The Elog Web Extraction Language*

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. This paper illustrates some aspects of the visual wrapper generation tool *Lixto* and describes its internal declarative logic-based language *Elog*. In particular, it gives an example scenario and contains a detailed description of predicates including their input/output behavior and introduces several new conditions. Additionally, entity relationship diagrams of filters and patterns are depicted and some words on the implementation are issued. Finally, some possible ramifications are discussed.

1 Introduction and System Architecture

Almost every kind of information is available on the Web, however one cannot query this information in a convenient way. The task of a *wrapper* is to identify and isolate the relevant parts of Web documents, and to automatically extract those relevant parts even though the documents may continually change contents and even (to a certain extent) structure. The wrapper transforms the extracted parts into XML (or a relational database) to make them available for querying and further processing. The idea of *Lixto* is to visually and interactively assist a developer in creating and using wrapper programs able to perform these tasks.

Lixto [3, 4] is a visual and interactive wrapper generation and data extraction tool which can be used to create *XML companions* of HTML pages. It can extract relevant information of an HTML page and pages which are linked to it. Information about related approaches on wrapper generation can be found in [1, 6–11]. In this paper we give an overview of the internal language used by *Lixto*, the logic-based declarative *Elog* web extraction language, in particular of its extraction predicates. The architecture of *Lixto* is as follows. The *Extraction Pattern Builder* guides a wrapper designer through the process of generating a wrapper. The extracted data of the sample page she works on is stored in an internal format called the *Pattern Instance Base*. As output, an *Extraction Program* is generated which can be applied onto structurally similar pages. The *Extractor* module is the interpreter of the internal language *Elog* (which is invisible to the wrapper designer), and can be used as stand-alone module on an

* All methods and algorithms of the *Lixto* system are covered by a pending patent. For papers on *Lixto* and further developments see www.lixto.com.

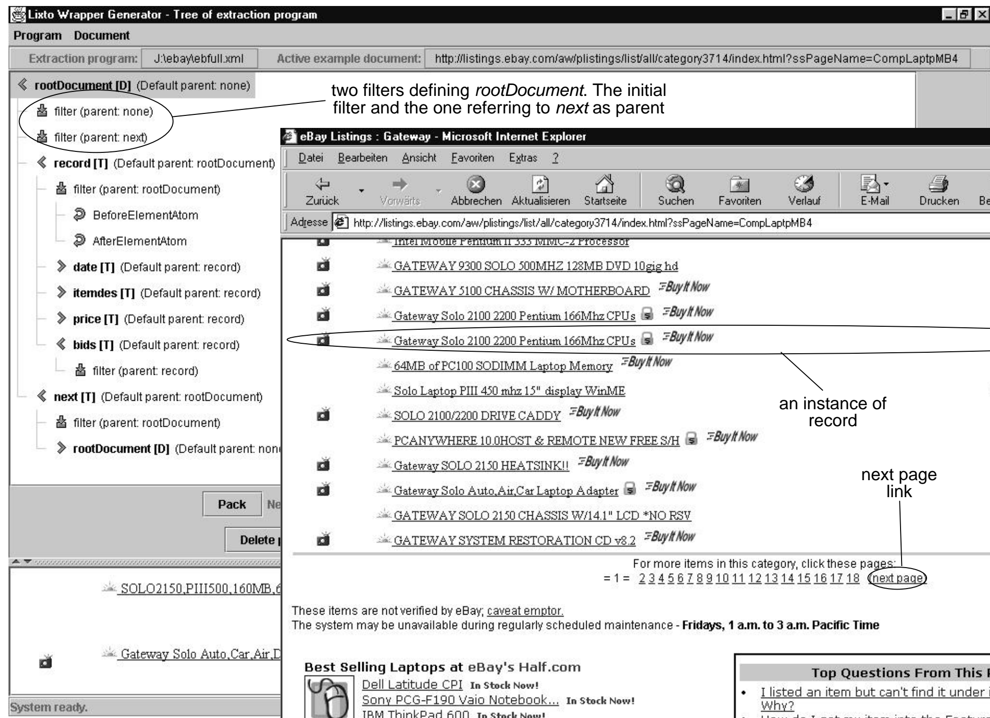

Fig. 1. Pattern hierarchy of a recursive *eBay* program and a sample *eBay* page

extraction program and an HTML page to generate an XML companion. To this end, the internal data format of the pattern instance base can be selectively mapped to XML by using the visual *XML Translation Builder* to generate an *XML Translation Scheme* which can be accessed by the *Extractor*.

