The Lixto Data Extraction Project – Back and Forth between Theory and Practice

Georg Gottlob and Christoph Koch
{gottlob,koch}@dbai.tuwien.ac.at
DBAI, TU Wien
A-1040 Vienna, Austria

Robert Baumgartner and Marcus Herzog
{baumgartner, herzog}@lixto.com
Lixto Software GmbH
A-1220 Vienna, Austria

Sergio Flesca
flesca@si.deis.unical.it
D.E.I.S. – Università della Calabria
87036 - Rende (CS), Italy

ABSTRACT

We present the Lixto project, which is both a research project in database theory and a commercial enterprise that develops Web data extraction (wrapping) and Web service definition software. We discuss the project's main motivations and ideas, in particular the use of a logic-based framework for wrapping. Then we present theoretical results on monadic datalog over trees and on Elog, its close relative which is used as the internal wrapper language in the Lixto system. These results include both a characterization of the expressive power and the complexity of these languages. We describe the visual wrapper specification process in Lixto and various practical aspects of wrapping. We discuss work on the complexity of query languages for trees that was inseminated by our theoretical study of logic-based languages for wrapping. Then we return to the practice of wrapping and the Lixto Transformation Server, which allows for streaming integration of data extracted from Web pages. This is a natural requirement in complex services based on Web wrapping. Finally, we discuss industrial applications of Lixto and point to open problems for future study.

1. INTRODUCTION

Nowadays, Web content is mainly available in the form of HTML documents. Such documents do not separate data from presentation and are ill-suited for being the target of database queries and most other forms of automatic processing. This problem has been addressed by much work on so-called Web *wrappers*, programs that extract the relevant information from HTML documents and translate it into a more machine-friendly format such as XML, which can be easily queried and further processed. The wrapping problem has been addressed by a substantial amount of work (see e.g. TSIMMIS [34], FLORID [27], DEByE [24], W4F [35], XWrap [25], and Lixto [4, 3, 26] for some research systems).

Web service designers often face the task of wrapping a large number of sites. In order to provide a useful Web service, the information from a significant number of source sites relevant to the domain of the service has to be integrated and made accessible in a uniform manner. Otherwise, a Web service may fail to attract the acceptance of the users it is intended for.

Moreover, Web page layouts may be subject to frequent change. This is often intentional – to discourage screen-scraping wrapper access and to force humans to personally visit the sites.

These are just two reasons for which wrapping tools need to assist humans to render the creation of wrappers a more manageable task. Two ways to approach this requirement have been proposed: the use of machine learning techniques to create wrappers automatically from annotated examples (e.g. [23, 31]) and the *visual specification* of wrappers. The first approach currently suffers from the need to provide machine learning algorithms with too many example instances – which have to be wrapped manually – and from negative theoretical results that put a bound on the expressive power of learnable wrappers.[1] Lixto takes the alternative route, that of visual wrapper specification. By this we ideally mean the process of interactively defining a wrapper from one (or few) example document(s) using mainly "mouse clicks", supported by a strong and intuitive design metaphor. During this visual process, the wrapper program should be automatically generated and should not actually require the human designer to know the wrapper programming language. This way, the wrapper creation process requires the user to work with only very few example documents, and this work is facilitated by a simple user interface.

Visual wrapping is now a reality supported by several implemented systems [25, 35, 3], however with varying thoroughness.

In this paper, we discuss the *Lixto* visual data extraction project [26], started in 2000 and by now a commercial enterprise with an established customer base, from two perspectives, from the one of theory and from the one of practice. This project has engendered several fundamental questions that led to theoretical results which we report on in this paper. Lixto has a number of unique characteris-

PODS 2004 June 14-16, 2004, Paris, France.
Copyright 2004 ACM 1-58113-858-X/04/06 . . . $5.00.

[1] For example, it is known that even regular string languages cannot be learned from positive examples only [13].

tics by which it distinguishes itself from the state-of-the-art in Web wrapping and which would not have been possible without foundational research using results and techniques from database theory that, however, remained focussed on producing a working, and practical, industrial-strength software system. Lixto's distinctive features are summarized in the following.

- Lixto employs a fully visual wrapper specification process, which allows for a steep learning curve and high productivity in the specification of wrappers. Neither manual fine-tuning nor knowledge of HTML or the internal wrapping language is necessary.
- With Lixto, very expressive visual wrapper generation is possible: It allows for the extraction of target patterns based on surrounding landmarks, on the content itself, on HTML attributes, on the order of appearance, and on semantic and syntactic concepts. *Lixto* even allows for more advanced features such as Web crawling and recursive wrapping.
- The visual specification framework is based on an internal logic-based language similar to datalog, *Elog*.
- Elog has been closely studied. In particular, it was shown that its core fragment captures *precisely* the expressiveness of monadic second-order logic (MSO) over trees – it is therefore quite expressive – and can still be evaluated very efficiently [14].

We believe that this presents Web wrapping as a significant new application of logic (programming) to information systems. The database programming language *datalog*, which has received considerable attention from the database theory community over many years (see e.g. [1]) but has ultimately failed to attract a large following in database practice, would deserve to experience a "rebirth" in the context of trees and the Web. Indeed, for datalog as a framework for selecting nodes from trees, the situation is substantially different from the general case of full datalog on arbitrary databases. Monadic datalog over trees has very low evaluation complexity, programs have a simple normal form, so rules never have to be long or intricate, and various automata-theoretic, language-theoretic, and logical techniques exist (cf. [38]) for evaluating programs or optimizing them which are not available for full datalog.

Our work on Lixto has led us to move – as the title of the present paper suggests – many times back and forth between systems and theory research. We believe that this journey – to both ends of this spectrum – has been vital to the outcome; Lixto in the present form would not have been possible without it. It has also motivated a range of work on related research problems. Some of this work, particularly on the expressiveness and the theory of query languages over trees and XML, is summarized here.

This paper will have the character of a survey of research related to Lixto, including a discussion of how we address the integration of wrappers and the definition of complex Web services using the *Lixto Transformation Server*. We also present industrial application case studies of Lixto and discuss open research problems.

The structure of the paper basically follows this outline. We start with our logic-based framework of wrapping and discuss monadic datalog over trees – basically a fragment of Elog – as a wrapper programming language (Section 2). Next we move to visual wrapper specification and the Lixto framework (Section 3). We also discuss the Elog language in some detail (Section 3.3). Then we present our results on the complexity of queries on trees (Section 4). Finally, we come back to the practice of wrapping and give an overview of the Lixto Transformation Server (Section 5) and present some real-world applications of Lixto. We conclude with Section 7.

2. A LOGICAL VIEW OF WRAPPING

2.1 Desiderata for Wrapping Languages

To allow for a foundational study of wrapping languages, we first need to establish criteria by which to compare such languages. In [14], four desiderata were proposed that a good wrapping language should satisfy.

Such a language should

(i) have a solid and well-understood theoretical foundation,

(ii) provide a good trade-off between complexity and the number of practical wrappers that can be expressed,

(iii) be easy to use as a wrapper programming language, and

(iv) be suitable for incorporation into visual tools.

Clearly, languages which do not have the right expressive power and computational properties cannot be considered satisfactory, even if wrappers are easy to define. A few words on the "right expressiveness" of a wrapper programming language are in order here.

It is understood in the literature that the scope of wrapping is a conceptually limited one. Information systems architectures that employ wrapping usually consist of at least two layers, a lower one that is restricted to extracting *relevant* data from data sources and making them available in a coherent representation using the data model supported by the higher layer, and a higher layer in which data transformation and integration tasks are performed which are necessary to fuse syntactically coherent data from distinct sources in a semantically coherent manner. With the term wrapping we refer to the lower, syntactic integration layer.[2] Therefore, a wrapper is assumed to extract relevant data from a possibly poorly structured source and to put it into the desired representation formalism by applying a number of transformational changes close to the minimum possible. A wrapping language that permits arbitrary data transformations may be considered overkill.

