Design and Implementation of the Physical Layer in WebBases: The XRover* Experience**

Hasan Davulcu[1], Guizhen Yang[2], Michael Kifer[2], and I.V. Ramakrishnan[2]

[1] XSB Inc., Stony Brook Software Incubator,
Nassau Hall Suite 115, Stony Brook, NY 11794 USA.
`hdavulcu@xsb.com`
[2] Dept. of Computer Science, SUNY Stony Brook, Stony Brook, NY 11794, USA.
`{guizyang,kifer,ram}@cs.sunysb.edu`

Abstract. Webbases are database systems that enable creation of Web applications that allow end users to shop around for products and services at various Web sites without having to manually browse and fill out forms at each of these sites. In this paper we describe *XRover* which is an implementation of the physical layer of the webbase architecture. This layer is primarily responsible for automatically locating and extracting dynamic data from Web sites, i.e data that can only be obtained by form fill-outs. We discuss our experience in building XRover using FLORA, a deductive object-oriented system.

1 Introduction

The World Wide Web is becoming the dominant medium for information delivery and electronic commerce. The number of users who routinely use the Web to buy goods and services continues to increase at a rapid pace. In response, software robots (called "shopbots") that allow consumers to quickly find out the best prices for comparable goods and services are beginning to emerge. Information about prices and other attributes of products are typically obtained by filling out forms at a vendor's site. Software robots retrieve such information by automatically navigating to relevant sites, locating the correct forms, filling them out and extracting the data of interest from web pages returned as the result.[1] Hence tools that can do automatic form fill-outs and extract relevant information from the data pages returned in response, are becoming very important.

One such enabling technology is a *webbase* [11, 3, 2, 6], which is a database system for managing and querying the dynamic Web content (i.e., data that can only be extracted by filling out multiple forms). Designing webbases is an active

---

[1] Jango and mySimon [8] are two examples of such shopbots.

area of current database research in view of the rapid proliferation of shopbots. Managing the dynamic Web content encompasses automating several tasks that include specifying and locating the data of interest (e.g. price information) in a Web site and extracting and integrating information from multiple sites into a coherent view.

In [6] we proposed a 3-layer architecture for designing and implementing webbases — an architecture that is akin to the traditional layering of database systems. The most significant difference between a webbase and a database is the absence of the traditional physical layer. The actual data in webbases is the exclusive domain of the Web server, and the only way a webbase can access it is by filing requests to the server, such as following links or filling out forms. Hence, the notion of *virtual physical layer (VPL)* was introduced for the lowest layer in the webbase architecture in order to provide a unifying view of all the data that can be retrieved by filing requests to the server. While the physical layer in databases describes data storage, VPL specifies how to navigate to the various data sources in the Web. In this way, VPL provides *navigation independence* by shielding the user from the complexities associated with retrieving raw data from Web sources and thereby presents a database view of the Web to the upper layers of the webbase architecture, namely the *logical layer* and the *external schema layer*. While these layers are similar to the corresponding upper layers in traditional databases, they have special semantic meaning in webbases. For instance, the logical layer provides *site independence* in the sense that it integrates and reconciles heterogeneous information available from different sites, which is available through VPL in navigation-independent, but nonetheless site-specific form.

We had proposed techniques centered around Transaction F-Logic [10, 9] that facilitate creation of wrappers for the virtual physical layer [6]. Our architecture makes it possible to automate data extraction from web data sources to a much higher degree than was previously possible. But the design and implementation of the VPL itself was left open and is the subject of this paper. Specifically we describe our experience with implementing the XRover using FLORA [14], a deductive object-oriented system that we recently developed.

A case for deductive object-oriented design of VPL: The first step in the design process is to develop a suitable data model for HTML pages. Observe that an HTML page is a semistructured data source comprising several elements, each having a tag that identifies the type of the element. For instance, a tag can identify an element as a paragraph, an image, a link, a table, a form, etc. We designed a syntactic *HTML object data model* to represent the elements in a page. In this model we define an object class corresponding to every tag. The HTML page is parsed and its elements are assigned an object class based on their tags.