This paper is organized as follows. In Section 2, the visual wrapper generation of *Lixto* is explained in an example-based way, whereas Section 3 is devoted to a description of the syntax and semantics of *Elog*, in particular its predicates, and a description of filters and patterns and their relationships. Section 8 gives an outline of the actual *Lixto* implementation. The final section describes some current and future work. For a general overview of *Lixto*, empirical results, and some comparison with competing tools we refer to [4]. For the details of one example, we refer to [3]. For further discussion of some advanced features of *Lixto* such as recursive aspects, we refer to [2].

2 User View

We briefly describe how to work with the *Extraction Pattern Builder*. In particular, the following screenshots (Figs. 1, 2, 9) are taken from the new release of the current *Lixto* beta version. With *Lixto*, a wrapper designer can create a wrapper

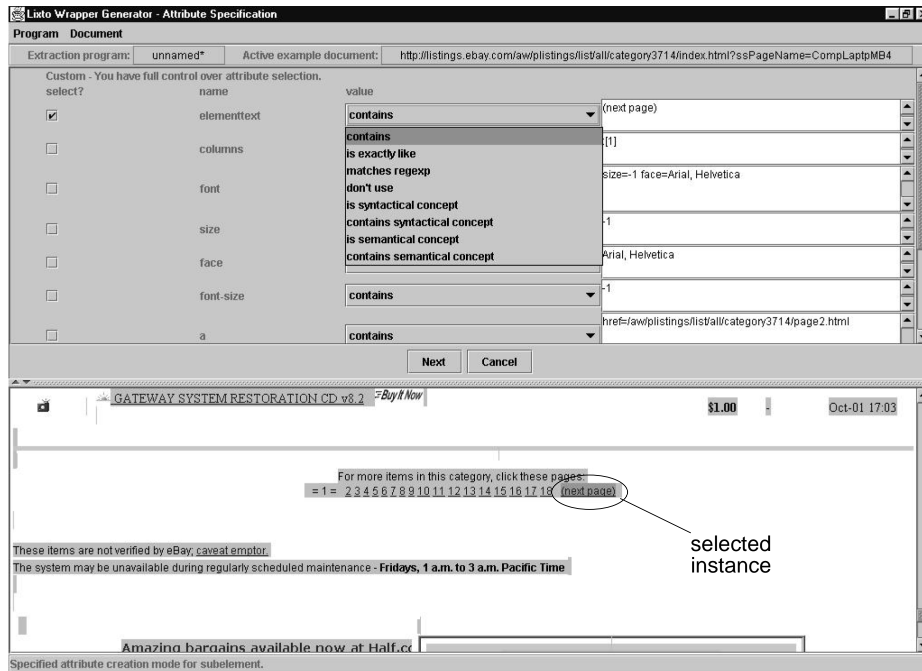

Fig. 2. Adding a filter extracting the tree region containing "next page"

program in a fully visual and interactive way by teaching the system what to extract based on one (or more) example pages. Each such program consists of a number of *patterns*. Each pattern characterizes one kind of information, e.g. all prices.

After creating a new program and opening a sample document, the designer can start to define patterns. She selects a pattern type (tree, string or document pattern) and a pattern name. Patterns carry user-defined names which are also used as default XML tags. To each pattern one or more filters can be added. A pattern extracts the instances matched by all its filters. In our implementation, when defining a filter, the designer selects an example instance and an attribute selection mechanism, and in the background the system generates a basic *Elog* rule representing this filter by choosing a suited element path and some attributes. Then the designer can test which instances are matched by the current filter.

If undesired targets are matched, she has the choice to refine the filter by adding conditions to it. Filters are added as long as some desired pattern instances are not yet matched. Alternately imposing conditions and adding new filters the desired information can be perfectly characterized.

