Declarative Information Extraction, Web Crawling, and Recursive Wrapping with Lixto*

Robert Baumgartner[1], Sergio Flesca[2], and Georg Gottlob[1]

[1] DBAI, TU Wien, Vienna, Austria
{baumgart,gottlob}@dbai.tuwien.ac.at
[2] DEIS, Università della Calabria, Rende (CS), Italy
flesca@si.deis.unical.it

Abstract. *Lixto* is a system and method for the visual and interactive generation of wrappers for Web pages under the supervision of a human developer, for automatically extracting information from Web pages using such wrappers, and for translating the extracted content into XML. This paper describes some advanced features of *Lixto*, such as disjunctive pattern definitions, specialization rules, and *Lixto*'s capability of collecting and aggregating information from several linked Web pages.

1 Introduction and Motivation

Extracting relevant information automatically from HTML Web pages of changing content, and converting the extracted information to a structured representation is an important problem, to which a lot of research has been dedicated [3, 7, 8, 10, 11, 13, 14]. XML was designed to enrich the semantics of Web information [1, 6]. Even if in some respects XML may not yet fulfill this goal perfectly, XML appears to be the right representation format for the information extracted from HTML. Programs that perform such extraction and translation tasks are referred to as *wrappers*. Wrappers can be hand-coded, e.g. in specialized languages such as Jedi [9] or Florid [12], or they can be produced via *wrapper generators*. Wrapper generators are software tools that generate wrappers via induction (such as e.g. [2, 10, 13]) or that semi-automatically support the generation of wrappers via an interactive process supervised by a human designer ([11, 14]). Wrapper generators support the task of reverse engineering, as the goal of a wrapper is to reverse the processing of dynamic Web sites that generate HTML starting from an internal structured representation (such as a relational database).

In a recent paper [5] we introduced *Lixto*, a new method and system for visually generating HTML/XML wrappers under the supervision of a human designer. *Lixto* allows a wrapper designer to interactively and visually define information extraction patterns on the base of visualized sample Web pages. These extraction patterns are collected into a hierarchical knowledge base that

* All new methods and algorithms of the *Lixto* system are covered by a pending patent. Future developments of *Lixto* will be reported at www.lixto.com.

constitutes a declarative wrapper program. The extraction knowledge is internally represented in a datalog like special-purpose logic programming language, called *Elog*. However, a user of *Lixto* is not concerned with the syntax of *Elog* and does not need to learn this language as she constructs an *Elog* wrapper program by purely visual and interactive primitives without ever seeing the resulting *Elog* program. Wrapper programs in *Elog* can be directly executed over input Web sites by an extractor module that interprets the *Elog* rules taking care of the evaluation of special built-in predicates. *Lixto* also allows a designer to define XML translation rules that specify how extracted content should be translated into XML, a so-called *XML translation scheme*. An XML translation scheme together with extraction pattern definitions (the *Elog* program) in addition enables the system to construct a Document Type Definition (DTD) which describes the characteristics of the output XML documents.

The advantages of the *Lixto* wrapper generator over competing approaches are mainly the following. *(1) Very high expressive power*, i.e., an unprecedented capability of defining sophisticated extraction patterns. *(2) Excellent visual support*: The wrapper designer's sole view of an example HTML document is the browser-displayed standard image of the document (no annotations, overlays, HTML-sources or DOM trees) and the wrapper designer uses directly this display for marking extraction patterns. *(3) Good learnability*, because no extraction language needs to be learned and neither HTML nor XML knowledge is necessary. *(4) Sample parsimony*, which means that very few sample pages (in most cases a single one) are needed in order to define robust wrappers for large classes of Web pages. A *(5) simple and smooth XML translation mechanism* that gives a designer several options for formatting or modifying the XML output.

Basic features of *Lixto* are described in [4, 5], where also a comparison to related research is given. The main goal of the present paper is to introduce and illustrate some of the more advanced features of the *Elog* language. All the presented advanced features can be visually created by using *Lixto* without knowing *Elog*. Details of the visual interface and the way of creating patterns can be found in [4] and [5], where a precise description of the pattern generation algorithm is given. There, these details are discussed for a restricted environment w.r.t. some advanced concepts discussed in this paper, but a quite similar approach can be used for these advanced features. The present paper is self-contained at the level of general description, but not at the level of details. For the latter, we refer to [5].

Among the advanced features we discuss here are *disjunctive wrapping*, i.e., defining one pattern through several alternative definitions; *pattern specialization*, i.e., defining a new pattern by restricting another pattern; interactively defining new *document patterns*, which are patterns corresponding to entire documents that are identified via extracted URLs; *Web crawling*, which, in this context, means that a pattern hierarchy is built that aggregates information from various Web pages by starting at a given input page and automatically following URLs to other pages; and *recursive wrapping* which means that recursive pattern structures (akin to recursive data types) can be constructed that allow the system to crawl to an indefinite number of Web pages and extract information from all these pages. We will also discuss some interesting *nonmonotonic issues* such as pattern minimization principles and the semantics of range restrictions. Moreover, this paper introduces *pattern graphs* for describing the structure of the pattern hierarchy interactively defined by a designer (see Figures 3,4,6, and 7). Note that pattern graphs for simple extraction tasks are trees, which means that there is a strict pattern hierarchy. When disjunctive pattern definitions are used, then the corresponding pattern graphs are dags, while with recursive wrapping they are cyclic graphs.

The paper is structured as follows. In the next two sections we give an overview of *Lixto* and a description of the basic features of the *Elog* language. Section 4 gives a closer look on some features. In Section 5 we illustrate the power of disjunctive pattern descriptions, whereas in Section 6 some light is shed on *Elog*'s aspects concerning link crawling and recursion. These sections introduce advanced features of the internal language of *Lixto* both with an abstract description and examples from the commercial domain. Section 7 discusses various nonmonotonic aspects of *Lixto* such as minimization, range conditions, and further recursive aspects introduced by pattern references.

2 Pattern Generation with *Lixto*

Architecture. The *Lixto* prototype consists of two main blocks: The *Wrapper Generator* and the *Program Evaluator*. One module of the wrapper generator, the *Interactive Pattern Builder*, allows a wrapper designer to create and to store a wrapper in form of an extraction program (a program in the language *Elog*). Moreover, the wrapper generator contains the *XML Translation Builder* that allows a designer to specify how extracted data should be translated into XML format and to store such a specification in form of an XML translation scheme. The program evaluator automatically executes an extraction program (performed by the *Extractor* module) and a corresponding XML translation scheme (performed by the *XML translator* module) over Web pages by extracting data from them and translating the extracted data into XML format. (For details see [5].)

Extraction Patterns. A wrapper is constructed by formalizing, collecting, and storing the knowledge about desired extraction patterns. Extraction patterns describe single data items or chunks of coherent data to be extracted from Web pages by their locations and by their characteristic internal or contextual properties. Extraction patterns are generated and refined interactively and semi-automatically with help of a human wrapper designer. They are constructed in a hierarchical fashion on sample pages by marking relevant items or regions via mouse clicks or similar actions, by menu selections, and/or by simple textual inputs to the user interface. A wrapper, in our approach, is thus a knowledge base consisting of a set of extraction patterns.