The core notion that we base our wrapping approach on is that of an *information extraction function*, which takes a labeled unranked tree (representing a Web document) and returns a subset of its nodes. A wrapper is a program which implements one or several such functions, and thereby assigns unary predicates to document tree nodes.

Based on these predicate assignments and the structure of the input tree, a new data tree can be computed as the result of the information extraction process in a natural way, along the lines of the input tree but using the new labels and omitting nodes that have not been relabeled (by some form of tree minor computation):

[2] In our framework, the higher, semantic integration layer is addressed by the Lixto Transformation Server.

Given a set of information extraction functions, one natural way to wrap an input tree t is to compute a new label for each node n (or filter out n) as a function of the predicates assigned using the information extraction functions. The output tree is computed by connecting the resulting labeled nodes using the (transitive closure of) the edge relation of t, preserving the document order of t. In other words, the output tree contains a node if a predicate corresponding to an information extraction function was computed for it, and contains an edge from node v to node w if there is a directed path from v to w in the input tree, both v and w were assigned information extraction predicates, and there is no node on the path from v to w (other than v and w) that was assigned information extraction predicates. We do not formalize this operation here; the natural way of doing this is obvious.

That way, we can take a tree, re-label its nodes, and declare some of them as irrelevant, but we cannot significantly transform its original structure. This coincides with the intuition that a wrapper may change the presentation of relevant information, its packaging or data model (which does not apply in the case of *Web wrapping*), but does not handle substantial data transformation tasks. We believe that this captures the essence of wrapping.

We assume unary queries in monadic second-order logic (MSO) over trees as the expressiveness yardstick for information extraction functions. MSO over trees is well-understood theory-wise [37, 10, 8, 11] (see also [38]) and quite expressive. In fact, it is considered by many as the language of choice for defining expressive node-selecting queries on trees (see e.g. [33, 32, 14, 21]; [36] acknowledges the role of MSO but argues for *even stronger* languages). In our experience, when considering a wrapping system that lacks this expressive power, it is usually quite easy to find real-life wrapping problems that cannot be handled (see also the related discussion on MSO expressiveness and node-selecting queries in [21]).

In this section, we discuss *monadic datalog* over trees, a simple form of the logic-based language datalog, as a wrapper programming language. Monadic datalog satisfies desiderata (i) to (iv) raised above, and as we will argue, the core of the Elog language inherits this property.[3]

A monadic datalog program can compute a *set* of unary queries ("information extraction functions") at once. Each intensional predicate of a program selects a subset of dom and can be considered to define one information extraction function. However, in general, not all intensional predicates define information extraction functions. Some have to be declared as auxiliary.

2.2 Tree Structures

Trees are defined in the normal way and have at least one node. We assume that the children of each node are in some fixed order. Each node has a label taken from a finite nonempty set of symbols Σ, the alphabet[4]. We consider only *unranked* finite trees, which correspond closely to parsed HTML or XML documents. In an unranked tree, each node may have an arbitrary number of children. An unranked ordered tree can be considered as a structure

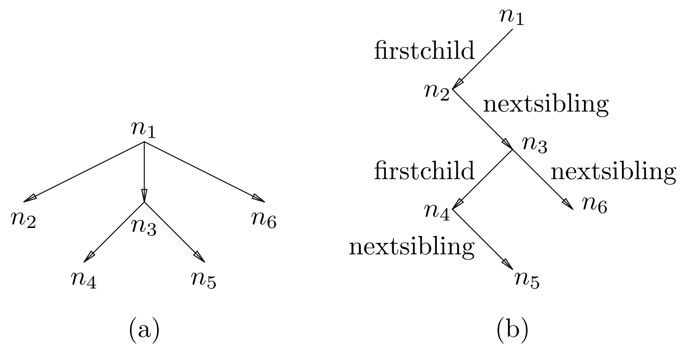


Figure 1: (a) An unranked tree and (b) its representation using the binary relations "firstchild" ($\swarrow$) and "nextsibling" ($\searrow$).

$$t_{ur} = \langle \text{dom, root, leaf, } (\text{label}_a)_{a \in \Sigma}, \text{firstchild, nextsibling, lastsibling} \rangle$$

where "dom" is the set of nodes in the tree, "root", "leaf", "lastsibling", and the "label_a" relations are unary, while "firstchild" and "nextsibling" are binary. All relations are defined according to their intuitive meanings. "root" contains exactly one node, the root node. "leaf" consists of the set of all leaves. "firstchild(n_1, n_2)" is true iff n_2 is the leftmost child of n_1; "nextsibling(n_1, n_2)" is true iff, for some i, n_1 and n_2 are the i-th and $(i+1)$-th children of a common parent node, respectively, counting from the left (see also Figure 1). $\text{label}_a(n)$ is true iff n is labeled a in the tree. Finally, "lastsibling" contains the set of rightmost children of nodes. (The root node is not a last sibling, as it has no parent.) Whenever the structure t may not be clear from the context, we state it as a subscript of the relation names (as e.g. in dom_t, root_t, ...).

By default, we will always assume trees to be represented using the schema (signature) outlined above, and will refer to them as τ_{ur}.

The *document order* relation $\prec$ is a natural total ordering of dom used in several XML-related standards (see e.g. [39]). It is defined as the order in which the opening tags of document tree nodes are first reached when reading an HTML or XML document (as a flat text file) from left to right.

2.3 Monadic Datalog

We assume the function-free logic programming syntax and semantics of the *datalog* language known and refer to [1] for a detailed survey of datalog. *Monadic datalog* [7, 14] is obtained from full datalog by requiring all intensional predicates to be unary. By unary query, we denote a function that assigns a predicate to some elements of dom (or, in other words, selects a subset of dom). For monadic datalog, one obtains a unary query by distinguishing one intensional predicate as the *query predicate*. By *signature*, we denote the (finite) set of all extensional predicates (with fixed arities) available to a datalog program. By default, we use the

[3] Elog, on the other hand supports visual features that allow to handle the most common tasks very quickly and easily. Moreover, it contains features that render it strictly more expressive than MSO.

[4] In this simple model, unrestricted sets of tags as well as string and attribute values are assumed to be encoded as lists of character symbols modeled as subtrees in our document tree.

signature τ_{ur} for unranked trees.[5]

EXAMPLE 2.1. The monadic datalog program over τ_{ur}

$$\begin{aligned} \mathit{Italic}(x) &\leftarrow \text{label}_i(x). && (1)\\ \mathit{Italic}(x) &\leftarrow \mathit{Italic}(x_0),\ \text{firstchild}(x_0, x). && (2)\\ \mathit{Italic}(x) &\leftarrow \mathit{Italic}(x_0),\ \text{nextsibling}(x_0, x). && (3) \end{aligned}$$

computes, given an unranked tree (representing an HTML parse tree), all those nodes whose contents are displayed in italic font (i.e., for which an ancestor node in the parse tree corresponds to a well-formed piece of HTML of the form $\langle i \rangle \dots \langle /i \rangle$ and is thus labeled "i"). The program uses the intentional predicate, *Italic*, as the query predicate. □

Monadic second-order logic (MSO) extends first-order logic by quantification over set variables, i.e., variables ranging over sets of nodes, which coexist with first-order quantification of variables ranging over single nodes. A unary MSO *query* is defined by an MSO formula φ with one free first-order variable. Given a tree t, it evaluates to the set of nodes $\{x \in \text{dom} \mid t \vDash \varphi(x)\}$.

The following holds for arbitrary finite structures:

PROPOSITION 2.2 (FOLKLORE). *Each monadic datalog query is MSO-definable.*

Throughout the paper, our main measure of query evaluation cost is *combined complexity*, i.e. where both the database and the query (or program) are considered variable.

PROPOSITION 2.3. (see e.g. [14]) *Monadic datalog (over arbitrary finite structures) is NP-complete w.r.t. combined complexity.*

2.4 Properties of Monadic Datalog over Trees

By restricting our structures to trees, monadic datalog acquires a number of additional nice properties. First,

THEOREM 2.4 ([14]). *Over τ_{ur}, monadic datalog has $O(|\mathcal{P}| * |dom|)$ combined complexity (where $|\mathcal{P}|$ is the size of the program and $|dom|$ the size of the tree).*

This follows from the fact that all binary relations in τ_{ur} have bidirectional functional dependencies; for instance, each node has at most one first child and is the first child of at most one other node. Thus, given a program $\mathcal{P}$, an equivalent ground program can be computed in time $O(|\mathcal{P}| * |\text{dom}|)$. Ground programs can be evaluated in linear time [29].