Assume an example scenario in which a developer wishes to create a wrapper program for *eBay*. First, the wrapper designer chooses a relevant example page, e.g. a page on notebooks of a particular brand (Fig. 1). Next, the designer adds

a pattern which identifies records by simply choosing one example record, and adding additional conditions such as "somewhere before an instance a headline must occur", and "somewhere after an instance a horizontal rule image must occur". After having constructed a *record* pattern, she chooses to add child patterns of *record* which characterize item descriptions, prices, dates and numbers of bids. In the case of dates she e.g. uses the fact that it is the last table data entry of a record. In the case of an item description, the wrapper relies on the occurrence of hyperlinks, and in case of prices, a term of the predefined concept *isCurrency* is required to be part of the contents. The generated *Elog* rules which are usually hidden from the wrapper designer are given in Figure 6.

As the relevant information is not presented on a single *eBay* page only, but split over several, in this case 18 pages, the designer is interested in mapping the data of subsequent pages in the same fashion to the same XML document. *Lixto* allows the user to re-use the pattern structure defined for a single page due to support of recursive features. In this case, the designer uses the value of the "next" link (see Fig. 2), whose attribute value *href* is used for extracting further *eBay* documents.

First, the wrapper designer clicks selects the "(next page)" element through two consecutive mouse-clicks directly in the browser display. Fig. 2 shows the manual attribute selection user interface. In this case, a unique match on each page is required. Therefore the wrapper designer imposes strict criteria such as that the content of the selected element has to contain the text "next page". Alternatively, the wrapper designer could choose to enter a regular expression or predefined concepts. In this case, the filter matches exactly the desired instance, so there is no need to impose additional conditions. As next step, the wrapper designer adds a filter to *ebaydocument* which points to the parent *next*. In this way, the pattern structure is altered from a plain tree to a cyclic pattern graph, and extracted instances of *next* are used as input instances to extract further instances of *ebaydocument*. The resulting pattern graph is visualized in *Lixto* as partially expanded infinite tree (see Fig. 1).

With the *XML Translation Builder*, a wrapper designer can define which patterns shall be mapped to XML and in which way. One can for instance define that all *eBay* records are treated on the same level, and that the *next* pattern is not written to XML. A part of the resulting XML companion is depicted in Figure 3. It can be used for further information processing. If the page structure does not change significantly, the *Extractor* will continue to work correctly on new pages, especially if rather stable conditions had been chosen by the wrapper designer. Even designers neither familiar with HTML nor capable of script programming can create complex *Lixto* wrappers.

3 *Elog* Language Definition

3.1 Document Model and Extraction Mechanisms

As explained in [2, 4], *Elog* operates on a tree representation of an HTML document. The nodes of the HTML tree are referred to as elements. Each element

is associated with an attribute graph which stores pairs of attribute-designators and corresponding attribute-values, a node number, and a start and end character offset. The extraction objects over which variables range are *tree regions* and *string sources* of a given HTML document. A tree region characterizes lists of elements (e.g. a list of three table rows) or single elements (e.g. a paragraph). A filter is internally represented as a "datalog like" rule. Conditions are reflected by additional body atoms.


```
    <bids>-</bids>
    <date>Jul-13 15:55</date>
    <price>$300.00</price>
    <itemdes>NEC Versa Laptop Pent. 150MHZ with MMX Tech.</itemdes>
  </record>
- <record>
    <bids>2</bids>
    <date>Jul-11 15:54</date>
    <price>$8.00</price>
    <itemdes>~NEC VERSA SERIES 8MB MEMORY UPGRADE NEW</itemdes>
  </record>
- <record>
    <bids>-</bids>
    <date>Jul-15 15:25</date>
    <price>$159.00</price>
    <itemdes>NEC VERSA 4080H P120/24MB/1GB/10.5/$159
      DUTCH</itemdes>
  </record>
- <record>
    <bids>12</bids>
    <date>Jul-15 14:35</date>
    <price>$51.00</price>
    <itemdes>NEC VERSA 5080X LAPTOP FOR PARTS..P233/13.3"</itemdes>
  </record>
- <record>
```


Fig. 3. XML output of *eBay* example

W.r.t. tree extraction, as defined in [4], an *element path definition* characterizes a path with some additional attribute requirements and is hence very similar to an *XPath* query. As an example, consider $(.\star.hr, [(size, 3), (width, .*)])$. This element path definition identifies horizontal rules (the star acts as a wildcard) of size 3 with some specified width attribute (regardless of the attribute value due to ".*"). Observe that for a simpler language definition we recently introduced the *hasProperty* predicate to express the required attributes independently, in this example e.g. with *hasProperty(X,size,3)* where X ranges over instances of $.\star.hr$. Additionally, an attribute condition might refer to a variable (see Fig. 6 in *price*) on which additional constraints are posed by some predicate.