While patterns are descriptions of data to be extracted, pattern instances are concrete data elements on Web pages that match such descriptions, and hence are extracted. *Lixto* distinguishes different types of patterns: Tree, string, and document patterns. Tree patterns serve to extract parts of documents corresponding to tree regions, i.e., to subtrees of their parse tree. String patterns serve to extract textual strings from visible and invisible parts of a document (an invisible part could be, e.g., an attribute value such as the name of an image). Document patterns are used for navigating to further Web pages.

Logical Organization of Patterns. The logical organization of an extraction pattern is as follows: each extraction pattern has a name and contains one or more so-called *filters*. Each filter provides an alternative definition of data to be extracted and to be associated with the pattern. The set of filters of a pattern is interpreted disjunctively (i.e., connected by logical ORs). Each filter is associated to a parent pattern from which it extracts the desired information. Tree (string) patterns are specified via tree (string) filters.

A tree filter contains a representation of a generalized parse tree path that matches a set of items on a Web page, and contains a set of conditions that these items must satisfy. All the conditions of a filter are interpreted conjunctively, i.e., an element of a Web page satisfies a filter if and only if it matches its generalized tree path and satisfies all the conditions of the filter. Similarly, a string filter specifies the characteristics of the text to be extracted (using a formal language), and possibly further conditions.

Lixto offers a wrapper designer the possibility to express various types of conditions restricting the intended pattern instances of a filter. The main types of conditions are inherent (internal) conditions, contextual (external) conditions, and range conditions. In addition to these three basic types of conditions, *Lixto* allows a designer to express auxiliary conditions like pattern reference conditions, concept conditions and comparison conditions. They are discussed as atoms of the *Elog* language in more detail in Section 3.

Visual Pattern Generation. Extraction patterns are defined by the designer in a hierarchical manner. A pattern that describes an entire document is referred to as a document pattern. In particular, the document pattern corresponding to the starting Web page, the so-called "home document pattern", is available as a pre-existing pattern. Other patterns are defined interactively. Filters or patterns are usually defined in the context of other patterns (so-called parent patterns). For example, a pattern `<name>` may be defined first, and then patterns `<firstname>` and `<familyname>`, etc., may be defined in the context of the source pattern `<name>`. For the majority of common extraction tasks, defining flat patterns or a strict hierarchy of patterns will in practice be sufficient. However, *Lixto* does not limit the pattern definition to be strictly hierarchical (i.e. tree-like). Moreover, pattern definitions are allowed to be recursive (similar to recursive type definitions in programming languages). While patterns are not required to form a strict hierarchy, pattern instances do always form one and can be arranged as a tree (or forest, in case they stem from different documents, which can be the case in recursive programs as explained in Section 6).

The visual and interactive pattern definition method allows a wrapper designer to define an extraction program and an associated XML translation scheme without any programming efforts. The *Lixto* Interactive Pattern Builder allows a wrapper designer to define filters and patterns with the help of one or more characteristic example pages, and to modify and store patterns. At various intermediate steps, the designer may test a partially or fully constructed filter or pattern, both on the example pages used to construct the pattern as well as on any other Web page. The result of such a test is a set of pattern instances, which is displayed by a browser as a set of highlighted items.

The filter description procedure for tree-filters can be described as follows: The designer marks an initial element on an example Web page (for example, a table). The system associates with this element a generalized tree path of the parse tree that (possibly) corresponds to several similar items (for example, several tables). The designer then tests the filter for the first time. If more than just the intended data items are extracted (and thus highlighted) as a result of the test, then the designer adds restrictive conditions to the filter and tests the filter again. This process is repeated as long as undesired data items are extracted. At the end of the process, the filter extracts only desired items. A similar procedure is used for designing string filters. However, for creating a string rule usually no example is selected, but some characterizations are visually composed, e.g. by relying on concept conditions. A pattern is designed by initially asserting one filter for the pattern, and, in case this is not sufficient (because testing shows that not all intended extraction items on the test pages are covered), by asserting successively more filters for the pattern under construction, until each intended extraction item is covered by at least one filter associated to that pattern.

Observe that the methods of filter construction and pattern construction correspond to methods of definition-narrowing and definition-broadening that match the conjunctive and disjunctive nature of filters and patterns, respectively. It is the responsibility of the wrapper designer to perform sufficient testing, and – if required by the particular application-test filters and patterns also on Web pages different from the initially chosen example pages. Moreover, it is up to the wrapper designer to choose suitable conditions that will work not only on the test pages, but also on all other target Web pages.

The visual and interactive support for pattern building offered by *Lixto* also includes specific support for the hierarchical organization of patterns and filters. A wrapper definition process according to *Lixto* (and consequently, a *Lixto* wrapper) is not limited to a single sample Web document, and not even to sample Web pages of the same type or structure. During wrapper definition, a designer may move to other sample Web pages (i.e., load them into the browser), continuing the wrapper definition there.

XML Translation. The XML Translation Builder which constitutes another interactive module of the wrapper generator, is responsible for supporting a wrapper designer during the generation of the XML translation scheme. By default, pattern names are used as output XML tags and the hierarchy of extracted pattern instances determines the structure of the output XML document. Thus, in case no specific action is taken by the designer, the pattern instances are translated into XML in a standard way without any need of further interaction. However, *Lixto* also offers the wrapper designer the option to modify the standard XML translation in the various ways: Renaming patterns, suppressing auxiliary patterns, writing some HTML attributes, and deciding whether instances of document patterns are all treated at the same level, or hierarchically ordered as defined by the extraction process. Moreover, to define a DTD based on an output, a wrapper designer can assign a multiplicity to each pattern, i.e. if one or several instances are required/allowed to occur within a parent pattern.

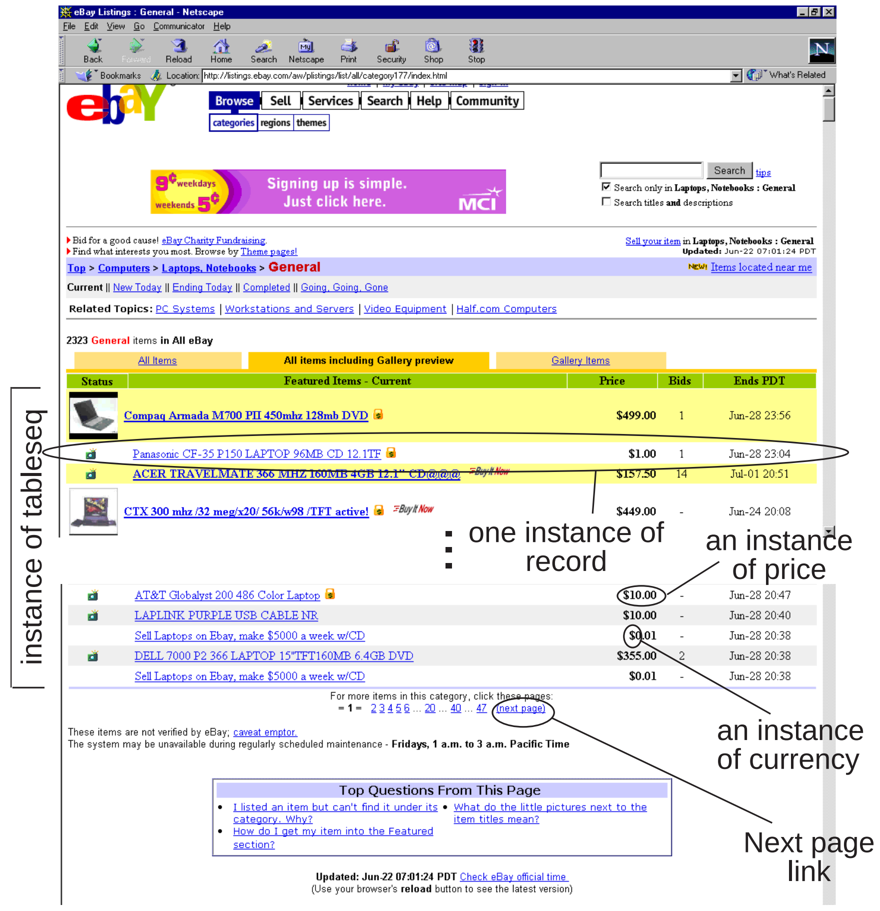
These desired modalities of the XML translation are determined during the wrapper design process by a very simple and user-friendly graphical interface and are stored in the form of an XML translation scheme that encodes the mapping between extraction patterns and the XML output in a suitable form.

