A Layered Architecture for Querying Dynamic Web Content

Hasan Davulcu*
University at Stony Brook
davulcu@cs.sunysb.edu

Juliana Freire
Bell Labs
juliana@research.bell-labs.com

Michael Kifer†
University at Stony Brook
kifer@cs.sunysb.edu

I.V. Ramakrishnan‡
University at Stony Brook
ram@cs.sunysb.edu

Abstract

The design of *webbases*, database systems for supporting Web-based applications, is currently an active area of research. In this paper, we propose a 3-layer architecture for designing and implementing webbases for querying dynamic Web content (i.e., data that can only be extracted by filling out multiple forms). The lowest, *virtual physical layer*, provides *navigation independence* by shielding the user from the complexities associated with retrieving data from raw Web sources. Next, the traditional *logical layer* supports *site independence*. The *top layer* is analogous to the external schema layer in traditional databases.

Within this architectural framework we address two problems unique to webbases — retrieving dynamic Web content in the virtual physical layer and querying of the external schema by the end user. The layered architecture makes it possible to automate data extraction to a much greater degree than in existing proposals. Wrappers for the virtual physical schema can be created semi-automatically, by asking the webbase designer to navigate through the sites of interest — we call this approach *mapping by example*. Thus, the webbase designer need not have expertise in the language that maps the physical schema to the raw Web (this should be contrasted to other approaches, which require expertise in various Web-enabled flavors of SQL). For the external schema layer, we propose a semantic extension of the universal relation interface. This interface provides powerful, yet reasonably simple, ad hoc querying capabilities for the end user compared to the currently prevailing "canned" form-based interfaces on the one hand or complex Web-enabling extensions of SQL on the other. Finally, we discuss the implementation of the proposed architecture.

1 Introduction

The trend of using the World Wide Web as the medium for electronic commerce continues to grow. Web users need to obtain information in ways that cannot be directly accomplished by the current generation of Web search engines. It is typical for a user to obtain information by filling out HTML forms (*e.g.*, to retrieve product information at a vendor's site or classified ads in newspaper sites). This process can become rather tedious when users need to make complex queries against information at multiple sites, *e.g.*, *make a list of used Jaguars advertised in New York City area, such that each car is a 1993 or later model, has good safety ratings, and its selling price is less than its Blue Book value*. Answering such complex queries is quite involved, requiring the user to visit several related sites, follow a number of links and fill out several HTML forms. Thus the problem of developing tools and techniques for creating web-based applications that allow end users to shop around for products and services on the Web without having to tediously fill out multiple forms manually, is both interesting and challenging. It is also of considerable importance in view of a recent survey that contends that 80% of all the data in the Web can only be accessed via forms [?].

Not surprisingly, the design of database systems for managing and querying data on the web, called *webbases* (*e.g.*, in [?]), is an active area of current database research. A significant body of research covering a broad spectrum of topics including modeling and querying the web, information extraction and integration continues to be developed (see [?] for a survey). Nevertheless research on the design of tools and techniques for managing and querying the *dynamic Web content* (*i.e.*, data that can only be extracted by filling out one or more forms) is still in a nascent stage.

There are several problems in designing webbases for dealing with dynamic Web content. Firstly, there is the problem of navigation complexity. For instance, while there has been a number of works that propose query languages for Web navigation [?, ?, ?, ?], they are only beginning to address the difficult problem of querying sites in which most of the information is dynamically generated. Navigating such complex sites requires repeated filling out of forms many of which *themselves* are dynamically generated by CGI scripts as a result of previous user inputs. Furthermore, the decision regarding which form to fill out next and how, or which link to follow might depend on the contents of a dynamically generated page.

Secondly, given the dynamic nature of the web, in

*This work was done while the author was at Bell Labs.

†M. Kifer was supported in part by NSF grant IRI-9404629.

‡This work is partially supported by the NSF grants IRI-9404629, CCR-9705998, 9711386, 9404921, CDA-9504275, 9303181, INT-9600598

order to build a practical tool to retrieve dynamic content from Web sites, one needs to devise automatic ways to extract and maintain navigation processes from the site structure. Lastly, once navigation processes have been derived, one needs to query the information they represent. Although traditional databases also provide sophisticated query languages, such as SQL or QBE, these interfaces are rarely exposed to the casual user, since they are still considered too complex. Naive users are usually given canned queries needed to perform a set of specific tasks. These canned interfaces served well in the case of fairly structured corporate environments, but they are too limiting for the wide audience of web users. A webbase would certainly benefit from a query language that is flexible enough to support interesting types of ad-hoc querying and yet is simple and natural to use.