The second extraction method, string extraction is usually applied to the decompose the content value of a leaf element. A *string definition* is characterized as regular expression in which additionally some variable references to e.g. predefined concepts might occur. As an example consider the string definitions occurring as second parameter of the *subtext* predicate in the rules defining *currency* and *amount* in Figure 6. Additionally, attribute values can be extracted using an *attribute definition*, which is simply a particular attribute designator.

The built-in predicates of the *Elog* language implement basic extraction tasks. *Elog* predicates are atoms of the form $P(t_1, ..., t_n)$ where P is a predicate name and $t_1 \dots t_n$ are either variables or constants whose values range over tree regions, string sources, constant path definition objects, and numerical arguments (such as distance parameters). The union of these classes is referred to as the *Herbrand universe* on which an *Elog* program operates.

3.2 Extraction Definition Predicates

Extraction definition predicates contain a variable which is instantiated with instances of a parent-pattern of the rule in which they occur, and based on the element path definition return instances of the new pattern. They form the basic body predicate of an extraction rule. Each *Elog* rule exactly contains one extraction definition predicate. These predicates form three different groups, depending if they extract a tree region, a string source, or a new document.

- $subelem(S, epd, X)$, $subsq(S, epd, X)$: A ground instance $subelem(s, epd, x)$ evaluates to true iff s is a tree region, *epd* an element path definition, x is a subtree in s and $root(x)$ matches *epd*. *subsq* operates similarly, but extracts tree regions (a list of elements as one extraction instance).
- *subatt(S,ad,X)*, $subtext(S, sd, X)$: The first predicate extracts from a tree region S attribute values of a given attribute definition *ad* as instances of X, whereas the second one extracts substrings which fulfill a string definition *sd*. Parent instances are tree regions in case of *subatt*, and may be both string or tree regions in case of *subtext*.
- *getDocument*(S, X), *getDocumentOfHref*(S, X) extract from a given URL as instance of S (a string source in the first case, and in the second case, an instance of a tree region whose *href* attribute is considered) a new document as a tree region.

3.3 *Elog* Rules and Patterns

Filters are represented using *Elog* extraction rules. Rules contain condition atoms which are explained in the next section.

A *standard rule* defines a component pattern X of a pattern S (thus, aggregation hierarchies can be defined). Standard rules are of the form: $New(S, X) \leftarrow Par(_, S), Ex(S, X), Cd(S, X, \ldots)[a, b], ..., [c, d]$, where *New* and *Par* are pattern predicates referring to the pattern defined by this rule and its parent pattern. S is the parent variable to be instantiated with a parent-pattern instance, X is the target variable to be instantiated with an extracted pattern instance, $Ex(S, X)$ is a tree (string) extraction definition atom, and the optional $Cd(S, X, \ldots)$ is a number of further imposed conditions on the target pattern. The extracted instances of a rule can be restricted to several intervals where $[a, b]$ expresses that the instance number a up to instance number b is considered. Additionally, a rule extracts minimal instances only.