3 An Overview of the *Elog* Extraction Language

As mentioned in the previous sections, patterns are internally represented using the declarative extraction language *Elog*. The *Elog* language is specifically designed for hierarchical and modular data extraction and it is ideally suited for representing and successively incrementing the knowledge about extraction patterns. It uses a datalog-like syntax and semantics, enriched with several predefined predicates related to information extraction. An *Elog* program is a collection of rules containing special extraction atoms in their bodies.

We illustrate the main characteristics of *Elog* using an example program which can be applied to *eBay* pages, e.g. to the sample page in Figure 1. Figure 2 shows an *Elog* program applied to a category search result page of *eBay*. In the following examples, we additionally use a *pattern graph* to represent a *Lixto* wrapper. A pattern graph is a directed graph whose nodes represent patterns and an arc from a pattern p_2 to a pattern p_1 specifies that there is a filter defining p_2 that extracts information from instances of p_1. Moreover, document, tree, and string patterns are represented using different shapes. Finally, it is possible to represent also information about the XML translation scheme using this graph. In particular, we specify that a pattern is translated to an XML element by writing a text "pattern name/elementname" into the pattern node. If the element name is missing, then the pattern name is used as default translation. The set of included attributes are embedded in a list, e.g. "[url, font]", and patterns that are not translated are drawn with dashed lines. It is possible to specify a minimum and maximum multiplicity on the arcs ("[min,max]", to specify the information used in the construction of the DTD (see the end of this section). When no multiplicity of a pattern is explicitly indicated in the pattern graph, then a minimum and maximum multiplicity of 1 for that pattern are assumed. The pattern graph of the program in Figure 2 is shown in Figure 3. In this case, as all filters of one pattern point to the same parent, it forms a tree.

An extraction program consists of a set of patterns. In *Elog*, a pattern p is represented by a set of rules having all the same head atom of the form $p(S, X)$. *Elog* rules define elements to be extracted from Web pages. Each rule corresponds to one filter. The head of an *Elog* rule r is always of the form $p(S, X)$ where p is a pattern name, S is a variable which is bound in the body of the rule to the


eBay Listings : General - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://listings.ebay.com/aw/plistings/list/all/category177/index.html
What's Related
Browse Sell Services Search Help Community
categories regions themes
9¢ weekdays weekends 5¢ Signing up is simple. Just click here. MCI
Search tips
instance of tableseq
one instance of record
an instance of price
an instance of currency
Next page link

☑ Search only in **Laptops, Notebooks : General**
☐ Search titles **and** descriptions

Bid for a good cause! eBay Charity Fundraising.
Find what interests you most. Browse by Theme pages!

Sell your item in **Laptops, Notebooks : General**
Updated: Jun-22 07:01:24 PDT

Top > Computers > Laptops, Notebooks > **General**   NEW! Items located near me

Current || New Today || Ending Today || Completed || Going, Going, Gone

Related Topics: PC Systems | Workstations and Servers | Video Equipment | Half.com Computers

2323 General items **in All eBay**

All Items | **All items including Gallery preview** | Gallery Items

Status	Featured Items - Current	Price	Bids	Ends PDT
	Compaq Armada M700 PII 450mhz 128mb DVD	$499.00	1	Jun-28 23:56
	Panasonic CF-35 P150 LAPTOP 96MB CD 12.1TF	$1.00	1	Jun-28 23:04
	ACER TRAVELMATE 366 MHZ 160MB 4GB 12.1" CD@@@ Buy It Now	$157.50	14	Jul-01 20:51
	CTX 300 mhz /32 meg/x20/ 56k/w98 /TFT active! Buy It Now	$449.00	-	Jun-24 20:08
	AT&T Globalyst 200 486 Color Laptop	$10.00	-	Jun-28 20:47
	LAPLINK PURPLE USB CABLE NR	$10.00	-	Jun-28 20:40
	Sell Laptops on Ebay, make $5000 a week w/CD	$0.01	-	Jun-28 20:38
	DELL 7000 P2 366 LAPTOP 15"TFT160MB 6.4GB DVD	$355.00	2	Jun-28 20:38
	Sell Laptops on Ebay, make $5000 a week w/CD	$0.01	-	Jun-28 20:38

For more items in this category, click these pages:
= **1** = 2 3 4 5 6 ... 20 ... 40 ... 47 (next page)

These items are not verified by eBay; caveat emptor.
The system may be unavailable during regularly scheduled maintenance - **Fridays, 1 a.m. to 3 a.m. Pacific Time**

Top Questions From This Page
- I listed an item but can't find it under its category. Why?
- How do I get my item into the Featured section?
- What do the little pictures next to the item titles mean?

Updated: Jun-22 07:01:24 PDT Check eBay official time
(Use your browser's **reload** button to see the latest version)

Fig. 1. Sample *eBay* page

parent-pattern instances of the filter corresponding to r, and X is the target variable which, at extraction time, is bound to some target pattern instance to be extracted (either a tree region or a textual string). The body of an *Elog* rule contains atoms that jointly restrict the intended pattern instances. For example, an *Elog* rule corresponding to a tree filter contains in its body an atom expressing that the desired pattern instances should match a certain tree path and another atom that binds the variable S to a parent-pattern instance.

In the example program, the pattern `<tableseq>` is used to extract a sequence of tables which represent records. Observe that in each search result page of *eBay*, a record is a whole table consisting of a single table row. This sequence of tables is required to be preceded by a table which contains the word "Current", and to be followed by an image representing a horizontal line.

```
ebaydocument(S, X) ← getDocument(S = $1, X)
    tableseq(S, X) ← ebaydocument(_, S),
                     subsq(S, (⋆.body. ⋆ .center, []), (.table, []), (.table, []), X),
                     before(S, X, (⋆.tr, [(elementtext, Current, substr)]), 0, 0, _, _),
                     after(S, X, (⋆.img, [(src, spacer.gif, substr)]), 0, 0, _, _)
      record(S, X) ← tableseq(_, S), subelem(S, .table, X)
     itemdes(S, X) ← record(_, S), subelem(S, (⋆.td. ⋆ .content, [(href, , 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)
        date(S, X) ← record(_, S), subelem(S, ⋆.td, X), notafter(S, X, .td, 100)
    currency(S, X) ← price(_, S), subtext(S, \var[Y], X), isCurrency(Y)
     pricewc(S, X) ← price(_, S), subtext(S, [0 − 9]+\.[0 − 9]+, X)
```

Fig. 2. *Elog* Extraction Program for a a single *eBay* page

The rule with head predicate $record(S, X)$ in Figure 2 identifies all tables within a specific area, which is the instance of $tableseq(\_, S)$. For each ground atom $tableseq(p, s)$ (where p and s are tree regions), this rule derives atoms of the form $record(s, x)$ for each table x contained in s. Thus the variable S identifies the context of the extraction, in this case, these are the instantiations of *tableseq*. Optionally, the body of an *Elog* rule may contain further atoms expressing conditions that the pattern instances should additionally satisfy. In particular, for each type of condition, there exists a built-in predicate (see below).