To address these problems, we propose a layered architecture, analogous to the traditional layering of database systems, for designing and implementing webbases for querying dynamic Web content. In our architecture, the lowest layer, which we call the "virtual physical layer", provides *navigation independence* because it shields the user from the complexities associated with retrieving data from raw web sources. Next up, the "logical layer", which is akin to the traditional logical database layer, provides *site independence*. Finally, the external schema layer is functionally analogous to the corresponding layer in traditional databases.

This analogy in terms of layering allows us to focus on developing techniques for problems that are unique to webbases, and for problems that are common to both webbases and traditional databases we can directly use the already known techniques. Based on the databases analogy, we can readily identify that the problem of mapping the logical to the physical layer in traditional databases is similar to what needs to be done in webbases with respect to the corresponding logical and the virtual physical layer. Thus all of the techniques developed in traditional databases for this mapping, such as schema integration and mediators, can all be directly applied to webbases.

On the other hand, retrieving the dynamic Web content in the virtual physical layer is a problem unique to webbases. Unlike the physical layer, in traditional databases we have no control over the data sources in the web. Automating retrieval of data from such sources, especially those generated by forms, is difficult. Similarly, there are important differences at the external schema layer. Indeed, Web users form a far larger audience and generally with much wider variation of skill levels than corporate databases users. For them, traditional query languages, such as SQL, are too complex. At the same time, the diverse nature of the audience makes it difficult to prepare satisfactory canned queries in many areas. Also, preparing canned interfaces for each domain is expensive. Therefore, it is desirable to have a query interface that permits both ad hoc querying and is simple to use.

In brief, our approach to both of the above problems is as follows. Mapping the relational schema onto the raw Web requires a calculus or algebra of some sort, which could be used to specify navigation expressions that "populate" the schema with data. This part is not new as other projects attempted the same [**?**]. However, these approaches have shortcomings. The webbase designer is required to have expertise in the underlying calculus, which is usually some web-enabling extension of SQL or relational algebra. Reported experiments [**?**] suggest that users resist this idea, because the underlying navigation languages are hard to master. In addition, given that Web sites change frequently, maintaining manually generated navigation expressions can be an arduous task.

What is different in our approach is that by separating the virtual physical layer from the logical layer we can create navigation expressions *semi-automatically*, through an interactive process that does not require the user to have any expertise in the formalism underlying the navigation calculus, and the webbase designer does not even need to see what the navigation expressions look like. To support such degree of automation and be able to represent complex navigation processes, the underlying formalism must have these properties:

- It must be high level and declarative, because it is much easier to create high-level specifications of navigation processes.
- It must be compatible with the formalism that underlies databases query languages (*i.e.*, with relational calculus), so that it is possible to compose user queries with navigation expressions in order to create a single expression that would ultimately fetch the desired answer to the query. This is akin to the process of answering queries against views, where view definition is substituted into the query. If the resulting expression is still part of some declarative formalism, then the entire query can be optimized using techniques that are akin to relational algebra transformations (but we do not discuss such techniques here).
- Due to the nature of the processes being modeled, the navigation calculus must support procedural and declarative in the same formalism. For instance, at a high level, the calculus should support statements such as "do this after doing that" or "do this provided that".
- The high-level specification formalism must be object-oriented. Since Web navigation has to deal with complex structures, such as web pages and forms, in a declarative environment, these structures are best represented as objects.
- Navigation calculus expressions should be *executable* specifications themselves.