HTML is a display-oriented mark-up language with only limited structural capabilities. In particular, it provides no machine understandable information to describe the *contents* of a page. Hence, we also need a semantic data model so as to be able to structure the syntactic objects presented in HTML and invoke

meaningful operations on them, such as *follow a link object, fill-out a form object*, or *query the value of a certain attribute* (say, in a table). For this purpose, we designed a semantic *navigation object model*, which consists of aggregate objects that draw information from the HTML model and enable automated navigation in Web sites. In database terms, navigation objects are semantic *views* over the purely syntactic HTML objects.

The first step, converting an HTML page into a set of objects, is a relatively simple task. The crux of the VPL is the design of the navigation object model and the *mapping* between the syntactic HTML object model and the semantic navigation object model. One important issue in this design is the *resilience* of that mapping, namely, the ability of the mapping to yield correct navigation objects in the face of variations and changes in the page layout. We propose a deductive rule-based approach for locating and extracting information from objects in Web pages. Such a paradigm lets us efficiently search for objects and their associated attributes with high degree of independence from the page layout.

The rest of this paper is organized as follows: In Section 2 we provide an overview of our approach to the design of the VPL. Section 3 describes the details of the design. Section 4 discusses XRover, our implementation of VPL using FLORA,[2] a recently developed deductive object-oriented system based on F-logic [10]. Our implementation experience is discussed in Section 5.

2   Our Approach

One of the most important tasks that a shopbot must do is to collect information and services from different sites and present it to the customer in an integrated, unified view. In many shopbot sites, most of this extraction happens automatically, by "learning" regular expressions that match the desired information. The learning process is guided by a set of simple heuristics, such as those described in [12]. These techniques work well for a typical consumer site, where information is obtained by filling out a simple keyword-based search form and the result is presented in a simple, structured table. It is much more difficult to deal with sites that cater to business customers where search forms allows complex parametric queries based on multiple attributes, and results are presented in multiple related HTML segments. For instance, Figure 1 shows part of a search result page on the Web site of a large distributor of electronic components.

The page consists of three visible tables (each providing a different kind of information for the electronic part), many more invisible tables, one form to enable purchase, plus a plain text header that provides classification information for the retrieved part. Such complex result pages vary widely from site to site and, to the best of our knowledge, no automatic techniques exist for extracting and integrating such complex information. However, the semantic structure of Web pages can be *mapped* with relative ease with the help of appropriate graphical

---
[2] http://xsb.sourceforge.net/flora/

tools. The purpose of such a tool is to let the user identify (and specify to the system) the objects of interest, such as the relevant tables, forms and links. The physical virtual layer of our webbase system provides the needed infrastructure to support the process of site mapping and complex information extraction.

Pricing

Order Quantity	Unit Price
1 – 9	16.08
10 – 50	13.16
Call 1–800–PQRSTUV for pricing on other quantities.	

Add Part # **00Z787** to Shopping Cart

[Add] [1]

Component Detail

Attribute	Value
Mfg Pt no	29021
Manufacturer	LOCTITE CORP.
Available	38
Description	CHEMICALS & SOLDER~THREADLOCKERS
Price each	16.08
Min Buy	1
Cat 117 Page No.	111
Cat 116 Page No.	162
Buy Mult	1
Cat 115 Page No	160

Additional Information for Threadlockers

Link/ Download	Description	Type	Size (Byte
Download	Catalog 116 View	text/html	Not Availab
Download	Catalog Detail (117)	text/HTML	Not Availab

Fig. 1. A Complex Catalog Search Result Page

The architecture of the VPL is object-oriented and it is implemented using FLORA. The approach has two main components:

1. A general mechanism for locating objects of interest on a page; and
2. An object model for describing aggregate objects in the navigation model. Navigation objects are queried by the higher levels of the webbase.

The first mechanism is based on the *object locator language* (OLL) — a special declarative language that allows the user to specify object location in a flexible way. It is akin to the language of extended path expressions in semistructured query languages [1]. The system includes a FLORA program that acts as an interpreter for this language. Since FLORA implements F-logic, which in itself is a powerful query language for semistructured data, building such an interpreter for OLL is very easy. Thus, when the user points to an HTML object of interest, an OLL expression must be generated in order to arrange for the subsequent retrieval of the object.