A unary query over trees is MSO-definable exactly if it is definable in monadic datalog.

THEOREM 2.5 ([14]). *Each unary MSO-definable query over τ_{ur} is definable in monadic datalog over τ_{ur}.*

(The other direction follows from Proposition 2.2.) Judging from our experience with the Lixto system, real-world wrappers written in monadic datalog are small. Thus, in practice, we do not trade the complexity compared to MSO (for which the query evaluation problem is known to be PSPACE-complete) for considerably expanded program sizes.

Each monadic datalog program over trees can be efficiently rewritten into an equivalent program using only very restricted syntax. This motivates a normal form for monadic datalog over trees.

DEFINITION 2.6. A monadic datalog program $\mathcal{P}$ over τ_{ur} is in *Tree-Marking Normal Form* (TMNF) if each rule of $\mathcal{P}$ is of one of the following three forms:

$$(1)\ \ p(x) \leftarrow p_0(x). \qquad (2)\ \ p(x) \leftarrow p_0(x_0),\ B(x_0, x).$$

$$(3)\ \ p(x) \leftarrow p_0(x),\ p_1(x).$$

where the unary predicates p_0 and p_1 are either intensional or of τ_{ur} and B is either R or R^{-1}, where R is a binary predicate from τ_{ur}. □

In the next result, the signature for unranked trees may extend τ_{ur} to include the "child" relation – likely to be the most common form of navigation in trees.

THEOREM 2.7 ([14]). *For each monadic datalog program $\mathcal{P}$ over $\tau_{ur} \cup \{child\}$, there is an equivalent TMNF program over τ_{ur} which can be computed in time $O(|\mathcal{P}|)$.*

2.5 Discussion

In the previous section, we have shown that monadic datalog has the expressive power of our yardstick MSO (on trees), can be evaluated efficiently, and is a *good* (easy to use) wrapper programming language. Indeed,

- The existence of the normal form TMNF demonstrates that rules in monadic datalog never have to be long or intricate.
- The monotone semantics makes the wrapper programming task quite modular and intuitive. Differently from an automaton definition that usually has to be understood entirely to be certain of its correctness, adding a rule to a monadic datalog program usually does not change its meaning completely, but *adds* to the functionality.
- Wrappers defined in monadic datalog only need to specify queries, rather than the full source trees on which they run. This is very important to practical wrapping, because this way changes in parts of documents not immediately relevant to the objects to be extracted do not break the wrapper. (That is, such wrappers are *schema-less*.)

Thus, monadic datalog over trees as a framework for Web information extraction satisfies the first three of our desiderata stated at the begin of this section (efficient evaluation, appropriate expressiveness, and suitability as a practical wrapper programming language). Only the fourth desideratum – the visual specification of wrappers – remains to be discussed. We address this issue next.

3. VISUAL WRAPPING WITH LIXTO

In this section, we discuss the Lixto Visual wrapper system. We first present its system architecture. Then we introduce the core visual specification procedure used in the Lixto wrapper generator [3, 4]. Finally, the Elog wrapping language is presented.

[5] Note that our tree structures contain some redundancy (e.g., a leaf is a node x such that $\neg(\exists y)\text{firstchild}(x, y)$), by which (monadic) datalog becomes as expressive as its *semi-positive* generalization. Semipositive datalog allows to use the complements of extensional relations in rule bodies.

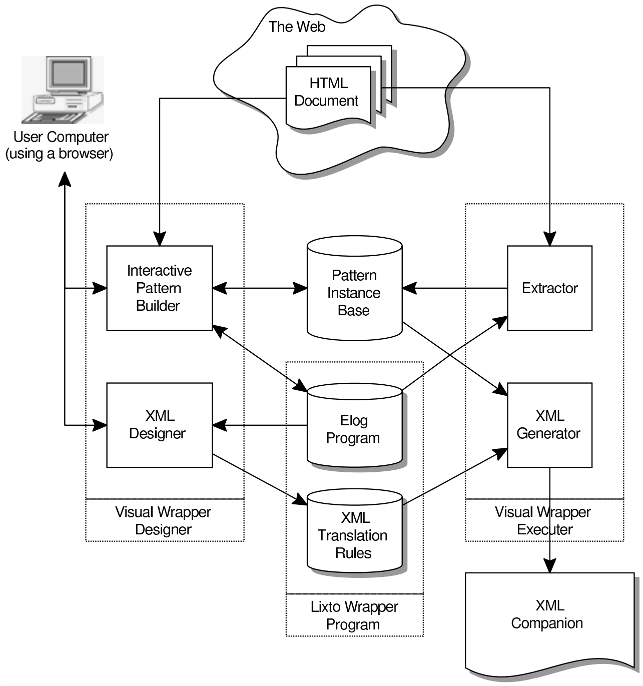


Figure 2: The Lixto Visual Wrapper System.

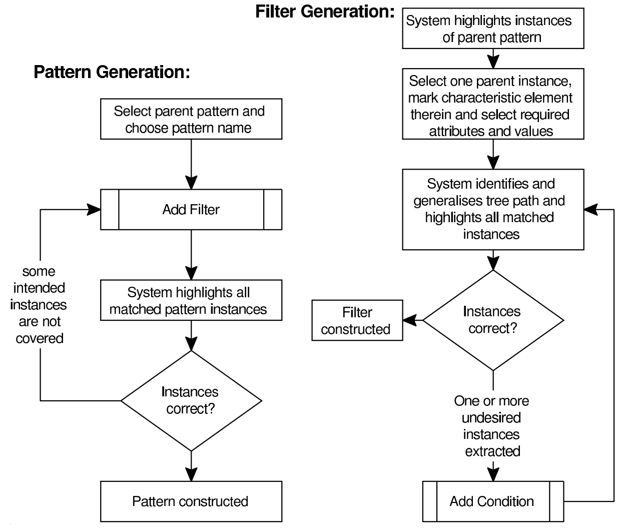

Filter Generation:
System highlights instances of parent pattern
Select one parent instance, mark characteristic element therein and select required attributes and values
System identifies and generalises tree path and highlights all matched instances
Instances correct?
Filter constructed
One or more undesired instances extracted
Add Condition
Pattern Generation:
Select parent pattern and choose pattern name
Add Filter
some intended instances are not covered
System highlights all matched pattern instances
Instances correct?
Pattern constructed

Figure 3: Creation of a New Pattern

Elog programs can be completely visually specified and are actually very similar to monadic datalog; the core of the language (called Elog$^-$ in [14] and studied there in detail) is monadic datalog as discussed before with a few minor syntactic restrictions which do not lower its expressiveness. Thus, the property that unary queries can be entirely visually specified is also inherited by MSO.

To provide a useful metaphor for the building blocks of wrappers, Lixto calls the visual counterparts of monadic intensional predicates *patterns* and those of rules *filters*.

3.1 Architecture

The *Lixto* Visual Wrapper Toolkit consists of the following modules (see Figure 2):

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

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

- With the *XML Designer*, the user chooses how to map extracted information – stored in the *pattern instance base* – to XML. This process includes the tasks of declaring some intensional predicates as auxiliary – tree nodes matching these do not necessarily propagate to the output XML tree – and of specifying which labels nodes receive based on the patterns matched. (The pattern name can act as a default node label in case a node matches only one pattern.)

- The *XML Transformer* module performs the actual translation from the extracted pattern instance base to XML.

3.2 Interactive Wrapper Generation

As discussed above, by visual wrapper specification, we refer to the process of interactively defining a wrapper from few example documents using, ideally, mainly "mouse clicks".