The description of each item (occurring in the second column of each record) is determined by the extraction rule whose head is $itemdes(S, X)$. The first atom in the rule body specifies that the context S of the extraction is a table and ensures that the variable S is instantiated with a table. The second atom in the rule body looks for subelements of the table that qualify as table columns with some specific properties, in particular requiring that they contain a link (*href*). The rule has as many matches as there are items on the given page. If the Web page is updated and two new records are inserted into the table, then the same rule will produce two more matches. Each match gives rise to a corresponding instantiation of the variable X.

Thus, the head predicates defined by an *Elog* program represent the extraction patterns defined by the wrapper program. For instance, the program in Figure 2 defines patterns such as `<record>`, `<itemdes>`. *Elog* rule bodies contain the following important ingredients. For a more detailed discussion about *Elog* predicates see Section 4.4 of [5].

Incompletely specified tree paths. These refer to the position(s) of the desired element(s) in the HTML tree. More details on the used document model are specified in [5]. There are various ways to specify a tree path pointing to e.g.

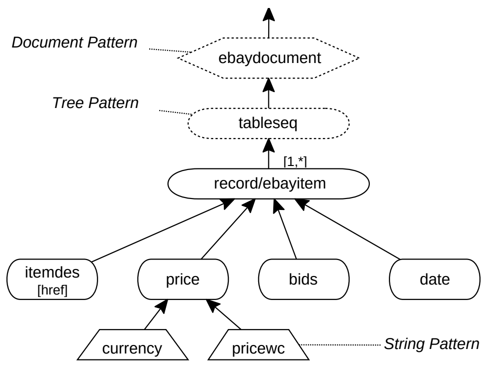


Fig. 3. Pattern Structure of Example of Figure 2

a table row in an *eBay* page. The fully specified tree path to this node is: *body.table.tr* (the elements satisfying these paths are referred to as matched pattern instances). Two incompletely specified tree paths to the same node are $.\star.body.\star.tr$ and $.\star.body.\star.table.\star.tr$, where the star signs are wildcards (the dots just act as concatenation sign). An incompletely specified tree path $.\star.name$ is an abbreviation of the skip-to sequence $(\Sigma - name)^*name$ where Σ is the alphabet of element types. The first discovered elements of the type "name" are considered in all possible paths. Observe that, interpreting the star in this way, a tree path $.\star.table$ identifies only the outermost tables in a document, and hence acts as some kind of minimization.

Attribute Conditions. An incompletely specified tree path may be too general for describing an intended extraction target. In that case, additional atoms in the rule body may express further restricting conditions. Among these are so-called *attribute conditions*. Attribute conditions impose restrictions on matched elements. For example, leaf nodes of the HTML tree representing text strings may have a *font-style* attribute which takes the value *italics* if the represented text is in italics. Moreover, we treat the contents of an element as special attribute *elementtext*. Consider the rule for *tableseq* in Figure 2: One of its predicates uses an attribute condition expressing that the elementtext needs to contain the word "Current" ("contain" due to the *substr* keyword) This attribute condition restricts the tree path $.\star.table$, which identifies tables by limiting the matches to those text fields that contain the word "Current". Attribute Conditions may require exact matches or partial matches, or satisfaction of a particular regular expression possibly extended by the use of variables.

Element Characterizations. A set of elements of a subtree of an HTML tree are identified with a tree path (starting from the subtree root), where additionally a set of attribute conditions is satisfied. Such a characterization is called an *element path definition.* Equivalently, *XPath* expressions can be used instead (with some extensions, such as the possibility to express that an attribute value is a concept like "isCity"). To simplify presentation, however, we stick to our introduced notation. A set of substrings can be identified by using a *string path definition*, which can either be a regular expression, or refer to a concept, or even combine both. Consider the example of Figure 2, in which the rule defining `<currency>` refers to a variable whose instances are currencies.

Tree Extraction Definition Predicates. These predicates specify that a variable should be instantiated with a node in the HTML tree which matches an element path definition. See, for example, the *subelem* atom of the fourth rule in Figure 2, where the variable X is instantiated to all those text fields that occur within `<record>` and contain a link. The variable S in this atom denotes the super entity or, as we call it, the *parent pattern*, from which the current target should be extracted via *subelem*. This parent pattern instance is constrained to be an instance of `<record>` by the first atom of the rule. Note that the tree path specified in a tree extraction definition predicate is always relative to the parent pattern, i.e., its starting point is a node corresponding to the parent pattern (in our example rule, an instance of `<record>`). Moreover, with *subregion*, a sequence of elements can be extracted (e.g. used in *tableseq* in Figure 2).

String Extraction Definition Predicates. In the HTML parse tree, strings are represented by the text of leaves of type *content*. However, we associate a string C_n to *every* node n of the parse tree by simply concatenating (in left-to-right order) all strings corresponding to leaves of the subtree rooted in n. The string C_n associated to node n is available in the *Lixto* system as the value of an additional attribute *elementtext* of any given node n. Several special conditions that express restrictions on such elementtexts can be expressed in *Elog*. *Elog* predicates expressing such special string conditions are referred to as *string extraction definition predicates.* As an example, consider the final two rules of the program of Figure 2. The last rule uses a regular expression as string path definition, the other one a variable reference to a concept atom (explained below). Moreover, *Attribute Extraction Predicates* such as *subatt* (see examples in Section 6) allow to extract the contents of attribute values.

Contextual Conditions. Contextual conditions specify that some other elements must or must not appear either before or after some instance. These contextual elements are not limited to text elements. 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. The rule defining a `<tableseq>` uses both an *after* and a *before* condition to express that one is interested in exactly the region between some specified elements. The definition of `<date>` uses a *notafter* condition to express that the column which contains the date is not followed by another column.

Internal Conditions. Such conditions require that some characteristic feature must or must not appear within an instance. Imagine, one wants to extract all tables containing a word typeset in italics. This could be obtained by adding an internal condition called *contains* to the body of the rule that defines the pattern `<record>`. This condition expresses that in the subtree rooted at the node representing the desired table row, a node must exist whose *font-style* attribute is defined and has the value *italics.*

Concept Conditions. These predicates define concepts of some built-in top-level ontology. For example, one may enrich the system with predicates $isEmail(X)$, $isCountry(X)$, or $isCurrency(X)$ (see Figure 2), stating that a string X represents an email address, a country, or a currency, respectively. These values of the variable X are created as output of concept attribute conditions or string path definitions (using $\backslash var[X]$). They are not required to be unary, e.g. $isDate(X, Y)$ is a binary predicate with output Y in standard date format.