In our system, we chose a subset of Transaction F-logic [**?**], which to the best of our knowledge, is the only language that supports all the above features in a uniform fashion. Transaction F-logic is an amalgamation of two other well-known formalisms: F-logic [**?**] and Transaction Logic [**?**]. Although our navigation calculus is much more

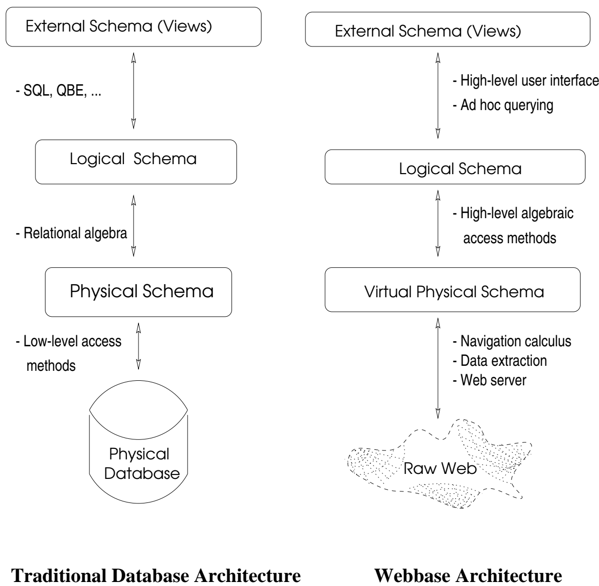

Figure 1: Traditional database architecture vs. webbase architecture

powerful (and complex) than other proposed languages for Web navigation, the Web designer does not need to know anything about it. Our approach makes it possible to create all necessary wrappers for the virtual physical schema semi-automatically, by simply asking the webbase designer to navigate through the sites of interest. We call this approach *mapping by example*. The virtual physical layer and the navigation calculus are described in Sections 3 and 4.

For the external schema layer, we propose a semantic extension of the universal relation interface [**?**, **?**], which we call *structured universal relation*. We argue that this interface provides powerful, yet reasonably simple ad hoc querying capabilities for the end user (e.g., a Web shopper) compared to the currently prevailing canned, form-based interfaces on the one hand and complex web-enabled extensions of SQL on the other. The external schema layer is described in Section 6.

Apart from the aforesaid sections, Section 2 introduces our layered architecture. Section 5 discusses the problems associated with the logical layer of a webbase; our implementation effort of the proposed architecture is described in Section 7; related work appears in Section 8, and concluding remarks in Section 9.

2 Architecture for the WebBase

The most significant difference between a webbase and a database is the absence of the physical level in the traditional sense. Indeed, actual data is the exclusive domain of the Web server, and the only way the webbase can access the data is through filing requests to the server by following links or by filling out forms.

Therefore, we introduce the notion of the *virtual physical database schema (VPS)*, which represents all the data there is to see by filing requests to the server. In many cases, the virtual physical layer cannot be constructed completely (or we might never know whether the known part of the VPS is complete). While the role of the physical layer in databases is to describe data storage, the role of VPS in webbases is to specify how to navigate to the various sources of information in the web. In this way, VPS provides *navigation independence* for webbase systems and presents a database view of the Web to the upper layers of the webbase. In this paper, we use the relational model to represent data in webbases. More details on the VPS layer appear in Sections 3 and 4.

We remark that, since the main focus in this paper is querying and navigation, we do not discuss updates and methods for data extraction from HTML pages. To the best of our knowledge, the former issue has not received much attention, while the latter has been researched extensively.

At the VPS layer, data collected from different sources resides in different relations, thus semantic and representational discrepancies are likely to exist between these relations. For instance, prices could be represented using different currencies and semantically identical attributes can have different names. These differences are smoothed out at the *logical layer* of the webbase architecture, which provides *site independence*, *i.e.*, independence from the specifics of the data sources that supply data to the webbase. Further details on the logical schema are presented in Section 5. We should note that resolution of semantic and representational differences between sites is *not* the subject of this paper. There is a vast body of research dedicated to this topic, and we could use the techniques developed there.

The top level in the webbase architecture is the *external schema layer*, which targets specific application domains (*e.g.*, used car ads, computer equipment, etc.) and is supported by a user interface that permits a high degree of ad hoc querying by naive Web users. As mentioned earlier, for such users traditional query interfaces are either too complex (SQL, QBE) or too rigid (canned and form-based). Thus, we need a query language that is flexible enough to support interesting types of ad-hoc querying and yet is simple to use. In search of such a language, we resurrected the *Universal Relation (UR)* query interface. The details of our implementation of the UR are presented in Section 6.

The following example illustrates the distinctions among different levels of abstraction in a dynamic webbase.