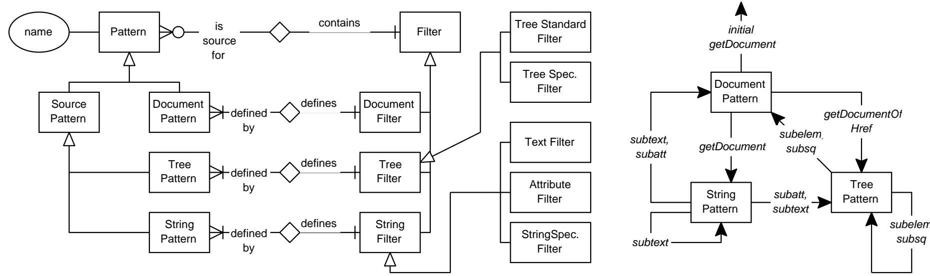

Fig. 4. Pattern Extended Entity Relationship Diagram

A *specialization rule* specializes a pattern Old to a pattern New: $New(S, X) \leftarrow Old(S, X), Cd(S, X, \ldots)[a, b], ..., [c, d]$. A *document rule* extracts new documents from given URLs. It can use below mentioned document conditions $DocCd$ as restrictions: $New(S, X) \leftarrow Par(_, S), getDocument(S, X), DocCd(S, X, \ldots)$. Document filters can also refer to relative URLs by accessing the information stored in the previous URL.

An *Elog pattern (definition)* p is a set of *Elog* rules with the same head predicate symbol. An *Elog* pattern is called homogeneous, if all its filters refer to the same parent pattern, otherwise heterogeneous. In case of a homogeneous pattern, the notion of "parent pattern" can be associated with a pattern rather than with its filters. It is forbidden to combine tree, string or document filters with each other within one pattern. The head predicate is an IDB predicate; it is visually defined with *Lixto* and named by the wrapper designer. This name is also used as default XML tag in the XML mapping [4]. Also the extracted instances of a pattern are minimized. In our current implementation, we chose to consider only those instances not contained in any other instance of the same parent-pattern instance.

Patterns (and their filters) are restricted in their use of parent patterns and pattern references as depicted on the right-hand part of Figure 4. An arc from pattern a to b indicates that the filters of a pattern of kind a can refer to patterns of kind b as parents with using the mentioned extraction definition predicate. An EER diagram illustrating relationships between patterns and filters is given in Figure 4 on the left side, and Figure 5 shows an EER diagram which illustrates filter and condition relationships in detail.

The semantics of extraction rules is very similar to the semantics of standard datalog rules. There are two possibilities to assign a semantics to an *Elog* program. The first is to define an own semantics (which exploits many similarities to Datalog), the second is to rewrite an *Elog* program (see [2]) as Datalog program and apply the standard Datalog semantics. An *Elog* program differs from Datalog in the following aspects:

- *Built-In Predicates.* In *Elog* several built-in predicates (e.g. *subelem*) are used which are restricted to a fixed input/output behavior. Moreover, constants for navigating tree regions and string sources are used.

Fig. 5. Filter Extended Entity Relationship Diagram

- *Range conditions.* A rewriting of range conditions introduces new rules containing negation and generates a stratified Datalog program. In *Elog*, range conditions are applied after evaluating the rule and by removing all instances not within the range intervals. In case an *Elog* program uses ranges and arbitrary pattern references, this intuitive meaning gets lost, and in the Datalog rewriting, the program is no longer stratified.
- *Minimizations.* By default, in *Elog* both the instances extracted by a single rule, and by a pattern are minimized, i.e. just the minimal tree regions or strings are considered. Representing this in Datalog requires additional rules and further built-in predicates such as *containedIn*.

In the current implementation of *Lixto*, full *Elog* is not yet supported, as some syntactic restrictions are made. For example, use of pattern references is limited.

3.4 Extraction Program and Pattern Instance Base

An *Elog extraction program* P is a collection of *Elog* patterns. A program can be represented via its pattern graph [2]. A directed arc is drawn from one pattern to another one, if there is at least one filter referring to the first pattern as parent. The pattern graph is a tree in case of homogeneous programs. We denote by $P(H)$ the *pattern instance base* created by evaluating all patterns of P over an HTML document H and storing them in their hierarchical order. The vertices of $P(H)$ are all pattern instances extracted from P with start document H.

```
ebaydocument(S, X) ← getDocument(S = $1, X)
ebaydocument(S, X) ← next(_, S), getDocumentOfHref(S, X)
        next(S, X) ← ebaydocument(_, S), subelem(S, (⋆.content, [(href, .∗),
                     (elementtext, (next page))]), X)
      record(S, X) ← ebaydocument(_, S), subelem(S, .table, X)
                     before(S, X, (⋆.tr, [(elementtext, . ∗ Current.∗)]), 0, 100, _, _)
                     after(S, X, (⋆.img, [(src, . ∗ spacer.gif)]), 0, 100, _, _)
     itemdes(S, X) ← record(_, S), subelem(S, (⋆.td. ⋆ .content, [(href, .∗)], X)
       price(S, X) ← record(_, S), subelem(S, (⋆.td, [(elementtext, \var[Y].∗)]), 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)
      amount(S, X) ← price(_, S), subtext(S, [0 − 9]+\.[0 − 9]+, X)
```