The unique aspect of our approach is how these expressions are generated. First, an OLL expression for the desired object is *automatically* created. This expression is fairly simple-minded, as it specifies the location of the object in rather rigid terms. This initial expression is similar to URI's in XML: it provides a sequence of simple navigation commands that direct the search towards the requisite object. However, such expressions are not appropriate for locating Web information, because the location of an object can change due to a page redesign or simply because the page is generated dynamically, by a script. Such changes tend to break Web extraction systems so resilience cannot be achieved by rigid, brittle locator expressions. Thus, at the next stage, we transform the initial OLL expression into an *unambiguous* and *resilient* expression that extracts to the same object. Here "unambiguous" means that the expression identifies just one object; this requirement guards against the possibility of over-generalizing the initial OLL expressions. "Resilience" means that the expression will be able to locate the requisite object under a large class of variations in the page layout. Some of the techniques used to create unambiguous resilient expressions are detailed in [7].

To illustrate the idea, consider the second visible table in Figure 1 (below the "Component Detail" header). This table is actually part of a bigger, invisible table, so the initial OLL expression would be generated as follows:

$$\texttt{table}, \texttt{table.tr}, \texttt{tr.td}, \texttt{td.img}, \texttt{table}, \texttt{table}, \texttt{td},$$
$$\texttt{table}, \texttt{img}, \texttt{table}, \texttt{text}, \texttt{table}, \texttt{table}, \texttt{form}, \texttt{text}, \texttt{table}$$

The actual initial expression is much more detailed—we skipped many of the intermediate features of the Web page. In this expression, the symbols correspond to HTML objects, the period "." means that the search must nest inside a complex data element, such as a table or a form, and the comma "," signifies horizontal scan accross the siblings in the HTML tree. The above expression tells us that in order to find the second visible table, we have to find the second top-level table in the HTML source, go inside the table (nest), find the second row and then start examining the fields of that row. Having found the second field, which happens to have complex internal structure, we must nest into that structure. Then we must scan this structure horizontally to find an image, skip two tables, find an out-of-place `<td>` element (which happens to be a formatting bug on this page), and then skip a number of images, tables, and text items to locate the table we need.

The problem with this addressing schema is that it is too brittle. It will get us the desired table for a particular instance of the page, but a page generated for a different catalog search request might look slightly different and the above address might then point to a wrong item (or not point anywhere at all). This problem was addressed in [7]. Combining the techniques described in that work with other heuristics, we can create a much more resilient OLL expression:

$$*.\texttt{text}[\texttt{contents} \rightarrow' *\, \texttt{Component}*'], \texttt{table}$$

This expression says that in order to find the desired table, we must find a text object that matches the word "Component" at any level of nesting and then

scan horizontally to the first table. This expression is much more resilient to changes in the page contents than the original one, and it stands a much better chance of being able to fetch the right object regardless of the actual search parameters, even in the presence of many types of page layout changes. Not only is the above expression more resilient, but it also can be processed faster using a deductive system, such as FLORA, because we can build an index on the `contents` attribute of the text class.

The second layer in our architecture, the aggregate navigation objects that unify the information scattered in disparate HTML segments, is essentially a view over the basic HTML model of a Web page. This view is specified using the *page extraction map*, which itself is a set of F-logic objects that use the OLL expressions to tell the system where the individual components of the navigation object are coming from. Page extraction maps are composed together to form a *site map* for the Web site.