Example 2.1 (Used Cars) A webbase for used car shopping in the metropolitan New York area might access the several sites, such as newspapers (Newsday[1] and New York Times), new car buying services (Car Point and Auto Web), blue book price references (Kelly's), reliability information (Car and Driver) and finance (Car Finance). We present a possible set of VPS relations that can be extracted from these sites.[2] To make the tables more compact, we use Car as a

[1] Newsday is a regional newspaper with circulation on Long Island and New York City.

[2] Although this example describes these sites fairly accurately, for illustration purposes we introduce simplifi cations as well as bring in

shorthand for the attributes $Make, Model, Year$.

Table 1 shows examples of VPS relations for various Web sites. The first line in the table illustrates that data for the Newsday's site might be presented in multiple hyper-linked pages, and depending on the user's request, data extraction might require navigating multiple pages. *e.g.*, *newsday* and *newsdayCarFeatures*.

The logical level relations for our webbase and their associated relational schemas are presented in Table 2, along with the corresponding mappings to the VPS layer.

The external schema layer is represented by the following universal relation, UsedCarUR, which contains the union of all the attributes of the logical layer:

`UsedCarUR(Car,Price,Features,Contact,`
`BBPrice,Safety,ZipCode,Duration,Rate)`

The mapping between external and the logical layer in the Universal Relation model is a rather subtle issue. In Section 6, we show that the known approaches (*e.g.*, [?]) are not suitable for Web applications and discuss a possible solution.

Now, the query posed in Section 1, "*make a list of used Jaguars advertised in New York City area sites such that each car is a 1993 or later model, has good safety ratings, and its selling price is less than its Blue Book value*," can be expressed against our webbase as follows:

`UsedCarUR(jaguar,Mdl,Year,Price,Featrs,`
`Contact,BBPrice,good,ZipCode,Duration,Rate),`
`Year` $\geq$ `1993,BBPrice` $>$ `Price` □

3 Virtual Physical Schema

An important difference between webbases and traditional databases is that webbases do not control the physical data and they are limited in the ways this data can be retrieved.

Given a virtual physical schema (VPS) for a relation, the corresponding data can usually be obtained only by filling out a form, which requires that the user specify values for a certain selection of attributes, some of which might be mandatory and some optional. In fact, there might be several alternative sets of optional/mandatory attributes per relation that limit the scope of data to be retrieved.

In addition, we must specify the navigation process that needs to be executed in order to get the data. This process is represented using *Navigation Calculus*, which is described in the next section. Therefore, for each relation schema, R, in the VPS layer, there is a quadruple, called a *handle*, represented as follows:

H = ⟨*mandatory-attrs, selection-attrs, R, expression*⟩

The set of *mandatory attributes* specifies the minimum information that the handle needs in order to invoke the navigation calculus *expression* (the fourth component) and retrieve the requisite data. The set of *selection attributes* specifies the additional attributes that might be also specified. These additional attributes are used by the expression and are eventually passed to the various Web servers who, presumably, use these attributes to return more specific answers. For convenience, we assume that *mandatory-attrs* ⊆ *selection-attrs*.

There can be several handles for the same relation. Different handles for the same relation must use different sets of mandatory attributes. However, different handles can have the same sets of selection attributes and the same navigation expression (for instance, the same HTML form might have two alternative sets of attributes; at least one of them must be filled in order to get a result).

We assume that all *handles for the same relation agree with each other*: if $H_1 = \langle M_1, S_1, R, E_1\rangle$ and $H_2 = \langle M_2, S_2, R, E_2\rangle$ are two handles for the same relation and we specify concrete values for a set of attributes S such that $M_1 \cup M_2 \subseteq S \subseteq S_1 \cap S_2$, then handles H_1 and H_2 return the same result.

Table 3 shows the sets of mandatory and selection attributes for some relations in the VPS of Example 2.1. The first column in the table lists relation schemas, the second column shows mandatory attributes for each schema, and the third shows the optional attributes (= *selection-attrs* – *mandatory-attrs*).