Fig. 6. *Elog* Extraction Program for linked *eBay* pages

There is an arc from vertex a to vertex b in $P(H)$ if and only if b is the parent-pattern instance of a. Each pattern instance is associated with a pattern name. The pattern instance base is a forest of hierarchically ordered pattern instances. Each tree corresponds to the extracted values from one particular HTML document. The pattern instance base is an intermediate data representation used by the XML translation builder to create an XML translation scheme and a corresponding XML companion.

As an example program, consider the *eBay* program in Figure 6. The pattern *ebaydocument* is a document pattern consisting of two filters with different parents. The first one refers to the starting document, which is in this case, fixed, whereas the second on follows the "next" link on each page. "$1" is interpreted as a constant whose value is the URL of the start document of a *Lixto* session. The used condition predicates are explained below.

3.5 Context and Internal Condition Predicates

Context condition predicates further restrain the instances matching an extraction definition atom based on surroundings. In Figure 7, we illustrate based on an example tree region, which nodes can be referred by context and internal conditions. By default, context conditions operate only within the current parent-pattern instance. Context conditions express that something must or must not occur before or after an instance. They can operate on tree regions or string sources, hence they use a string definition or an element path definition.

In our actual implementation, the before and after conditions are further qualified by an interval *(start,end)* of relative distance parameters expressing how far the external element may occur from the desired pattern instance to be

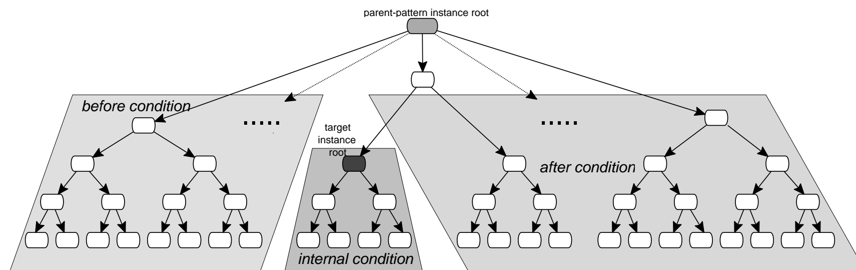

Fig. 7. Conditions

extracted. For an example of condition predicates, see the rule defining *record* in Figure 6. There, an after and a before condition are used. The first two parameters are parent-instance and target-instance variable followed by an element path definition; the next arguments indicate a minimum and a maximum distance, and the two final parameters are output variables. These are instantiated with the actual instance and the actual distance, and could be referred by further predicates.

Further supported contextual condition predicates are *above* and *below*, which make use of an additionally defined attribute designator *colno* which expresses column numbers. A ground instance $below(s, x, epd, y)$ evaluates to true iff s and x are tree regions, *epd* is an element path definition and y is a subtree of s such that $root(y)$ is matched by *epd*, and y occurs before x and has the same value for the attribute *colno*. If a designer wants to extract the third column of a table, but only starting with the fifth line, then she first defines to use *colno=3*, and then imposes the condition that it occurs under the contents as given in the fifth entry (or some attribute information of the fifth entry). *below* is similar to *after* where *epd* contains "colpos=[value]" with the difference that the value depends on the *colno* value of the target instance and vice versa. Moreover, we offer the possibility of contextual conditions which refer to elements outside the parent-pattern instance, e.g. within the "grandparent", i.e. etc. Such an extended concept is very useful together with *below/above* predicates in hierarchical extraction.

Internal condition predicates include *contains*, which is used for restricting extracted pattern instances based on properties contained in the instances themselves, i.e. in the tree regions that constitute those instances. The *firstsubtree* condition states that the first child of a tree region must contain a particular element – this is very useful for defining lists of elements, as it gives the possibility to express that the first and last child must contain some elements with specific properties.