Comparison Conditions. These are predefined relations for predefined ontological classes of elements. Using these conditions, one can e.g. compare two dates (binary predicate), or require that an email address exists (unary predicate).

Pattern References. Each standard filter contains a reference to its parent pattern which defines the context of a rule. For example, see the rule defining `<itemdes>` in Figure 2. It refers to `<record>` as parent. The substitution for S is the actual tree region which acts as parent instance. Moreover, additional pattern references can be used, for instance to express that an instance of some pattern always occurs after an instance of another pattern. Such additional pattern references open the way for reference recursion (see Section 7 for details).

Range Conditions. A range condition further restricts the set of pattern instances extracted by a filter by selecting only a subset of the pattern instances which satisfy the conditions in the body of the filter. Indeed the pattern instances extracted from a certain parent pattern instance are ordered according to their position in the document, and a range condition selects only those pattern instances that belong to the required range of solutions. To any rule a range condition such as "[3,7]" can be added, indicating that the solution only includes the third up to the seventh matched target. Counting can occur starting with the first or with the last instance.

Using the above predicates, a *standard extraction* rule looks as follows:

$$\mathtt{New(S,X)} \leftarrow \mathtt{Par(\_,S), Ex(S,X), Co(S,X,\ldots)[a,b]}$$

where S is the parent instance variable, X is the pattern instance variable, $Ex(S, X)$ is an extraction definition predicate, and the optional $Co(S, X, \ldots)$ 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. This standard rule reflects the principle of aggregation.

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* iff 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 instances are matched (i.e. instances that do not contain any other instances). This is a nonmonotonic concept discussed in Section 7. Observe that range criteria are applied after non-minimal targets have been sorted out. Note that range conditions are well-defined only in the case of no reference recursion (cf. to Section 7).

A *pattern definition* (for short, *pattern*) is a set of extraction rules defining the same head. We distinguish document, tree and string patterns. To tree patterns, only tree extraction rules can be asserted, and to string patterns only string extraction rules. The third kind of patterns, document patterns, are discussed in the next section. 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 set of matched target instances of a pattern additionally obeys a minimality criterion (see Section 7). In patterns, even in those consisting of a single rule, overlapping targets may occur. Observe that we do not pose the requirement that each rule belonging to a given pattern refers to the same parent pattern. This, together with the capability of document navigation, allows for recursion over patterns as explained in more detail in Section 6.

An extraction program P is a set of patterns. *Elog* program evaluation differs from Datalog evaluation in the following three aspects: The use of built-in predicates, various kinds of minimization, and the use of range conditions. Moreover, atoms are not evaluated over an extensional database of facts representing a Web page, but directly over the parse tree of the Web page.

The application of 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 to a given and possible further HTML pages (see the notion of document filters in Section 4). Each pattern produces a set of instances. Each pattern instance contains a reference to its parent instance. Observe that the pattern instance base always forms a forest, regardless of the structure of the pattern graph. We consider the instances of document filters as root node of each tree of this forest. The pattern instance base can be translated into XML as already described in Section 2.

4 A Closer Look at some *Lixto* Features

In this section, we discuss some more advanced features of *Lixto*, in particular two further kinds of rules. A standard rule reflects the principle of aggregation, however, designers of wrappers sometimes wish to express specialization. For instance, if one rule extracts a set of tables, it might be desirable to create a rule which restricts the extracted tables to those which contain some particular feature. A *specialization rule* looks as follows:

$$\texttt{New(S,X)} \leftarrow \texttt{Old(S,X)}, \texttt{Co(S,X,\ldots)}[\texttt{a},\texttt{b}]$$

In such a rule a pattern is specialized, i.e. some of the parent-pattern instances are returned as pattern instances of the new pattern definition. It does not contain a parent-pattern reference and an extraction definition atom; instead it only contains a pattern reference. Observe that equally to specialization rules, generalization rules can be used by simply creating multiple specialization rules for one pattern which refer to different patterns and do not contain any conditions. Another kind of rule is the *document rule*, 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 rules, one can crawl to further documents.

$$\texttt{New(S,X)} \leftarrow \texttt{Par(\_,S)}, \texttt{getDocument(S,X)}$$

Each *Elog* program has an initial rule using the *getDocument* atom with user-specified input. The initial document rule is the only rule without a parent-pattern reference. Instead, it uses a variable "$1" (or a fixed URL) which is instantiated to a string source representing an URL during run time (the start document). Document filters can be applied to document patterns only. Parents of tree patterns are either tree or document patterns, parent of string patterns are tree or string patterns, and parents of document patterns are string patterns.

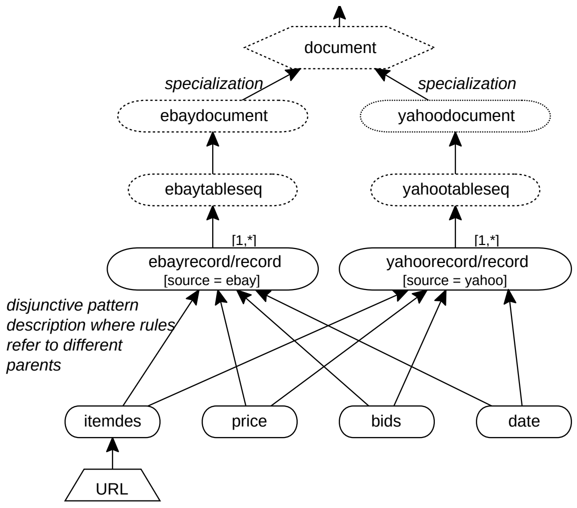


Fig. 4. Wrapper for *eBay/Yahoo* using Specialization and Disjunctive Patterns

Figure 4 illustrates the use of document rules together with specialization rules. This example moreover illustrates the use of disjunctive pattern definitions pointing to two different parents which actually evolved in this case from two different kind of documents. Consider the root pattern `<document>` and its child patterns `<ebaydocument>` and `<yahoodocument>`. Both are specializations requiring that the document is an *eBay* page (a category search result on `www.ebay.com` such as `http://listings.ebay.com/aw/plistings/list/all/category3707/index.html`), or a *yahoo auctions* page (i.e., a search result of `auctions.yahoo.com`), respectively. Observe that the patterns `<ebaydocument>` and `<yahoodocument>` are not document patterns, but tree patterns, since they refer to instances of tree regions. The predicate *contains* is an internal condition, expressing that there is an element in X which satisfies the given element path definition.

$$
\begin{aligned}
\texttt{document(S,X)} &\leftarrow \texttt{getDocument(S} = \$1\texttt{,X)}\\
\texttt{ebaydocument(S,X)} &\leftarrow \texttt{document(S,X)},\\
&\quad\ \texttt{contains(X,}(\star\texttt{.body},[(\texttt{elementtext},\texttt{eBay},\texttt{substr})]),\_)\\
\texttt{yahoodocument(S,X)} &\leftarrow \texttt{document(S,X)},\\
&\quad\ \texttt{contains(X,}(\star\texttt{.body},[(\texttt{elementtext},\texttt{Yahoo},\texttt{substr})]),\_)
\end{aligned}
$$