4 Navigation Calculus

Navigation maps. The basic data structure that enables automated access to virtual relations residing in the VPS of a webbase are the *navigation maps* for the participating sites. Intuitively, a navigation map codifies all possible access paths that a site presents for populating a virtual relation. A navigation map is a labeled directed graph (see Figure 2) where the nodes represent the structure of static or dynamic Web pages, and the labeled edges represent possible actions (*i.e.*, following a link or filling out a form) that can be executed from a dynamic page. Our navigation maps are closely related to the Web schemes of the Araneus project [?, ?], but our modeling of the Web is process-oriented, which facilitates creation of the navigation expressions from navigation maps.

Mapping the virtual physical schema onto the raw Web requires a calculus of some sort. One obvious candidate would be the relational calculus or algebra, extended with web-specific primitives (and some other known extensions, like the unnesting operator of Ulixes [?]). The Araneus and the Ariadne projects [?, ?] take this approach. However, these formalisms are not powerful enough to express complex navigation processes on the web. For instance, as shown in Figure 2, a navigation process to access the used car ads in the classified section at the site `www.newsday.com` requires following a link (*link(auto)*), filling out a form (*form f1(make)*), then making an if-then-else choice depending on the resulting page —if the page is not a data page, another form (*form f2(model,featrs)*) will have to be filled out. The length of the sequence is not fixed. It is usually one or two, depending on the number of answers that match the initial query. Once the final data page is reached, an iteration to collect data is needed (repeatedly hitting the "More" button).

features found in other sites.

Description	*VPS Level Relations*
Used Car Ads	newsday(Car,Price,Contact,Url), newsdayCarFeatures(Url,Features,Picture) nyTimes(Car,Features,Price,Contact)
Dealer Cars Blue Book Prices	carPoint(Car,Price,Features,ZipCode,Contact) autoWeb(Car,Price,Features,ZipCode,Contact) kellys(Car,Condition,BBPrice)
Reliability	carAndDriver(Car,Safety)
Interest Rates	carFinance(Car,ZipCode,Duration,Rate)

Table 1: VPS Level Relations

Logical Level Relations	*Definitions*
classifieds(Car,Price,Contact,Features)	$\pi_{\texttt{Car,Price,Contact,Features}}(\texttt{newsday} \bowtie \texttt{newsdayCarFeatures})$ $\cup\ \pi_{\texttt{Car,Price,Contact,Features}}(\texttt{nyTimes})$
dealers(Car,Price,Contact,Features)	$\pi_{\texttt{Car,Price,Contact,Features}}(\texttt{carPoint})$ $\cup\ \pi_{\texttt{Car,Price,Contact,Features}}(\texttt{autoWeb})$
blue_price(Car,Condition,BBPrice)	$\pi_{\texttt{Car,Condition,BBPrice}}(\texttt{kellys})$
reliability(Car,Safety)	carAndDriver
interest(Car,ZipCode,Duration,Rate)	carFinance

Table 2: Logical Level Relations

Examples like this and our experience with other, more complex, sites shows that navigation processes are best represented using a calculus that allows recursion and has the notion of ordering of events. In addition, the calculus must deal with complex structures, such as web pages, forms, etc., which are best represented as objects.[3]

Unlike other projects that deal with navigation processes on the web, we do not invent yet another, new navigation algebra or calculus. The calculus that satisfies all the requirements stated above is actually well-known: it is a subset of serial-Horn Transaction F-logic [?], a natural cross between Transaction Logic [?] and F-logic [?]. In fact, the Florid system [?], based on F-logic, has proved to be very successful for Web applications. Because Florid lacks the Transaction Logic component, it is not suitable to be used as a calculus for encoding navigation processes.

The object model. F-logic extends classical logic by making it possible to represent complex objects on a par with traditional flat relations. A navigation map is a collection of F-logic objects, such as the following object that represents one of the forms to be filled out at the Newsday's site:

```
submit_form : action[form → form01[cgi →
                "/cgi − bin/nclassyNDD.x/";
                method → "post";
                mandatory→→{make,model};
                optional→→{year}];
        source → www.newsday.com]
```

In the first line, *submit_form:action* says that the object *submit_form* belongs to class *action*. It has attributes *form* and *source*. The attribute *form* has the value *form01*, which represents the form to be filled out. *form01* is itself a complex object with four attributes: *cgi* and *method* are *single-valued*, and *mandatory* and *optional* are multi-valued. The attribute *source* of the object *submit_form* represents the page to which the action belongs. In addition, the object has a method, *doit* (defined below), whose purpose is to execute the action.