3.6 Auxiliary Conditions and Conditions on Document Filters

Concept predicates are unary or binary relations. They refer to semantic concepts such as *isCity(X)*, expressing that the string is a city name, or syntactic ones like *isDate(X,Y)*. Predicates like $isDate(X, Y)$ can create an output variable

– e.g. for an input date x, the system returns a date in normal form y. Some predicates are built-in, however more concepts can be added to the system using the convenient *Lixto concept editor*. Moreover, for several concepts, *comparison predicates* allow to compare values (e.g. dates) such as $<(X,Y)$.

Pattern predicates of the head are the IDB predicates defined by the wrapper designer. Those of the body refer to previously created patterns. Each matched pattern instance is element of one designated pattern; for example, *price* may be a pattern name and the matched targets are its instances. Each aggregation filter contains a reference to its parent pattern. Additionally, as discussed above, further pattern references are possible. For instance, a price pattern can be constructed by imposing the constraint that immediately before a target of pattern *item* needs to occur, which is expressed in *Elog* as $before(S, X, .\star.content, 0, 1, Y, _), item(_, Y)$.

Range conditions do not correspond to predicates in the usual sense. They allow a designer to express conditions on the cardinality of the set of targets extracted with a filter based on their order of appearance in the parent-pattern instance. Such restriction intervals need not to be contiguous, as a set of intervals can be used. Negative numbers reflect counting from the opposite side.

On document filters the following conditions can be imposed: $smaller(X, v)$ requires that the size of a Web page as instance of X is smaller than some given value v in KB. $samedomain(X, Y)$ evaluates to true iff instances of X and Y (where Y is usually a constant) are URLs of the same domain. Moreover, one can specify a number v for each document pattern: If the pattern has been evaluated already for v times, then no further evaluation occurs.

3.7 Input-Output Behavior of Elog predicates

As already mentioned, one additional feature of *Elog* is that built-in predicates have an input-output behavior (adornments) that prevents them from being freely used in *Elog* rules. An extraction definition predicate for instance uses a parent-pattern variable as input, and a pattern variable as output which is used as input variable in condition predicates. An atom can be evaluated only after all its input variables are bound to effective values. The following list specifies for each argument position of each built-in predicate the type, input (i) or output (o) of variables that occur within these position. An underscore indicates that the type (input/output) is irrelevant.

$subelem(i,i,o)$	$subsq(i,i,o)$	$subtext(i,i,o)$	$subatt(i,i,o)$
$getDocument(i,o)$	$before(i,i,i,i,i,o,o)$	$notbefore(i,i,i,i)$	$below(i,i,i,o)$
$contains(i,i,o)$	$notcontains(i,i)$	$firstsubtree(i,i)$	$parentpattern(_,o)$
$isPattern(i,i)$	$isConcept(i)$	$isConcept(i,o)$	$compare(i,i)$

As an example of a unary concept, consider *isCity(i)*, and of a binary concept, consider *isDate(i,o)*. An example of a pattern reference is *price(i,i)*, and an example of a comparison condition is $<(i,i)$.

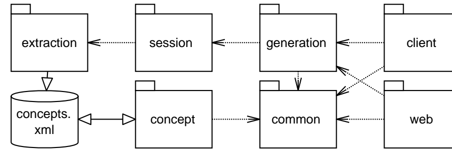

Fig. 8. Sketch of Lixto's package structure

The element path definition (string definition) is usually a constant input, but can additionally contain variables. These variables are treated as output variables. They occur in case of a reference to some concept atom (e.g. see the above example rule defining *price*). The instances of each variable that occurs in an element path definition are as usual all possible matches.

Consider the following example: $price(S, X) \leftarrow record(_, S), subelem(S, (.\star .td, [(elementtext, \backslash var[Y].*)]), X), isCurrency(Y)$. The predicate *record* is evaluated and all instances s of S are generated; they are used to evaluate the extraction definition predicate. *subelem* computes possible instances x and y of X and Y based on the given tree path. All possible substitution instances (s, x, y) are stored. After y is bound, *isCurrency(Y)*, is evaluated.