The page extraction map object corresponding to the second table in the above example looks as follows:

```
oll(*.text[contents -> '*Component*'],table) : normal_table[
        column_names -> rel_oll( .tr(1) );
        init_row -> 2;
        row(Row) -> rel_oll( .tr(Row) );
        total_rows -> rel_oll( .last )
].

oll(*.text[contents -> '*Component*'],table.tr(1))
: header_row[
        column_name(Column) -> rel_oll( .th(Column).text );
        width -> rel_oll( .last )
].

oll(*.text[contents -> '*Component*'],table.tr(Row))
: data_row[
        column(Column) -> rel_oll( .td(Column).text );
        width -> rel_oll( .last )
].
```

The above specifies that the HTML segment pointed to by

$$*.\texttt{text}[\texttt{contents} \rightarrow' * \texttt{Component}*'], \texttt{table}$$

is a *normal_table* in the navigation object model and its header column can be extracted from the segment pointed to by the OLL expression

$$*.\texttt{text}[\texttt{contents} \rightarrow' * \texttt{Component}*'], \texttt{table.tr(1)}$$

where `tr(1)` means the first row in the table. The rows can be extracted from the segment pointed to by $*.\texttt{text}[\texttt{contents} \rightarrow'*\texttt{Component}*'], \texttt{table.tr(Row)}$ where *Row* is a parameter. The second extraction map object is interpreted similarly.

Some of the navigation objects extracted with the help of this extraction map object are as follows:

```
nav_obj3     : normal_table.      nav_obj4     : header_row.
nav_obj3[                         nav_obj4[
   column_names -> nav_obj4;        column_name(1) -> 'Attribute';
   row(1) -> nav_obj5;              column_name(2) -> 'Value';
   row(2) -> nav_obj6;              width -> 2
   ...                            ].
   total -> 10
].


nav_obj5     : data_row.          nav_obj6     : data_row.
nav_obj5[                         nav_obj6[
   column(1) -> 'Mfg Pt No';        column(1) -> 'Manufacturer';
   column(2) -> '29021';            column(2) -> 'LOCTITE CORP.';
   width -> 2                        width -> 2
].                                ].
```

We discuss navigation objects in further detail in subsequent sections.

3   Architecture of the Virtual Physical Layer (VPL)

There are two aspects in VPL implementation:

1. *Site Mapping* which is done *once* per site.
2. *Run-time query processing* driven by the site maps.

We first explain the process of site map construction.

Site Map Construction. The process is shown in Figure 2. XRover begins by using an HTTP library to fetch the Web page. This page is then parsed by an HTML parser that translates the page into a set of F-Logic objects, and for each object its OLL expression is computed. For instance, the following objects describe two consecutive tables in an HTML page:

```
htmlobj3 : table.                htmlobj11    : table.
htmlobj3[                        htmlobj11[
   parent -> htmlobj2;              parent -> htmlobj2;
   position -> 0;                   position -> 1;
   rows -> htmlobj4;                rows -> htmlobj12;
   oll -> [table];                  oll -> [table,table];
   width -> 640;
   border -> 0
].                               ].
```
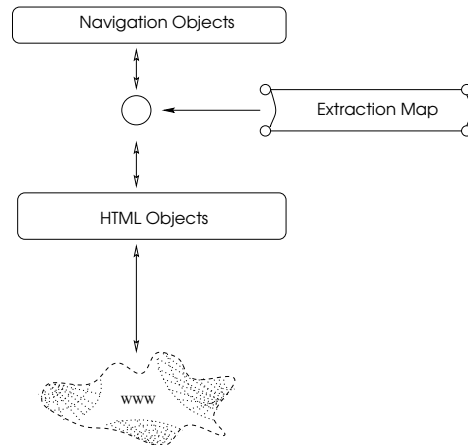
Fig. 2. Site Map Construction

Next, a page extraction map is created using a graphical editor by drag and drop operations on the HTML object tree, such as the one illustrated in Figure 3. The oll's in the page map are then optimized and made more resilient as described in Section 2. This process is repeated for every page of interest, including those dynamically generated by scripts.

Extraction maps for individual pages are put together to form a *site map*, which encodes all access paths to the data of interest. A site map can be viewed as a labeled directed graph where the nodes represent the extraction map objects, and the labeled edges represent possible actions on the navigation objects (i.e., following a link or filling out a form) that can be executed from that page.

The overall structure of a site-map for an electronics catalog could be as depicted in Figure 4.