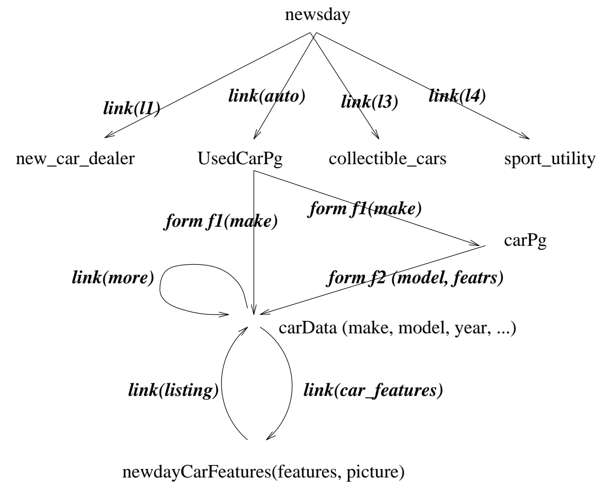

Figure 2: Navigation map for Newsday Classified Car Ads

Figure 3 presents the schemas (called signatures) of some of the objects we use to model navigation maps. The double-shafted arrows $\Rightarrow$ and $\Rightarrow\!\!\!\Rightarrow$ (as opposed to $\rightarrow$ and $\twoheadrightarrow$ in the previous example) signify that these expressions declare the *types* of the attributes and methods rather than their states.

[3] Observe that the user-level view of the database is represented using the relational model. However, the underlying navigation process (which is invisible to the end user) is based on the object model, since it has to deal with Web pages and other objects, which are not part of the user view.

VPS	*Mandatory*	*Optional*
`newsday(Make,Model,Price,Contact,Url)`	`Make`	`Model`
`newsdayCarFeatures(Url,Features,Picture)`	`Url`	
`nyTimes(Make,Model,Features,Price,Contact)`	`Make`	
`kellys(Make,Model,Condition,BBPrice)`	`Make,Model`	`Condition`

Table 3: Virtual Physical Schema

`currentUrl(pid,url)`	Current URL of browsing process PID
`action[`	Declaration of Class Action
`object⇒{link,form}`	Action can apply to a form or a link
`source⇒url;`	Page where the action belongs
`targets⇒⇒web_page;`	Where this could lead us
`doit@attrValPair⇒web_page]`	Method to execute action
`submit_form : action`	Form fillout is an action
`follow_link : action`	Following a link is an action
`web_page[`	Declaration of Class WebPage
`address⇒url;`	URL of page
`title⇒string;`	Title of the page
`contents⇒string;`	HTML contents of page
`actions⇒⇒{action}]`	List of actions found in the page
`data_page :: web_page`	The class of data Web pages is a *subclass* of `web_page`
`data_page[extract⇒relation]`	Data pages have a data extraction method
`link[`	Declaration of Class Link
`name⇒string;`	Name of link
`address⇒url]`	URL of link
`form[`	Declaration of Class Form
`cgi⇒url;`	CGI script's URL associated with this form
`method⇒meth;`	CGI invocation method
`mandatory⇒⇒attribute;`	Mandatory attributes of this form
`optional⇒⇒attribute;`	Optional attributes of this form
`state⇒⇒attrValPair]`	State of form (set of attribute-value pairs)
`attrValPair[`	Declaration of Class AttrValPair
`attrName→string;`	Name of the attribute part
`type→widget;`	Checkbox, select, radio, text etc.
`default→Object;`	Default value of the attribute
`value→Object]`	The value part

Figure 3: Common WWW Data Structures Represented in Navigation Calculus

Navigation expressions. F-logic provides a declarative calculus for representing complex objects on the Web, but to model navigation processes one needs a formalism for the representation and sequencing of actions. These facilities are provided by *Transaction Logic* [**?**], a conservative extension of classical logic, which is suitable for representing complex declarative processes that both query the underlying database state *and* update it. The following subset of serial-Horn Transaction Logic complements the subset of F-logic described above and provides an expressive navigation calculus for enabling access to raw Web data.