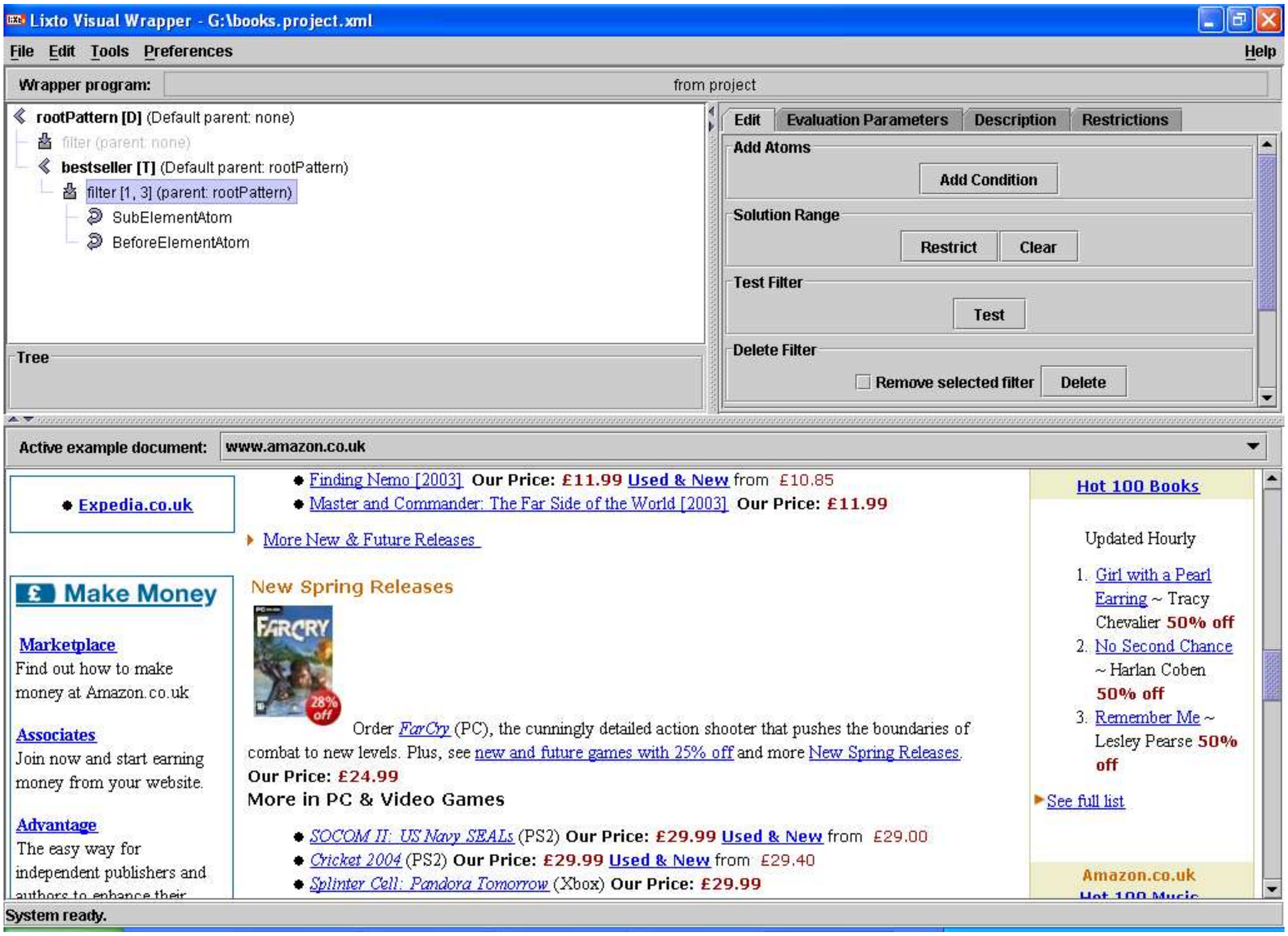
The visual wrapping process in systems such as Lixto heavily relies on one main operation performed by users: By marking a region of an example Web document displayed on screen using an input device such as a mouse, the node in the document tree best matching the selected region can be robustly determined. By selecting a reference region followed by a second region inside the former, it is possible to define a fixed path π in an example document.

Let $\text{subelem}_{a_1 \ldots a_n}(x, y)$, where $a_1 \ldots a_n \in \Sigma^*$ is a word from the labeling alphabet interpreted as a directed path in the tree, be true if, for each $1 \leq i \leq n$, the i-th node in the path from node x to y excluding x is labeled a_i. Note that "subelem" can be expressed by a fixed conjunction of child and label atoms, so we will consider it as a shortcut rather than a new built-in predicate. (Theorem 2.7 provides a method to eliminate child atoms to obtain programs strictly over τ_{ur}.) For example, $\text{subelem}_{a.b}(x, y)$ is a shortcut for $\text{child}(x, z)$, $\text{label}_a(z)$, $\text{child}(z, y)$, $\text{label}_b(y)$, where z is a new variable.

Given an example document representative for a family of documents to be wrapped, a user may be guided in the graphical specification of a rule as follows.

- First, a destination pattern p is selected from those existing or newly created and a parent pattern p_0 is selected from among the patterns defined so far. Initially, the only pattern available is the "root" pattern.

  The "root" pattern corresponds to the extensional predicate root of τ_{ur} and is the only exception to the correspondence of patterns and intensional predicates.



Wrapper program: from project

rootPattern [D] (Default parent: none)
filter (parent: none)
bestseller [T] (Default parent: rootPattern)
filter [1, 3] (parent: rootPattern)
SubElementAtom
BeforeElementAtom

□ Remove selected filter

Active example document: www.amazon.co.uk

- Expedia.co.uk

Make Money

Marketplace
Find out how to make money at Amazon.co.uk

Associates
Join now and start earning money from your website.

Advantage
The easy way for independent publishers and authors to enhance their

- Finding Nemo [2003] Our Price: £11.99 Used & New from £10.85
- Master and Commander: The Far Side of the World [2003] Our Price: £11.99

More New & Future Releases

New Spring Releases

Order FarCry (PC), the cunningly detailed action shooter that pushes the boundaries of combat to new levels. Plus, see new and future games with 25% off and more New Spring Releases.
Our Price: £24.99

More in PC & Video Games

- SOCOM II: US Navy SEALs (PS2) Our Price: £29.99 Used & New from £29.00
- Cricket 2004 (PS2) Our Price: £29.99 Used & New from £29.40
- Splinter Cell: Pandora Tomorrow (Xbox) Our Price: £29.99

Hot 100 Books

Updated Hourly

1. Girl with a Pearl Earring ~ Tracy Chevalier 50% off
2. No Second Chance ~ Harlan Coben 50% off
3. Remember Me ~ Lesley Pearse 50% off

See full list

Amazon.co.uk Hot 100 Music

System ready.

Figure 4: Program Tree View of Books Example

- The system can then display the document and highlight those regions in it which correspond to nodes in its parse tree that are classified p_0 using the wrapper program specified so far.
- A new rule is defined by selecting – by a few mouse clicks over the example document – a subregion of one of those highlighted. The system can automatically decide which path π relative to the highlighted region best describes the region selected by the user.

  This way, the rule $p(x) \leftarrow p_0(x_0),\ \mathrm{subelem}_\pi(x_0, x).$ is obtained, which the system adds to the wrapper program.
- If a filter definition is too general, the user can refine the filter rule by generalizing the path π or adding restricting conditions (e.g. a unary atom $p_1(x)$). For each such restriction, the system adds the corresponding condition atom to the filter rule. These tasks can be carried out visually as well (see [3]).

This procedure is also depicted in Figure 3.

To obtain the expressiveness of MSO, little power has to be added via conditions; one only has to be able to refer to root, leaf, and leftmost sibling nodes of the tree and to patterns via unary atoms; moreover, one has to be able to specify "nextsibling" atoms [14]. TMNF rules such as $p(x) \leftarrow p_0(x_0), \mathit{firstchild}(x_0, x)$ can then be specified by selecting a child node (say) labeled a of an instance of pattern p_0 in an example document, selecting p as destination pattern (this produces the rule $p(x) \leftarrow p_0(x_0), \mathit{subelem}_a(x_0, x)$), generalizing from the specified path a (the result is $p(x) \leftarrow p_0(x_0), \mathit{subelem}\_(x_0, x)$), and adding the condition that x has no left sibling (= is a first sibling). The Elog$^-$ fragment of Elog, discussed in detail in [14], has precisely the expressive power of MSO.