The above represents a simple site map with three nodes: `page1`, `page2` and `page3`. There is an edge from `page1` to `page2` labeled `table(2).tr(2).td(2).form(1)` which corresponds to a form invocation. The `items` attribute of the form in the page extraction map would describe its queriable attributes. Also, there is an edge from `page2` to `page3` with the label `table(2).tr(1).td(1).a.action` which represents a link that could be followed to retrieve additional part information such as the information presented in Figure 1.

Runtime query processing. The purpose of this sub-system is to automatically extract data in response to user queries. Its overall operation is as depicted in Figure 5. When a user query arrives, the *navigation planner* determines the sequences of pages to be followed using the site maps and navigation objects that are needed to answer the user's query. It then constructs a *navigation plan* for the query. For example, if the user just requests pricing information for an
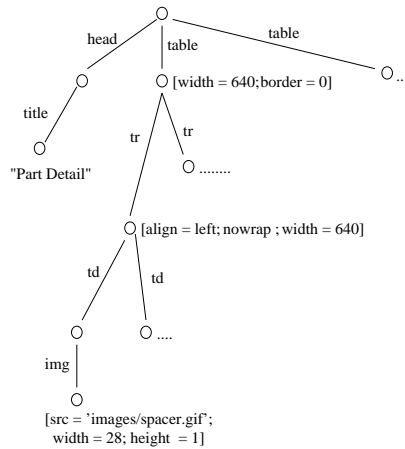
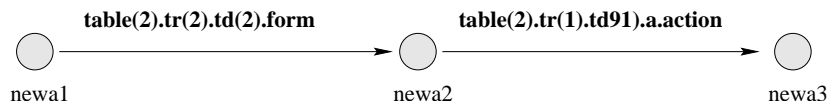**Fig. 3.** An HTML parse-tree

Fig. 4. A simple site map.

electronic part then only the pricing table needs to be extracted from the HTML page of Figure 1.

Next, the *navigation plan* is passed to the *plan evaluator* which accesses the actual Web pages. It parses and translates page contents into FLORA HTML objects. From these objects, the *Extractor* module extracts the navigation objects of interest using the page extraction map, and the cycle repeats until the entire query plan evaluation is completed. The resulting navigation objects are returned to the user or to the higher levels of XRover.

4   XRover Implementation: Status and Statistics

XRover was built using FLORA, a deductive object-oriented system implemented through source-to-source translation into XSB [13], which is a fast deductive engine that is based on tabled resolution approach. This technique is known to produce fast executable code and combines the advantages of top-down and bottom-up query processing.

The overall system is about 1,500 lines of FLORA code and it took less than two man months to implement. The average size of a site map is under 100 lines.

We also developed a graphical site-mapping tool to facilitate the job of building site maps. Using this tool one can construct the extraction map for a page
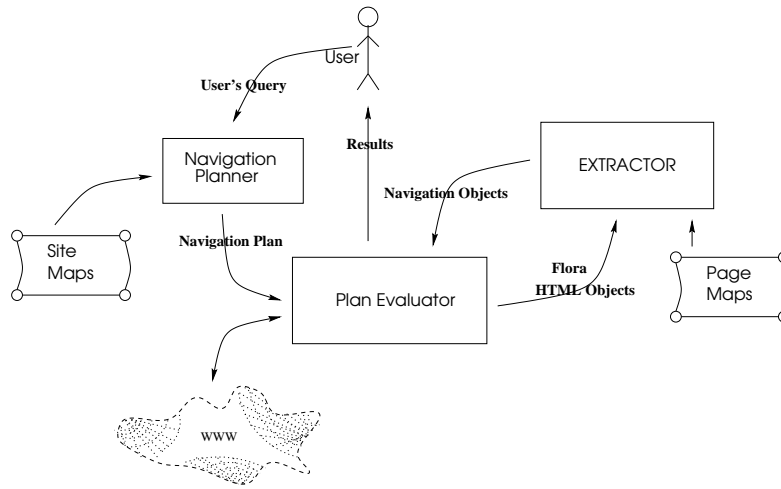
Fig. 5. Run-time query processing.

by dragging and dropping the relevant objects from the HTML parse tree. The site-mapping GUI was written entirely in Java using the Swing library.