For the purpose of this paper, it suffices to explain the informal, procedural reading of some of the connectives of Transaction Logic Underlying the logic and its semantics is a set of database *states* and a collection of *paths*. A *path* is a finite sequence of database states. For instance, if s_1, $s_2, ..., s_n$ are database states, then $\langle s_1, s_2, ..., s_n \rangle$ is a path of length n. Just as in classical logic, Transaction Logic formulas assume truth values. However, unlike classical logic, the truth of these formulas is determined over paths, *not* at states. If a formula, ϕ, is true over a path $\langle s_1, ..., s_n \rangle$, it means that ϕ can *execute* starting at state s_1. During the execution, the current state will change to s_2, s_3, ..., etc., and the execution terminates at state s_n.

Procedurally, a Transaction Logic formula can be understood as a transaction or a query (depending on whether it changes the database state or not). Semantically (and procedurally) these formulas have several common attributes of database transactions, such as atomicity and isolation. With this in mind, the intended meaning of the new connectives of Transaction Logic can be summarized as follows:

- $\phi \otimes \psi$ *means*: execute ϕ then execute ψ.

```
newsday(Make,Model,Price,Contact,Url) ←                                      Find car ads at Newsday:
   newsdayPg.actions : follow_link[object → link(auto);                      Follow link(auto)
                                       doit@()→UsedCarPg]                    to used car ads;
   ⊗ UsedCarPg.actions : submit_form[object → form(f1);
                                       doit@(Make)→CarPg1]                   Fill form(f1) using Make;
   ⊗ (CarPg1 : data_page[extract→→tuple(Contact,Price,Url)]                  Either extract data,
       ∨ (CarPg1.actions : submit_form[object→form(f2);                      or fill form(f2)
                                       doit@(Make,Model)→CarPg2]             using Make & Model,
          ⊗ CarPg2 : data_page[extract→→tuple(Contact,Price,Url)])           then extract data
```

Figure 4: The Navigation Process of Retrieving Used Car Advertisements from Newsday Site

- $\phi \vee \psi$ *means*: execute ϕ *or* execute ψ non-deterministically. This connective is useful for specifying alternative execution branches in a navigation process.

Figure 4 shows the navigation process to extract car ads from the Newsday site. Here we use path expressions as shortcuts for longer F-logic expressions, as described in [?, ?]. For the benefit of the reader who is not fluent in Transaction F-logic, we annotated each clause in Figure 4, so the meaning of the navigation expression should be self-explanatory. Navigation expressions do not always need to be that complex. For instance, navigating to the *NewsdayCarFeatures* page, which is part of our VPS, can be achieved using the much simpler expression:

```
newsdayCarFeatures(Url,Features,Picture) ←
   (DataPg : data_page).actions[object →
   link(_)[name → "CarFeatures"];
      doit@() → _[address → Url;
      extract → tuple(Features,Picture)]]
```

This expression says that to get to a car features page, we must first get to a data page (*DataPg*) that has a link called "Car Features", follow this link, and then extract the features from the page. Of course, in a bigger system, we would have to qualify this initial page even further to avoid mis-navigation. The interesting point here is that the page denoted as *DataPg* is not an entry point to any navigation process that a regular Web user might perform. Indeed, as seen in Figure 4, one has fill out one or two forms to reach this page. However, this is not a concern at the VPS layer. It is a job of the logical layer, described in Section 5, to order joins in such a way that the relation *newsday* of Figure 4 is computed first (which would give us the desired page, *DataPg*).

Even if the above Transaction F-logic expressions look a bit complex to the reader, the most important aspect of our webbase architecture is that nobody, except the system builder (*i.e.*, us), needs to ever see these expressions. It is easy to see that the above expressions closely mimic the structure of the navigation map in Figure 2 and, in fact, they can be *derived automatically* directly from that map in linear time in the size of the map. (We do not present the translation algorithm, due to space limitation, because this would require that we spell out the structure of the navigation maps in much greater detail.)

It is important to realize that the translation from the map to the calculus expression has been greatly facilitated by:

- our process-oriented object model, whose objects correspond to nodes and links of the navigation map;
- the fact that the F-logic component of our navigation calculus naturally supports this object model; and
- the Transaction Logic component that represents the process structure encoded in the navigation map.

Finally, once the translation is done, the resulting navigation expressions can be directly executed by a Transaction F-logic interpreter when user queries posed against the external schema level of the webbase eventually turn into queries against the VPS layer.