Very few example documents are needed for defining a wrapper program: It is only required that for each rule to be specified, there exists a document in which an instance of the parent pattern can be recognized and an instance of the destination pattern relates to it in the desired manner.

Figure 4 shows a screenshot for the Lixto Interactive Pattern Builder, showing a tree view of the patterns and filters already defined (top left), user interface elements to add, change, delete, or test patterns and filters (top right), and the Lixto Browser (bottom), which displays an example document by means of which patterns and filters can be visually defined. The example wrapper created here is meant to extract bestsellers from the Amazon.com site.

3.3 The Elog Language

The full Elog language extends monadic datalog as a wrapper language by a number of features. In particular, there are various forms of conditions to properly restrict filters so as to exclude "false positives" while wrapping; Elog supports string-based as well as tree-based wrapping, stratified (datalog) negation, navigation via certain forms of regular paths (optionally with so-called *distance tolerances*), and Web crawling. Many features only serve as shortcuts to simplify the wrapper specification process and to improve productivity, but some actually render the full Elog language of [3] strictly more expressive than MSO [14]. Presenting all these features in detail is beyond the scope of this paper, but a detailed overview of the full Elog language is given in [4, 3]. Some points will be discussed next.

The maybe most striking change from monadic datalog to Elog is that in the internal syntax of Elog, pattern predicates are binary. While it may seem that this invalidates our theoretical considerations regarding expressive power (and complexity) made earlier, this is in fact not true. Elog satisfies syntactic restrictions that make it in a sense monadic datalog with a dyadic syntax but basically with the favorable properties of the former (for details see [14]).

A standard Elog rule[6] is of the form

$$New(S, X) \leftarrow Par(\_, S), Ex(S, X), \Phi(S, X)$$

where S is the parent instance variable (in terms of which the filter is defined in the visual specification process), X is the pattern instance variable, $Ex(S, X)$ is an extraction definition atom, and $\Phi(S, X)$ is a (possibly empty) set of condition atoms. *New* and *Par* are pattern predicates.

In a sense, the second argument position of each pattern atom corresponds to the argument of our previously monadic pattern atoms, while the first represents its parent pattern (or the root node). The purpose of this is very practical: The binary pattern relations define a multigraph that is the basis of the transformation of the wrapped data into XML.

The *Lixto* Visual Wrapper – and thus Elog – offers two basic mechanisms of data extraction – tree and string extraction. For tree extraction, we employ the "subelem" predicate, which however allows for a richer way of specifying paths than discussed above; paths may consist of certain regular expressions over tag names and may also put conditions on the values of HTML node attributes.

The second extraction method is string-based, and allows to wrap strings at the leaves of the HTML parse tree, which do not have any further tag structure. This feature is used via a "subtext" predicate, which is analogous to "subelem" but takes a *string path definition* – a regular expression specifying which substrings of the element texts to be extracted – as a predicate instead of a path expression matching a path in the document tree.[7]

The Lixto visual wrapper supports a wide range of conditions, which allow to define many wrappers by very few and simple steps. The main types of conditions are (a) **context conditions** that express that e.g. the target pattern instance must appear before or after some specific element. (b) so-called **internal conditions** that express that some specific element must (not) appear inside the target pattern, (c) **concept conditions**, and (d) **pattern reference conditions**.

All of these can be added to a wrapper program fully visually in Lixto, without having to deal with Elog.

Context condition predicates specify that some other subtree or text must (or must not) appear before or after the desired extraction target. Compared to "nextsibling", "before" and "after" predicates are much more flexible in that they allow for nodes before or after the target pattern instance node to be arbitrarily distant, even though it is of course possible to require the paths to such nodes to match a regular expression, conditions on attributes, etc., and even the distance to be within a certain tolerance interval.

Internal condition predicates impose conditions on the internal structure of subtrees matching patterns. These include predicates for checking whether a tree contains a certain subtree or whether a node is the first among those matching a path.

Concept condition predicates subsume semantic concepts like $isCountry(X)$ or $isCurrency(X)$ (see Figure 5) and syntactic ones like $isDate(X)$, which is true if string X represents a country, currency, or date, respectively. Some predicates are built-in to enrich the system, while more can be interactively added. Syntactic predicates are created as regular expressions, whereas semantic ones refer to an ontological database. Moreover, **Comparison Conditions** such as $<(X, Y)$ allow for the comparison of data values (e.g. dates).

Finally, **pattern reference conditions** allow to add further pattern atoms to a rule, besides the "parent" pattern with respect to which each filter rule is defined in the basic visual specification procedure discussed in the previous section.

Figure 5 shows an example Elog program, which defines a wrapper for *eBay* pages. The wrapper applies to pages that contain lists of items offered for auction. Each entry in such a list contains an item description, a price with an associated currency name, and the number of bids made so far. The details of the Elog program are technical and aim to exploit HTML formatting to robustly spot the data to be extracted. At the time of writing this, on *eBay* pages, every offered item is stored in its own table. This sequence of tables is extracted with the pattern `<tableseq>`, which asks for the (largest) sequence of nodes that are children of the "body" node of the document starting with a "table" node and ending with a table node, such that the first node immediately follows the list header (which on such pages is a "table" itself, containing the text "item") and the final node is immediately followed by an "hr" HTML node. By `<record>`, we extract the individual records. The remaining patterns are all defined relative to such a record. For example, the `<itemdes>` pattern extracts item descriptions, which are nodes within the record labeled "a" – the item description field is the only one hyperlinked within a record. The pattern `<price>` uses a concept attribute, namely *isCurrency* – which matches strings like $, DM, Euro, etc. The `<bids>` pattern uses a reference to the `<price>` pattern. The final filter rule employs string extraction.

After this discussion of the practical aspects of Web wrapping and the Lixto Visual Wrapper system, another look on the theoretical side of wrapping is in place. Theorems 2.4 and 2.7 show that monadic datalog with the "child" and

[6] We moreover permit so-called *specialization rules* such as

$$greentable(S, X) \leftarrow table(S, X),$$
$$contains(X, (.td, [color, green, exact]), \_).$$

which lack the extraction atom and – rather than making a step (down) in the HTML tree, match a subset of the nodes matched by the parent pattern.

[7] There are two further analogous extraction predicates, "subsq" and "subatt", for which we refer to [4].

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

Figure 5: Elog Extraction Program for Information on eBay

"nextsibling" (und unary) relations can be evaluated in polynomial time. Full Elog introduces a number of powerful built-in predicates for navigating between two nodes in a tree, and such rules can be used to build cyclic rules of arbitrary size. This raises the question for the complexity of cyclic rules (conjunctive queries) and programs over tree relations beyond "child" and "nextsibling" (such as "before"/"following" and "descendant"), which we consider next.

4. COMPLEXITY ISSUES

We have seen in Theorem 2.4 that monadic datalog over trees defined by unary relations and the binary relations "firstchild", "child", and "nextsibling" are P-complete and can be solved in time linear in the size of the database and linear in the size of the tree.