4 Implementation and Package Structure

Lixto is implemented entirely in Java. We chose to implement our own *Elog* interpreter instead of using an existing Datalog interpreter and rewriting rules into Datalog. Figure 4 gives an overview of *Lixto*'s package structure: *extraction* is the *Elog* interpreter. It is accessed by *session*, where the actual rule and condition creation is carried out. *generation* handles the pattern generation algorithm, which decides which action is followed by which step in the interactive creation process. Currently, two frontends are supported: A local *client* and a servlet frontend (*web*). The latter offers the wrapper designer the possibility to mark relevant data areas in her favorite browser, e.g. *Netscape* or *Internet Explorer*. *common* contains shared objects and message files, whereas in *concept* the syntactic and semantic concept editors (for a sample screenshot see Fig. 9) are located.

5 Ramifications, Current and Future Work

Various extensions of *Elog* are obvious such as using various forms of negation. Current theoretical research investigates the expressive power of *Elog* wrappers over unranked labeled trees. Further ramifications of *Elog* include universally quantified conditions and complement extraction, e.g. to remove advertisments of Web pages. Additionally, the framework of document navigation is being extended to also give the possibility to issue post requests. Moreover, current work is devoted to implementing consistency check alerts that can be defined in

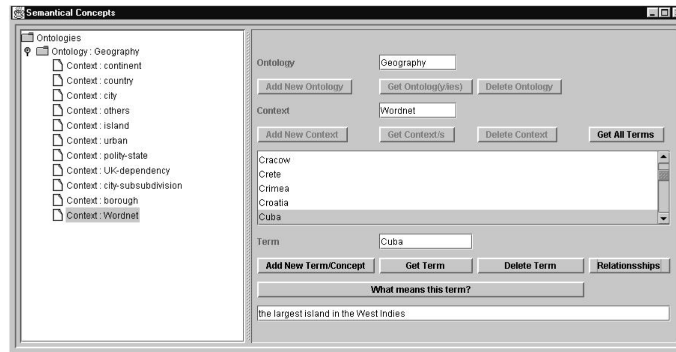

Fig. 9. Lixto's semantic concepts editor

the XML translation builder: The user is given the choice to impose a required multiplicity of an element. Based on this, an XML document type definition can be created, and moreover, warnings are given if a criterion is not satisfied on some input Web page. A *Pattern Description Language* is being developed which translates *Elog* rules into colloquial expressions. A conversion tool that transforms a subset of *Elog* programs into XSLT will be developed, hence, for a limited class of programs, simply a stylesheet and an HTML page can be used to produce an XML companion. Finally, some AI methods will be added to support the user, and *Lixto* will be embedded into the *InfoPipes* framework [5].

References

1. P. Atzeni and G. Mecca. Cut and paste. In *Proc. of PODS*, 1997.
2. R. Baumgartner, S. Flesca, and G. Gottlob. Declarative information extraction, web crawling and recursive wrapping with Lixto. In *Proc. of LPNMR*, 2001.
3. R. Baumgartner, S. Flesca, and G. Gottlob. Supervised wrapper generation with Lixto. In *Proc. of VLDB (Demonstration Session)*, 2001.
4. R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with Lixto. In *Proc. of VLDB*, 2001.
5. M. Herzog and G. Gottlob. InfoPipes: A flexible framework for M-commerce applications. In *Proc. of TES*, 2001.
6. C-N. Hsu and M.T. Dung. Generating finite-state transducers for semistructured data extraction from the web. *Information Systems*, 23/8, 1998.
7. N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proc. of IJCAI*, 1997.
8. L. Liu, C. Pu, and W. Han. XWrap: An extensible wrapper construction system for internet information. In *Proc. of ICDE*, 2000.
9. I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In *Proc. of 3rd Intern. Conf. on Autonomous Agents*, 1999.
10. B. Ribeiro-Neto, A. H. F. Laender, and A. S. da Silva. Extracting semi-structured data through examples. In *Proc. of CIKM*, 1999.
11. A. Sahuguet and F. Azavant. Building light-weight wrappers for legacy web datasources using W4F. In *Proc. of VLDB*, 1999.