *Elog* programs that can crawl to other pages, i.e. follow links during extraction, and can recursively wrap linked sequences of web pages. For such applications, the pattern structure does no longer form a tree because filters of one pattern definition may refer to different parent patterns (in a similar fashion as recursive data types). For example, recursive *Elog* programs may follow a "next" button and navigate to further pages during extracting, while extracting instances of the same patterns. See Fig. 10 for extending the *eBay* example of Fig. 8 to follow a "next" button, and extract for each page the same kind of information. In this example, the pattern `<document>` has an initial filter which uses the user-provided page ($1), and an additional filter, which uses `<nexturl>` as parent pattern (whose instances are strings representing URLs).

```
    next(S,X)  ←  document(_,S),subelem(S,(⋆.content,[(a,,substr),(elementtext,Next,exact)]),X)
 nexturl(S,X)  ←  next(_,S),subatt(S,href,X)
document(S,X)  ←  getDocument($1,X)
document(S,X)  ←  nexturl(_,S),getDocument(S,X)
```

Figure 10: Recursive Extension of the *Elog* program of Figure 8

Web crawling and recursion in *Lixto* is described in more detail in [4]. Currently we are extending the interactive pattern builder to cover these aspects. Furthermore, a server-based *Lixto* version is currently being implemented – it uses simple web interfaces and works in the user's favourite browser. Future work focuses on automation heuristics for optional use, including to work on multiple example targets at once. Additionally, *Lixto* wrappers will be embedded into a personalisable information channel system.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web - From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[2] B. Adelberg. NoDoSE - a tool for semi-automatically extracting semi-structured data from text documents. In *Proc. SIGMOD*, 1998.

[3] P. Atzeni and G. Mecca. Cut and paste. In *Proc. PODS*, 1997.

[4] R. Baumgartner, S. Flesca, and G. Gottlob. Declarative information extraction, web crawling and recursive wrapping with Lixto. To appear in *Proc. LPNMR*, 2001.

[5] R. Baumgartner, S. Flesca, and G. Gottlob. Supervised wrapper generation with Lixto. To appear in *Proc. VLDB Demo*, 2001.

[6] D. Cabeza and M. Hermenegildo. Distributed WWW programming using (Ciao-)Prolog and the PiLLoW library. *TPLP*, 1(3), 2001.

[7] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a graphical query language for querying and restructuring XML documents. In *Proc. WWW Conf.*, 1999.

[8] D. Chamberlin and al. (Eds.). XQuery: A query language for XML. http://www.w3.org, 2001.

[9] H. Davulcu, G. Yang, M. Kifer, and I.V. Ramakrishnan. Computat. aspects of resilient data extract. from semistr. sources. In *Proc. PODS*, 2000.

[10] D. Florescu, A. Deutsch, A. Levy, D. Suciu, and M. Fernández. A query language for XML. In *Proc. 8th Intern. WWW Conference*, 1999.

[11] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the web. In *Proc. Workshop on Mang. of Semistructured Data*, 1997.

[12] C-N. Hsu and M.T. Dung. Generating finite-state transducers for semistructured data extraction from the web. *Information Syst.*, 23/8, 1998.

[13] G. Huck, P. Fankhauser, K. Aberer, and E.J. Neuhold. JEDI: Extracting and synthesizing information from the web. In *Proc. COOPIS, IEEE CS Press*, 1998.

[14] N. Kushmerick. Wrapper verification. *World Wide Web Journal*, 2000.

[15] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proc. IJCAI*, 1997.

[16] A.Y. Levy and D.S. Weld. Intelligent internet systems. *Artificial Intelligence*, 118(1-2), 2000.

[17] L. Liu, C. Pu, and W. Han. XWrap: An extensible wrapper construction system for internet information. In *Proc. ICDE*, 2000.

[18] W. May, R. Himmeröder, G. Lausen, and B. Ludäscher. A unified framework for wrapping, mediating and restructuring information from the web. In *WWWCM*. Sprg. LNCS 1727, 1999.

[19] B. McLaughlin and J. Hunter. jdom.org Package. http://www.jdom.org/.

[20] I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In *Proc. 3rd Intern. Conf. on Autonomous Agents*, 1999.

[21] A. Sahuguet and F. Azavant. Building light-weight wrappers for legacy web data-sources using W4F. In *Proc. VLDB*, 1999.

[22] D.F. Savarese. OROmatcher - Regular Expressions for Java. http://www.savarese.org/oro/.

[23] M. Stonebraker and J. Hellerstein. Content integration for e-business. In *Proc. Sigmod*, 2001.

[24] B. Thomas. Anti-unification based learning of T-wrappers for information extraction. In *Workshop on Machine Learning for IE*, 1999.