Relations such as "child" and others such as "descendant" play an important role in various query languages on trees, such as XPath (and thus XQuery and XSLT); there, they are called *axes*. There are two main modes of navigation in trees, horizontal and vertical. For horizontal navigation, one can distinguish between navigating among sibling nodes and among nodes – intuitively – further left or right in the tree (the "following" axis in XPath). The most natural axis relations are thus *Child*, *Child**, *Child*$^+$, *Nextsibling*, *Nextsibling**, *Nextsibling*$^+$, and *Following*, where

$$\mathit{Following}(x, y) := \exists z_1, z_2\ \mathit{Child}^*(z_1, x) \wedge \\ \mathit{Nextsibling}^+(z_1, z_2) \wedge \mathit{Child}^*(z_2, y).$$

Note that if we consider complexity rather than expressiveness, we do not need to deal with relations such as *Firstchild* in addition; we may assume a unary predicate *Firstsibling* such that

$$\mathit{Firstchild}(x, y) \Leftrightarrow \mathit{Child}(x, y), \wedge \mathit{Firstsibling}(y).$$

A natural question is to ask for the complexity of monadic datalog programs over these axes, or, to start with a more basic problem, conjunctive queries (which can be seen as datalog programs containing only a single nonrecursive rule). Note that conjunctive queries over trees also have natural applications in computational linguistics, term rewriting, and data integration [18].

In the case that all individual rules are acyclic (conjunctive queries), it is known from [14] that monadic datalog over arbitrary axes can be evaluated in linear time. However, not all Elog programs have only acyclic rules.

As already observed in Proposition 2.3, while full datalog is EXPTIME-complete (c.f. e.g. to [9]), monadic datalog over arbitrary finite structures is in NP (actually, NP-complete). For a lower bound on trees, it is known [28] that already Boolean conjunctive queries over structures of the form $\langle (P_i)_i, \text{child}, \text{child}^* \rangle$ are NP-hard w.r.t. combined complexity.

A detailed study of the tractability frontier of conjunctive queries over trees is presented in full in the paper [18] in this proceedings volume. As observed there, the subset-maximal polynomial cases of axis sets are

- $\{\text{child}^+, \text{child}^*\}$,
- $\{\text{child}, \text{nextsibling}, \text{nextsibling}^+, \text{nextsibling}^*\}$, and
- $\{\text{following}\}$.

That is, for each class of conjunctive queries over a subset of one of these three sets and over unary relations, the query evaluation problem is polynomial (with respect to combined complexity). We have the dichotomy that for all other cases of conjunctive queries using our axis relations (e.g. *Child* and *Child*$^+$), the problem is NP-complete.

Obviously, the complexity of monadic datalog over a given set of axes is always the same as that of conjunctive queries over the same axes.

The special case that queries are acyclic is also worth studying, since the probably most important node-selecting query language on trees, XPath, is naturally tree-shaped.

All XPath engines available in 2002 took exponential time in the worst case to process XPath [15]. However,

Theorem 4.1 ([15]). *XPath 1 is in PTIME w.r.t. combined complexity.*

This result is based on a dynamic programming algorithm which, in an improved form [15, 17] yielded the first XPath engine guaranteed to run in polynomial time.

Most people use only the most common features of XPath, so it is worthwhile to study restrictive fragments of this language. In [15], we introduced *Core XPath*, the navigational fragment of XPath, which includes both horizontal and vertical tree navigation with axes, node tests, and boolean combinations of condition predicates. As shown there, Core XPath can be evaluated in time linear in the size of the database and linear in the size of the query. However,

Theorem 4.2 ([16]). *Core XPath is P-hard w.r.t. combined complexity.*

This property – shared by XPath, of which Core XPath is strictly a fragment – renders it highly unlikely that query evaluation is massively parallelizable (= in the complexity class *NC*, c.f. [19]) or that algorithms exist that take less than a polynomial amount of space for query processing.

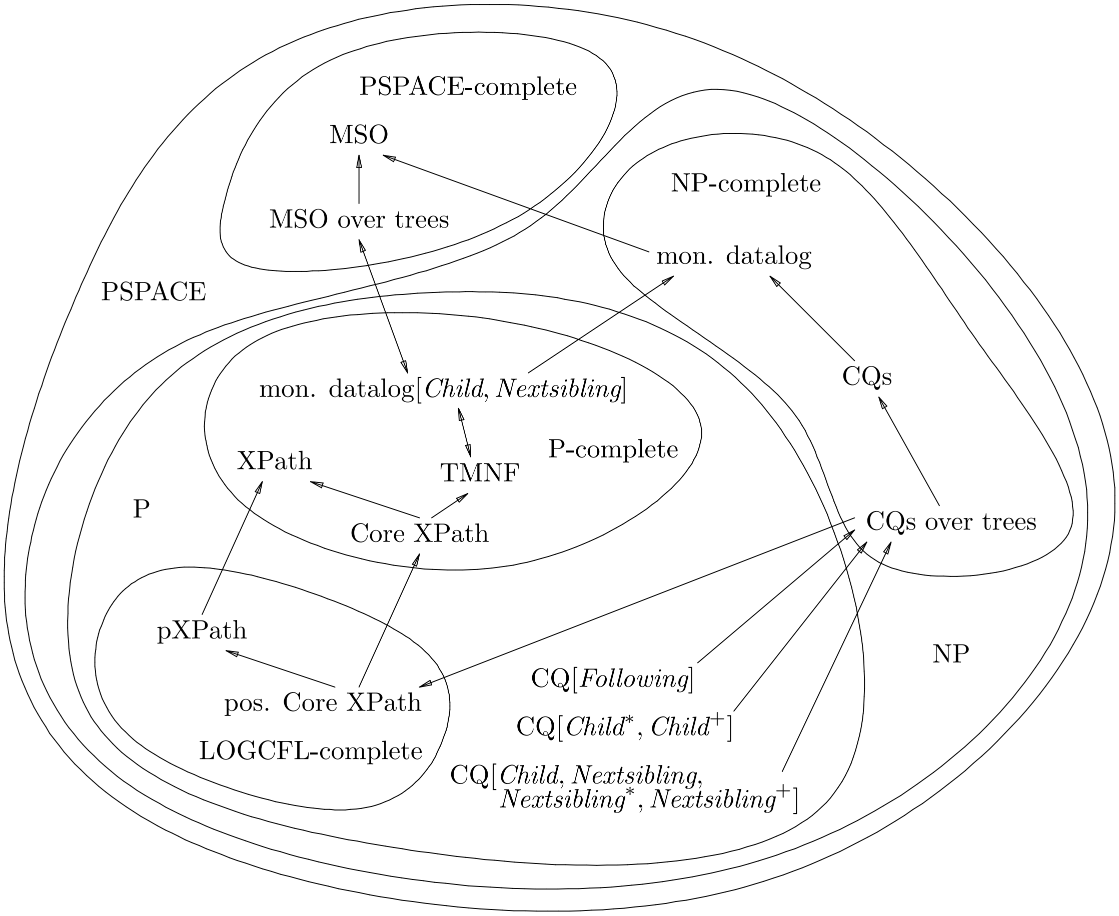


Figure 6: Complexity and expressive power of query languages over trees.

Interestingly, if we remove negation in condition predicates, the complexity of Core XPath is reduced to LOGCFL, a parallel complexity class in NC_2 [16].

THEOREM 4.3 ([16]). *Positive Core XPath is LOGCFL-complete w.r.t. combined complexity.*

This generalizes to a very large fragment of full XPath (called pXPath), from which besides negation only few very minor features have to be removed to obtain

THEOREM 4.4 ([16]). *pXPath is LOGCFL-complete w.r.t. combined complexity.*

Further results on the complexity of various fragments of XPath 1 can be found in [16].

Positive Core XPath queries correspond to acyclic positive queries over axis relations. Interestingly, each conjunctive query over axis relations can be mapped to an equivalent acyclic positive query, however there are no polynomial translations for doing this [18]. Thus,

COROLLARY 4.5. *For ever conjunctive query over trees, there is an equivalent positive Core XPath query.*

Of course, when talking about conjunctive queries over trees, we assume that all binary relations in the signature are relations from our set of axes.

Finally, Core XPath queries can be mapped to monadic datalog in linear time. The slightly curious fact here is that this remains true in the presence of negation in Core XPath (for which no analogous language feature exists in datalog.)

THEOREM 4.6 ([12]). *Each Core XPath query can be translated into an equivalent TMNF query in linear time.*

An overview of the results discussed in this section can be found in Figure 6. The Venn diagram notation refers to complexity classes (we make the usual complexity-theoretic assumptions that $LOGCFL \subset P \subset NP \subset PSPACE$) and the arrows refer to expressive power; $L_1 \rightarrow L_2$ means that each query in language L_1 can be translated into an equivalent query in L_2. The notation $L[F]$ refers to the queries of language L using only binary relations from axis set F and unary relations.