5 Disjunctive Pattern Construction

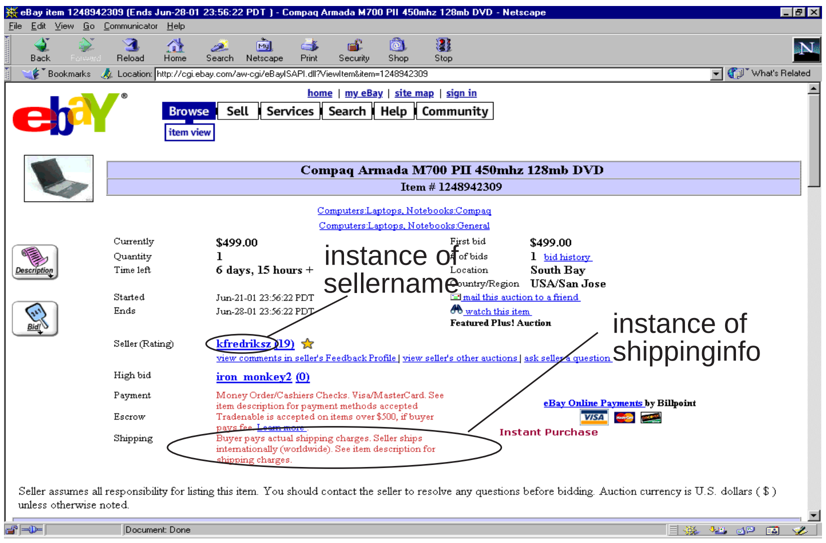
There are several cases, where it is necessary to define more than one filter for the same pattern to express how to extract desired pieces of information from a Web page. In this section we show some real world examples where it is useful to define a pattern using a disjunction of filters. Moreover, we show that is generally possible that different filters of the same pattern can extract information from different parent patterns. Let us first consider an example where a wrapper designer wants to define a pattern consisting of filters that describe extraction targets for different page types. Assume a wrapper extracts prices from two kind of Web pages displaying books and their prices, where pages of the first kind are US pages and pages of the second kind are UK pages. The characteristic features of prices are a dollar sign on US Web pages and a pound sterling sign on UK pages. Assume, furthermore, the current sample page is a US page. A pattern named `<price>` should thus be defined via two filters: the first taking care of US pages and the second of UK pages. After having visually created an appropriate filter for prices in USD on an already loaded US sample page, the designer switches to a UK sample page and visually defines the second filter for the `<price>` pattern on that page. The wrapper then works on both types of pages.

In *Lixto* it is not only possible to create a pattern consisting of several filters, but also that filters of a particular pattern definition refer to a different parent pattern. Again, consider the example in Figure 4. For both the `<ebaydocument>` and the `<yahoodocument>` pattern we now have to extract the list of available items (records). Since records are structured differently in *eBay* and *yahoo auctions*, it is necessary to create for each kind of page a record pattern of its own, i.e. `<ebayrecord>` and `<yahoorecord>`. Once we have defined the patterns for the records, the patterns `<itemdes>`, `<price>`, `<bids>` and `<date>` can be easily defined with one filter for each kind of record. Although this wrapper works fine for both *yahoo* and *eBay* auctions, it still only returns results from one summary page as it does not follow the "next" link, and also is not capable of extracting detail information. Moreover, using the pattern `<itemdes>` as parent, a string pattern *URL* is defined using an attribute filter. This attribute filter extracts the value of the link to detailed information of the particular item. This attribute filter works for both sites, since both store the URL pointing to the detail page in the corresponding *href* attribute.

$$\texttt{URL(S,X)} \leftarrow \texttt{itemdes(\_,S)},\texttt{subatt(S,href,X)}$$

An attribute filter uses the extraction definition predicate *subatt* to extract an attribute value of instances of S and instantiates a string source X with it. The following additional features are currently implemented and can be added via *Lixto*'s XML Tool:

1. The pattern `<ebayrecord>` and `<yahoorecord>` can be both mapped to the XML element `<record>`, and an attribute *source* of `<record>` can be defined, which takes the constant value *eBay* or *yahoo*, respectively.
2. In case the string source of `<URL>` is a relative URL, a prefix variable (BASE) can be added to it, which has the value of the base URL of the document from which the information is extracted. This variable can also be used for following relative links when crawling to further pages (see next section).
3. Auxiliary patterns such as `<ebaydocument>` and `<yahoodocument>` can be decided to not being mapped to XML, and a DTD can be created by additionally assigning a multiplicity to each data type (Figure 4).


File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: What's Related
home | my eBay | site map | sign in
Browse Sell Services Search Help Community
item view
Description
Bid!
instance of sellername
instance of shippinginfo
Document: Done

eBay item 1248942309 (Ends Jun-28-01 23:56:22 PDT) - Compaq Armada M700 PII 450mhz 128mb DVD - Netscape

http://cgi.ebay.com/aw-cgi/eBayISAPI.dll?ViewItem&item=1248942309

Compaq Armada M700 PII 450mhz 128mb DVD
Item # 1248942309

Computers:Laptops, Notebooks:Compaq
Computers:Laptops, Notebooks:General

Currently	$499.00	First bid	$499.00
Quantity	1	# of bids	1 bid history
Time left	6 days, 15 hours +	Location	South Bay
		Country/Region	USA/San Jose
Started	Jun-21-01 23:56:22 PDT		mail this auction to a friend
Ends	Jun-28-01 23:56:22 PDT		watch this item
			Featured Plus! Auction
Seller (Rating)	kfredriksz (19)		
	view comments in seller's Feedback Profile | view seller's other auctions | ask seller a question		
High bid	iron_monkey2 (0)		
Payment	Money Order/Cashiers Checks. Visa/MasterCard. See item description for payment methods accepted		eBay Online Payments by Billpoint
Escrow	Tradenable is accepted on items over $500, if buyer pays fee. Learn more.		Instant Purchase
Shipping	Buyer pays actual shipping charges. Seller ships internationally (worldwide). See item description for shipping charges.		

Seller assumes all responsibility for listing this item. You should contact the seller to resolve any questions before bidding. Auction currency is U.S. dollars ($) unless otherwise noted.

Fig. 5. Ebay item description page

6 Web Crawling and Recursive Wrapping

6.1 Following Links

For each item, *eBay* pages contain a reference to a page containing detailed information about the item itself. In the previous section, we have shown how to extract the URL pointing to detail pages, but we did not further use it. In this section we extend the wrapper program to extract also the detailed description of each item. This is an instance of a general class of applications, where a wrapper needs to collect and group together elements from several pages. The wrapper designer thus needs to "teach" the system on the base of sample pages how to follow URLs and collect the elements from the different pages. On *eBay*, each item is described by a line stating summary information for each given auction item. Each such line contains a link to a Web page with more detailed information on the respective item, such as the seller name and the shipping information (Figure 5).