Even though both FLORA and XRover are just prototypes, we observed that the system has very acceptable performance. A typical number of XRover accesses per Web site is three pages, and results are returned within 3 seconds. In most cases, the response time is dominated by network delays.

So far we have built two applications with XRover – a direct mail marketing service for a large pharmaceutical company and a electronic parts portal for the U.S. Defense Logistics Agency. In the former we extract names and addresses of potential customers from phone directories posted at the web sites of various medical institutions. The parts portal provides price, availability and technical data of electronic parts from various vendor and OEM catalogs on the Web.

5 Conclusion

We described the object-oriented architecture of the virtual physical layer of XRover, a Web based information system that presents a unified database view over multiple Web sites. The implementation was done using FLORA — a DOOD system based on F-logic. The experience gained during the course of this project is perhaps the most interesting part, because DOOD systems are still rare and there are not that many applications developed with them. As expected, the use of a high-level DOOD language significantly reduced the implementation effort: the entire VPL layer was implemented in less than two man-months. Despite the fact that both XRover and the FLORA programming environment are just prototypes, the performance is quite acceptable — about 3 seconds per site — and further significant optimizations are possible.

The support for object-oriented design in FLORA was crucial for helping us produce clear and concise data models at several levels of detail, and the deductive nature of FLORA made it easy to implement the query evaluator and the various interpreters (*e.g.*, the OLL interpreter). The high-level, declarative nature of FLORA made it easy to glue the various pieces of the system together.

Even more important is what was learned about the shortcomings of FLORA as implementation platform. First, a practical DOOD system requires a good module system that can simplify the task of developing multi-file projects. Second, the current implementation of FLORA relies on the underlying XSB system to do most of the optimization. It has been realized that significant speedup can be achieved through F-logic-specific source transformation techniques [14]. Finally, we discovered that DOOD systems need better support for declarative update primitives, such as the ones proposed for Transaction Logic [4, 5]. These features are being added in the upcoming implementation of FLORA 2.0 [14].

Acknowledgement. The authors would like to thank the anonymous referees for their helpful comments.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann, San Francisco, CA, 2000.
2. J.L. Ambite, N. Ashish, G. Barish, C.A. Knoblock, S. Minton, P.J. Modi, I. Muslea, A. Philpot, and S. Tejada. Ariadne: A system for constructing mediators for internet sources. In *Proc. of SIGMOD*, 1998.
3. P. Atzeni, A. Masci, G. Mecca, P. Merialdo, and E. Tabet. Ulixes: Building relational views over the web. In *Proc. of ICDE*, page 576, 1997.
4. A.J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.
5. A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.
6. H. Davulcu, J. Freire, M. Kifer, and I.V. Ramakrishnan. A layered architecture for querying dynamic web content. In *ACM SIGMOD Conference on Management of Data*, June 1999.
7. H. Davulcu, G. Yang, M. Kifer, and I.V. Ramakrishnan. Computational aspects of resilient data extraction from semistructured sources. In *ACM Symposium on Principles of Database Systems*, May 2000.
8. http://www.jango.com. Jango Corporation.
9. M. Kifer. Deductive and object-oriented data languages: A quest for integration. In *Int'l Conference on Deductive and Object-Oriented Databases*, volume 1013 of *Lecture Notes in Computer Science*, pages 187–212, Singapore, December 1995. Springer-Verlag. Keynote address at the 3d Int'l Conference on Deductive and Object-Oriented databases.
10. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42:741–843, July 1995.
11. G. Mecca, P. Atzeni, A. Masci, P. Merialdo, and G. Sindoni. The araneus web-base management system. In *Proc. of SIGMOD*, pages 544–546, 1998.

12. M. Perkowitz, R.B. Doorenbos, O. Etzioni, and D.S. Weld. Learning to understand information on the internet: An example-based approach. *Journal of Intelligent Information Systems*, 8(2):133–153, March 1997.

13. K. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In *ACM SIGMOD Conference on Management of Data*, pages 442–453, New York, May 1994. ACM.

14. G. Yang and M. Kifer. FLORA: Implementing an efficient dood system using a tabling logic engine. in preparation, 2000.