5. LIXTO TRANSFORMATION SERVER

The usual setting for the creation of services based on Web wrappers is that information is obtained from multiple wrapped sources and has to be integrated; often source sites have to be monitored for changes, and changed information has to be automatically extracted and processed. Thus, push-based information systems architectures in which wrappers are connected to pipelines of postprocessors and integration engines which process streams of data are a natural

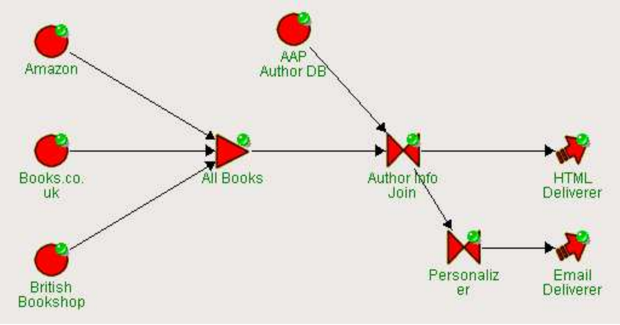


Figure 7: Small information pipeline integrating information about books.

scenario, which is supported in the Lixto suite by the *Lixto Transformation Server* [20, 6].

The overall task of information processing is composed into stages that can be used as building blocks for assembling an information processing pipeline which we call information pipe. The stages are to (1) *acquire* the required content from the source locations; this component resembles the Lixto Visual Wrapper, (2) *integrate* it, (3) *transform* it, and (4) *deliver* results to the end users.

The actual data flow within the Transformation Server is realized by handing over XML documents. Each stage within the Transformation Server accepts XML documents (except for the wrapper component, which accepts HTML documents), performs its specific task, and produces an XML document as result. This result is fed to the successor components which in turn performs the next information processing stages. Components which are not on the boundaries of the network are only activated by their neighboring components. Boundary components (i.e., wrapper and deliverer components) have the ability to activate themselves according to a user specified strategy and trigger the information processing on behalf of the user.

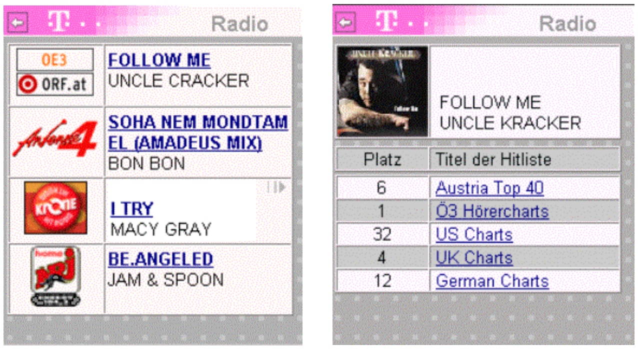
From an architectural point of view, the Lixto Transformation Server may be conceived as a container-like environment of visually configured information agents. The "pipe flow" can model very complex unidirectional information flows (see Figure 7). The use of components also modularizes information processing, so the service can be maintained and updated smoothly. Moreover, information services may be controlled and customized from outside of the server environment by various types of communication media (HTTP, SMS, RMI etc.).

6. APPLICATIONS

In this section we report on several real world applications of *Lixto*. In all these applications, the tasks of integrating, transforming and delivering information extracted using the Lixto Visual Wrapper are performed by the Lixto Transformation Server.

6.1 Mobile Applications: Now Playing!

"Now Playing" introduces a future UMTS-scenario for mobile entertainment. It was developed for the T-Mobile Future House (UMTS demonstration lab) in Vienna, Austria, using Lixto Technology. The application (Figure 8) aims at (1) monitoring the playlists of individual radio stations on PDA, (2) displaying current songs, (3) integrating current song with data from charts, and from a lyrics server.



FOLLOW ME
UNCLE CRACKER

SOHA NEM MONDTAM EL (AMADEUS MIX)
BON BON

I TRY
MACY GRAY

BE.ANGELED
JAM & SPOON

FOLLOW ME
UNCLE KRACKER

Platz	Titel der Hitliste
6	Austria Top 40
1	Ö3 Hörercharts
32	US Charts
4	UK Charts
12	German Charts

Figure 8: Now Playing

The playlists of national (Austrian) and international radio stations are taken from their web sites in real time and integrated into a portal of its own. It is possible to listen to the live audiostream of the currently played songs on mobile devices such as PDAs and view information on songs, such as titles, artists, and lyrics. Moreover, images of the CD covers are offered together with the current rankings of selected songs in five major music charts.

Data is extracted from 14 different web sites using the Lixto Visual Wrapper. These Web sites are split into three groups: radio channels, charts and lyrics. Each of these three information sources is upgraded at periodic intervals ranging from a few seconds (radio channels) up to hours or days (charts and lyrics). As soon as each module has translated its data into XML, the Lixto Transformation Server integrates the incoming XML data.

At the very end of the information flow chain is the syndication for mobile devices. The application layout is optimized to suite the small display of PDAs and at the same time offer good navigation capabilities. Starting from the main page, the end user can choose whether to view international or national (Austrian) radio stations. Each of these pages offers an index of four radio stations, and a link to a live audiostream if offered by the respective radio station. If the user asks detailed information about the currently played songs, all additional information is presented in an additional window, e.g., chart ranking in selected charts, image of the CD and lyrics.

Ranking in the charts is also independently accessible by simply clicking on one of the chart links, skimming through the returned list and selecting one of the songs. Additionally, if offered by the respective chart and supported by the mobile end device, a short intro of the song is played as well.

6.2 Flight Schedules Information

Travel – in particular flight – information services are vital to travelers around the globe. Although flight information is usually available on the Web, it is often not available at a central site for all service providers. In the case of flight information, timetables of individual flights are either scattered into different airport information systems or into the portals of individual airlines.

As a traveler is out of home by definition, this kind of information is best communicated over mobile devices.

A flight schedule information application of Lixto is presented in more detail in [6]. The user may subscribe to specific flights either by providing the flight number or the departure and destination location. The system will send the actual flight status to the user by means of an SMS message, but only if the status changed between consecutive requests.

6.3 Press Clipping: Financial News

This Lixto application, discussed in detail in [2], extracts news from various press Web sites, aggregates the extracted information with the latest stock quotes and creates a new Web site in both HTML and WML formats displaying this integrated information.

A specific feature in this scenario is the chosen XML structure of the news items. As the XML structure of every component is completely user-defined, we chose to use the standard format NITF (News Industry Text Format), which is a part of the NewsML (News Markup Language) specification. NITF and NewsML are generally used to save, exchange, and display news information. If somebody (e.g. a content provider) is running a system with NewsML, the integration of the NITF data delivered by the *Lixto Transformation Server* can be realised very easily using an additional XML deliverer.

6.4 Agrochemical Applications

A B2C application in the agrochemical domain is described in detail in [5]. We created a Viticulture Information Portal offering general vine news, vine crops growing news, localized pesticide information, new pesticides, recommendations on plant pest controls, and manufacturer news. The portal integrates the above information with weather information from various Web sources. Wine growers can personalize their information based on region and priority.

6.5 Applications in the Automotive Industry

Many business processes in the automotive industry are carried out by means of Web portal interaction. Business critical data of various divisions such as quality management, marketing and sales, engineering, procurement, supply chain management, and competitive intelligence has to be manually gathered from Web portals and Web sites. By automating this process, automotive part suppliers can dramatically reduce the costs associated with these processes while at the same time improving the speed and reliability with which these are carried out. Instead of manually browsing and searching for results on these sites, Lixto automatically gathers the data and renders the results in XML. Data in this format is then ideally suited to be processed by enterprise applications or to be distributed through various communication channels.

6.6 Business Intelligence

A further important application of Lixto is business intelligence: A typical scenario is to monitor product prices and company news offered by competitors through their Web sites. Information obtained in this way may be used to interpret changes in market share and to quickly react to changes in sales strategies of competitors.