The designer adds a child document pattern `<detaildocument>` to the string pattern `<URL>` which resulted from extracting the value of the *href* attribute of `<itemdes>`. For this, the designer proceeds by following one example detail document, loading the corresponding page, and defining the remaining relevant patterns (such as "sellername" and "shippinginfo") as child patterns of this new document pattern. Figure 6 illustrates an expanded *Elog* program of Figure 3, which defines an attribute filter extracting an URL (as in Figure 4), and a further document pattern consisting of one filter to extract detailed information for each item. The auxiliary patterns `<URL>` and `<detaildocument>` are not mapped to XML via the XML translation scheme. The navigation to a detail document looks as follows:

$$\begin{aligned} \mathtt{URL}(\mathtt{S},\mathtt{X}) &\leftarrow \mathtt{itemdes}(\_,\mathtt{S}), \mathtt{subatt}(\mathtt{S},\mathtt{href},\mathtt{X}) \\ \mathtt{detaildocument}(\mathtt{S},\mathtt{X}) &\leftarrow \mathtt{URL}(\_,\mathtt{S}), \mathtt{getDocument}(\mathtt{S},\mathtt{X}) \end{aligned}$$

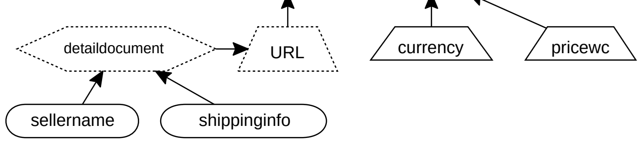


Fig. 6. Following Links

6.2 Recursive Wrapping

As we have already pointed out, each filter of a given pattern may refer to a different parent pattern. Here, we show how to apply this feature to reuse patterns. This paves the way for creating recursive programs. We call this kind of recursion *pattern recursion*. Another kind of recursion, *reference recursion*, based on pattern references is discussed in Section 7.

Let us first consider the example program below.

```
document(S, X) ← getDocument($1, X)
   table(S, X) ← document(_, S), subelem(S, . ⋆ .table, X)
   table(S, X) ← table(_, S), subelem(S, . ⋆ .table, X)
```

It extracts all nested tables within one page, starting with the outermost, and stores them in this hierarchical order in the pattern instance base. The second rule of `<table>` is iteratively called, until no further table can be extracted.

Another possible use of recursively defined wrappers is the following real-world application. Usually a wrapper designer does not want to extract data from a single *eBay* page on notebooks, but from all pages which are connected to each other via a “next page” link. We illustrate how the *eBay* program of Figure 6 can be extended to follow the next link and can reuse the already created pattern structure. Thus, the pattern `<ebaydocument>` is a document pattern consisting of two filters with different parents. The first one refers to the specified start document, whereas the second one follows the “next” link on each page. This part of the program looks as follows:

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

Recall that ”$1” is interpreted as a constant whose value is the URL of the start document of a *Lixto* session. This initial filter was already present in the previous example, and is the starting point of evaluation. The second filter refers to a different parent pattern, which is `<nexturl>`. Instances of the pattern `<nexturl>` are string sources which represent an URL. The pattern `<nexturl>` is created via an attribute filter which extracts via *subatt* the value of “href” present in the element which contains the text “next page”.

In the second rule defining the pattern `<ebaydocument>`, the variable S is instantiated with string sources which represent URLs. For each “next” link, a new instance of `<ebaydocument>` is created, pointing to the next page. This new page serves as parent pattern for `<tableseq>` and `<next>`. The pattern structure is hence re-used for this new page. In this example, two different document patterns are used, on the one hand `<ebaydocument>`, on the other hand `<detaildocument>`. Instances of the pattern `<ebaydocument>` are the summary pages, whereas instances of `<detaildocument>` are the detail information pages for each item. In an XML translation scheme, the wrapper designer moreover wants to state how the documents are arranged inside the XML document. Although further instances of `<ebaydocument>` are hierarchically embedded in the previous one, the wrapper designer may maintain all `<record>` instances on the same level.

In the visual interface of *Lixto*, a document pattern can be generated without the need to manually define auxiliary patterns. Instead visual guidance is offered for creating a single rule which uses a sequence of extraction definition predicates. For this example program, this single rule can be represented as follows:

$$\begin{aligned}\texttt{ebaydocument}(\texttt{S},\texttt{X}) \leftarrow\ & \texttt{ebaydocument}(\_,\texttt{S}), \texttt{subatt}(\texttt{Y},\texttt{href},\texttt{Z}), \texttt{getDocument}(\texttt{Z},\texttt{X})\\ & \texttt{subelem}(\texttt{S},(\star.\texttt{content},[(\texttt{href},,\texttt{substr}),\\ & \ (\texttt{elementtext},(\texttt{next page}),\texttt{exact})]),\texttt{Y}),\end{aligned}$$

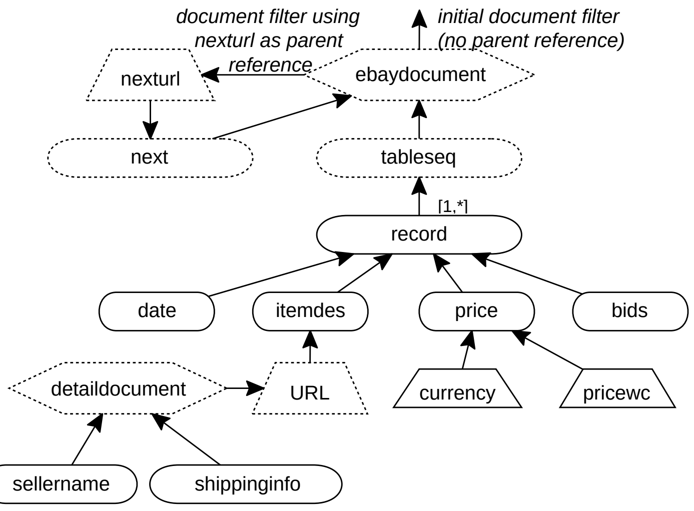


Fig. 7. Recursive Extraction

7 Nonmonotonic Issues

Minimization of pattern instances. The set of matched targets of an *Elog* pattern are minimized in the way that pattern instances which contain other instances of the same pattern w.r.t. the same parent-pattern instance, are ignored. Pattern minimization applies both to tree and string rules. If a pattern consists of a single filter, the minimized set of its matched targets equals the initial set except if the extraction definition predicate of the filter is *subregion* (which extracts a sequence of elements).

Consider the following simple example. Assume the major headlines of a particular newspaper Web page are a table consisting of various table data (the wrapper designer is interested in all the contents), and the minor headlines of the same newspaper which appear at the same page, are table columns of another table. The minor headlines are moreover characterized by a red font, and the major headlines contain a link (*href*) somewhere. However, the table containing the data of all minor headlines also contains links (i.e. the *href* attribute is a characteristic attribute occurring in these two tables only). A program for extracting all headlines can be written in the following way, where *par* is the parent pattern identifying the relevant area of the newspaper page.

$$\begin{aligned}\mathtt{headline(S,X)} &\leftarrow \mathtt{par(\_,S), subelem(S,.\star.table,X)}\\ &\quad\ \mathtt{contains(S,(.content,[(href,,substr)]),X)}\\ \mathtt{headline(S,X)} &\leftarrow \mathtt{par(\_,S), subelem(S,.\star.td,X)}\\ &\quad\ \mathtt{contains(S,(.content,[(font-color,red,exact)]),X)}\end{aligned}$$

Hence, the first rule also matches the table which contains all minor headlines. However, since in this table, other pattern instances are matched, too, only the minimal instances are returned, which are in this case the table columns. For the major headlines, however, the table is extracted. Another example is the minimization of the set of instances generated by a single rule:

$$\mathtt{tableseq(S,X)} \leftarrow \mathtt{par(\_,S), subregion(S,.\star.body.\star.center,.table,.table,X)}$$

Such a rule (with additional conditions) is used in the *eBay* program of Figure 2. However, with no additional condition, the semantics is to extract all possible sequences of tables and to minimize the result; since the minimal sequences of tables are sequences of a single table, this rule returns such instances only. To enforce a particular longer sequence of tables, such as the sequence of tables containing the relevant data of sold items, some before and after conditions need to be added. In the case of *eBay*, immediately before and after the target instance a particular text or image shall occur, respectively. This returns a single pattern instance, the sequence of desired record tables.