In one specific application developed for a financial services company that provides information on ethical and responsible investment, Lixto integrates reports automatically taken from the sites of a number of organizations such as the UNO, Human Rights Watch, Greenpeace, etc., which analysts study to assess companies by their treatment of the environment, employment of children, activities in countries with high levels of corruption, and others.

6.7 Power Trading

In a further application of Lixto developed for a major electric power trader, spot market prices for electric power are integrated from major European power trading sites. This information is automatically integrated with weather and water level information and imported into the customer's information systems used for trading and risk management.

7. OPEN PROBLEMS

We conclude this paper with a list of open research problems that we have come across while working on Lixto and which have a considerable theoretical component.

- Tree wrapper learning. While a substantial amount of work has been done on automatic wrapper induction from example documents (e.g. [23, 31, 22, 30]), this approach suffers from the problem that a large number of examples are required to learn from. Visual specification could allow to guide a supervised learning process to require very few examples only. One goal is to render Lixto "more intelligent" using machine learning techniques, in order to reduce the work required from the human wrapper designer using a visual wrapping system even further.
- Data extraction from PDF. There is a substantial interest from industry in wrapping documents in formats such as PDF and PostScript. In such documents, wrapping must be mainly guided by a reasoning process over white space and Gestalt theory (It is actually quite difficult for a computer to e.g. separate the articles on the front page of a daily newpaper, even if available in machine-readable form.), which is very different from Web wrapping and will require new techniques and wrapping algorithms.

Finally, one interesting problem in the context of the complexity of queries over trees that has remained open is the tractability frontier of the complexity of conjunctive queries over trees accessible through regular expressions over basic tree relations such as "firstchild" and "nextsibling". (The XPath axes are a special case of this setting, where e.g. child = firstchild.nextsibling*.)

8. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] R. Baumgartner, S. Eichholz, S. Flesca, G. Gottlob, and M. Herzog. *Semantic Markup of News Items with Lixto.* 2003.

[3] R. Baumgartner, S. Flesca, and G. Gottlob. "Declarative Information Extraction, Web Crawling, and Recursive Wrapping with Lixto". In *Proc. LPNMR'01*, Vienna, Austria, 2001.

[4] R. Baumgartner, S. Flesca, and G. Gottlob. "Visual Web Information Extraction with Lixto". In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, 2001.

[5] R. Baumgartner, S. Flesca, G. Gottlob, and M. Herzog. "Building Dynamic Information Portals - A Case Study in the Agrarian Domain". In *Proc. IS*, 2002.

[6] R. Baumgartner, M. Herzog, and G. Gottlob. “Visual Programming of Web Data Aggregation Applications”. In *Proc. IIWeb-03*, 2003.

[7] S. Cosmadakis, H. Gaifman, P. Kanellakis, and M. Vardi. “Decidable Optimization Problems for Database Logic Programs”. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 477–490, Chicago, Illinois, USA, 1988.

[8] B. Courcelle. “Graph Rewriting: An Algebraic and Logic Approach”. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2, chapter 5, pages 193–242. Elsevier Science Publishers B.V., 1990.

[9] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. “Complexity and Expressive Power of Logic Programming”. *ACM Computing Surveys*, **33**(3):374–425, Sept. 2001.

[10] J. Doner. “Tree Acceptors and some of their Applications”. *Journal of Computer and System Sciences*, **4**:406–451, 1970.

[11] J. Flum, M. Frick, and M. Grohe. “Query Evaluation via Tree-Decompositions”. In *Proc. ICDT’01*, volume 1973 of *LNCS*, pages 22–38. Springer, Jan. 2001.

[12] M. Frick, M. Grohe, and C. Koch. “Query Evaluation on Compressed Trees”. In *Proc. LICS’03*, Ottawa, Canada, June 2003.

[13] E. Gold. “Language Identification in the Limit”. *Inform. Control*, 10:447–474, 1967.

[14] G. Gottlob and C. Koch. “Monadic Datalog and the Expressive Power of Web Information Extraction Languages”. *Journal of the ACM*, **51**(1):74–113, 2004.

[15] G. Gottlob, C. Koch, and R. Pichler. “Efficient Algorithms for Processing XPath Queries”. In *Proc. VLDB 2002*, Hong Kong, China, 2002.

[16] G. Gottlob, C. Koch, and R. Pichler. “The Complexity of XPath Query Processing”. In *Proc. PODS’03*, 2003.

[17] G. Gottlob, C. Koch, and R. Pichler. “XPath Query Evaluation: Improving Time and Space Efficiency”. In *ICDE’03*, Bangalore, India, Mar. 2003.

[18] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive Queries over Trees. In *Proc. PODS’04*, 2004.

[19] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, 1995.

[20] M. Herzog and G. Gottlob. “InfoPipes: A Flexible Framework for M-Commerce Applications”. In *Proc. TES*, 2001.

[21] C. Koch. “Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach”. In *Proc. VLDB 2003*, pages 249–260, 2003.

[22] R. Kosala, H. Blockeel, M. Bruynooghe, and J. V. den Bussche. “Information Extraction from Web Documents based on Local Unranked Tree Automaton Inference”. In *Proc. IJCAI*, 2003.

[23] N. Kushmerick, D. Weld, and R. Doorenbos. “Wrapper Induction for Information Extraction”. In *Proc. IJCAI*, 1997.

[24] A. H. F. Laender, B. Ribeiro-Neto, and A. S. da Silva. “DEByE – Data Extraction By Example”. *Data and Knowledge Engineering*, **40**(2):121–154, Feb. 2002.

[25] L. Liu, C. Pu, and W. Han. “XWRAP: An XML-Enabled Wrapper Construction System for Web Information Sources”. In *Proc. ICDE 2000*, pages 611–621, San Diego, USA, 2000.

[26] http://www.lixto.com.

[27] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. “Managing Semistructured Data with Florid: A Deductive Object-oriented Perspective”. *Information Systems*, **23**(8):1–25, 1998.

[28] H. Meuss, K. U. Schulz, and F. Bry. “Towards Aggregated Answers for Semistructured Data”. In *Proc. ICDT’01*, pages 346–360, 2001.

[29] M. Minoux. “LTUR: A Simplified Linear-Time Unit Resolution Algorithm for Horn Formulae and Computer Implementation”. *Information Processing Letters*, **29**(1):1–12, 1988.

[30] Mostrare project. www.grappa.univ-lille3.fr/mostrare/.

[31] I. Muslea, S. Minton, and C. Knoblock. “A Hierarchical Approach to Wrapper Induction”. In *Proc. 3rd Intern. Conf. on Autonomous Agents*, 1999.

[32] F. Neven and T. Schwentick. “Query Automata on Finite Trees”. *Theoretical Computer Science*, **275**:633–674, 2002.

[33] F. Neven and J. van den Bussche. “Expressiveness of Structured Document Query Languages Based on Attribute Grammars”. *Journal of the ACM*, **49**(1):56–100, Jan. 2002.

[34] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. “A Query Translation Scheme for Rapid Implementation of Wrappers”. In *DOOD’95*, pages 161–186, Singapore, 1995. Springer.

[35] A. Sahuguet and F. Azavant. “Building Intelligent Web Applications Using Lightweight Wrappers”. *Data and Knowledge Engineering*, **36**(3):283–316, 2001.

[36] H. Seidl, T. Schwentick, and A. Muscholl. “Numerical Document Queries”. In *Proc. PODS’03*, pages 155–166, San Diego, California, 2003.

[37] J. Thatcher and J. Wright. “Generalized Finite Automata Theory with an Application to a Decision Problem of Second-order Logic”. *Mathematical Systems Theory*, **2**(1):57–81, 1968.

[38] W. Thomas. “Languages, Automata, and Logic”. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 7, pages 389–455. Springer Verlag, 1997.

[39] World Wide Web Consortium. XML Path Language (XPath) Recommendation. http://www.w3c.org/TR/xpath/, Nov. 1999.