Pattern minimization can be expressed in *Elog* extended with stratified negation and a suitable built-in predicate *contained_in(X,Y)* expressing offset-wise containment of X in Y. In particular, a set of filters of $p(S,X)$ defining the pattern p is rewritten in the following way. Consider the initial pattern definition:

$$\begin{aligned}\mathtt{p(S,X)} &\leftarrow \mathtt{par_1(\_,S), Ex_1(S,X), Co_1(S,X,\ldots)}\\ \mathtt{p(S,X)} &\leftarrow \cdots\\ \mathtt{p(S,X)} &\leftarrow \mathtt{par_n(\_,S), Ex_n(S,X), Co_n(S,X,\ldots)}\end{aligned}$$

The pattern name is renamed to p' and additional rules are added:

$$\begin{aligned}\mathtt{p'(S,X)} &\leftarrow \mathtt{par_1(\_,S), Ex_1(S,X), Co_1(S,X,\ldots)}\\ \mathtt{p'(S,X)} &\leftarrow \cdots\\ \mathtt{p'(S,X)} &\leftarrow \mathtt{par_n(\_,S), Ex_n(S,X), Co_n(S,X,\ldots)}\\ \mathtt{p''(S,X)} &\leftarrow \mathtt{p'(S,X), p'(S,X_1), contained\_in(X_1,X)}\\ \mathtt{p(S,X)} &\leftarrow \mathtt{p'(S,X), not\ p''(S,X)}\end{aligned}$$

The final rule requires that instances of X and X_1 are both from the same parent pattern instance (otherwise, if they stem from different parent-pattern instances, minimization is usually undesired). In the rewriting, p' is the pattern predicate initially being built by different filters. Each instance $p(s, x)$, which is non-minimal, i.e. for which there exists a smaller valid $p''(s, x)$, is not derived. Only minimal instances are derived.

Ranges. The semantics of range criteria $[a, b]$ of a filter rule $NewPat(S, X) \leftarrow filterbody[a, b]$ can also be expressed by a suitable rewriting of the rule. A range condition assumes that an order relation is defined among pattern instances extracted by the same parent pattern instance, thus in the rewriting we assume the presence of a predicate $greater(S, X, Y)$ which evaluates to *true* if X and Y are instances derived from S and X precedes Y (using character offsets for comparison). The first step of rewriting consists of adding a new predicate $NewPat'$ that is defined by a unique filter $NewPat'(S, X) \leftarrow filterbody$. Then, two predicates $FirstSol$ and *succ* are defined. $FirstSol$ selects from the instances in $NewPat'$ the first instance, and *succ* defines a successor relation among instances in $NewPat'$ (due to the lack of space we omit the formal definition). The complete rewriting is as follows:

$$\begin{aligned}
\mathtt{NewPat(S,X)} &\leftarrow \mathtt{NewPat'(S,X), Solposition(S,X,P), a \leq P \leq b}\\
\mathtt{Solposition(S,X,1)} &\leftarrow \mathtt{NewPat'(S,X), FirstSol(S,X)}\\
\mathtt{Solposition(S,X,P)} &\leftarrow \mathtt{Solposition(S,X',P'), NewPat'(S,X), succ(S,X',X), P = P' + 1.}
\end{aligned}$$

In both predicates $FirstSol$ and *succ*, the predicate $NewPat'$ appears negated, hence, the predicate $NewPat$ depends on negation of all the predicates appearing in $filterbody$.

Pattern Reference Recursion and Ranges. Using ranges together with pattern references might introduce unstratified negation. Using pattern references can introduce *reference recursion.* Still, without ranges, a unique model is returned. However, additionally allowing range conditions to occur in such recursive rules requires to use a semantics akin to the stable model semantics (returning multiple models) or well-founded semantics (returning a minimal model) as this introduces unstratified negation into the program (considering the above rewriting). For the following example (possibly containing additional filters for p and q), a nonmonotonic semantics is required.

$$\begin{aligned}
\mathtt{p(S,X)} &\leftarrow \mathtt{par(\_,S), subelem(S,epd,X), before(S,X,\ldots,Y), q(S,Y)[a,b]}\\
\mathtt{q(S,X)} &\leftarrow \mathtt{par(\_,S), subelem(S,epd,X), before(S,X,\ldots,Y), p(S,Y)[c,d]}
\end{aligned}$$

Observe that a program which uses range and pattern recursion, but no reference recursion, is always locally stratified, i.e. its ground instantiation is stratified. For implementation issues, we limit pattern references in the way that the program remains locally stratified. This is a subset of programs whose rewriting contains only stratified negation.

8 Current/Future Work

Further work includes to consider various extensions of *Elog* such as using stratified negation instead of special negative predicates like *notbefore*, to extend handling of pattern references together with recursion as discussed above, to study further possibilities of conditions such as universially quantified ones (that require all elements to have a particular feature), and complement extraction (e.g. to remove advertisments from Web pages). An editor of *Elog* rules will be offered for more experienced wrapper designers who nevertheless lack programming facilities. This editor describes *Elog* patterns using a colloquial pattern description language. A concept editor for adding syntactic and semantic concepts to the list of built-in predicates is currently under construction. Moreover, the *Lixto* prototype is currently being re-designed as servlet version allowing pattern generation in the user's favorite browser. Finally, an *Elog2XSLT* conversion tool is going to be developed which will transform a subset of possible *Elog* programs into XSLT.

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. of SIGMOD*, 1998.
3. P. Atzeni and G. Mecca. Cut and paste. In *Proc. of PODS*, 1997.
4. R. Baumgartner, S. Flesca, and G. Gottlob. Supervised wrapper generation with Lixto. To appear in *Proc. of VLDB (Demonstration Session)*, 2001.
5. R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with lixto. To appear in *Proc. of VLDB*, 2001.
6. S. Chawathe. Describing and manipulating XML data. *Bulletin of the IEEE Technical Committee on Data Engineering*, 22(3):3-9, 1999. Invited paper.
7. H. Davulcu, G. Yang, M. Kifer, and I.V. Ramakrishnan. Computat. aspects of resilient data extract. from semistr. sources. In *Proc. of PODS*, 2000.
8. C-N. Hsu and M.T. Dung. Generating finite-state transducers for semistructured data extraction from the web. *Information Systems*, 23/8, 1998.
9. G. Huck, P. Fankhauser, K. Aberer, and E.J. Neuhold. JEDI: Extracting and synthesizing information from the web. In *Proc. of COOPIS*, 1998.
10. N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proc. of IJCAI*, 1997.
11. L. Liu, C. Pu, and W. Han. XWrap: An extensible wrapper construction system for internet information. In *Proc. of ICDE*, 2000.
12. 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.
13. I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In *Proc. of 3rd Intern. Conf. on Autonomous Agents*, 1999.
14. A. Sahuguet and F. Azavant. Building light-weight wrappers for legacy web datasources using W4F. In *Proc. of VLDB*, 1999.