5 Logical Layer

The VPS layer provides a relational view of data that can be retrieved from a Web site, thereby hiding navigation details. In contrast to this, we use a *logical layer* to provide a uniform interface to data arriving from multiple sources. By separating these layers, we achieve *site independence*. This means both independence from the differences in vocabulary and representation used by different sites *as well as* complete transparency with respect to where the data is coming from.

Table 2 shows a possible mapping of Logical relations into VPS relations. While VPS layer has eight relations that shield the user from navigation details, the five logical relations in the example show a view of the Web data that is completely transparent with respect to the location of the data source.

In this paper, we are not concerned with the issues pertaining the mapping of the logical layer onto VPS. This mapping can be done using conventional techniques (*e.g.*, relational algebra, or Datalog rules) or we could use more advanced techniques, which might offer certain advantages in the Web environment (*e.g.*, [?, ?]).

However, one issue related to this mapping must be addressed. The problem is that unlike traditional databases, VPS relations can only be accessed by supplying values for certain sets of mandatory attributes. Since logical relations are mapped onto the physical ones, it is clear that they also can be accessed only by providing values for certain attributes. The process of determining these sets of attributes is called *binding propagation* (because, in abstract terms, sets of mandatory attributes in HTML forms correspond to variable bindings in programming languages).

The problem of binding propagation has been well-studied in the literature (see *e.g.*, [?, ?]). In the following, we propose a much simpler description, which also differs from other works in two respects: (1) it handles not only conjunctive queries, but also *all* relational algebraic queries; and (2) instead of deriving bindings for a given query on the fly, it statically determines all allowed bindings for each logical relation.

Let α denote a relational algebraic expression over VPS relations, and we need to determine the bindings (or sets of mandatory attributes) for the resulting relation of this expression. The binding propagation algorithm can be described by the following rules, each corresponding to one of the allowed relational operators:

- Let $\alpha = V$, where V is a VPS relation, if M is a binding for V, then M is also a binding for α.
- Let $\alpha = E_1 \cup E_2$ or $\alpha = E_1 - E_2$, where E_1 and E_2 are relational expressions over VPS, if M_1 is a binding for E_1 and M_2 is a binding for E_2, then $M_1 \cup M_2$ is a binding for α.[4]
- Let $\alpha = \pi_X(E)$ or $\alpha = \sigma_\phi(E)$, if M is a binding for E then M is also a binding for α.
- Let $\alpha = E_1 \bowtie E_2$, if $M_1,\ M_2$ are bindings for $E_1,\ E_2$, respectively, then $M_1 \cup (M_2 - (E_1 \cap E_2))$ and $M_2 \cup (M_1 - (E_1 \cap E_2))$ are both bindings for α. Here $E_1 \cap E_2$ denotes the set of common attributes of the relation schemas for E_1 and E_2.

From these rules, it is also easy to derive an algorithm for join ordering under the given set of bindings, *i.e.*, an ordering $R_1, ..., R_n$ that guarantees that for each i ($1 \leq i \leq n$), all mandatory attributes of R_i belong to the union $\cup_{j=1}^{i-1} R_j$. Clearly, the existence of such an ordering is necessary and sufficient for a join to be computable under the given set of mandatory attributes. However, in the presence of multiple sets of mandatory attributes per VPS relation, such an algorithm would be exponential. In fact, [?] shows that this problem is NP complete in this case.

To illustrate the above binding propagation algorithm, consider the logical level relation *classifieds* from Example 2.1. Since $Make$ is the only mandatory attribute of the relation *newsday* and Url is the only mandatory attribute of $newsdayCarFeatures$, by the join rule above, $\{Make\}$ turns out also to be the only mandatory binding for $newsday \bowtie newsdayCarFeatures$. Similarly, $Make$ is the only mandatory attribute of *nyTimes*. Therefore, by the union and projection rules, $\{Make\}$ is the only mandatory binding for classifieds.

6 External Schema

Casual users query the webbase through the external schema. Traditionally, end users have been given access to limited interfaces that allow only a fixed set of *canned* queries. These canned interfaces served well in the case of fairly structured business environments, but, as remarked earlier, they are too limiting for a casual web user. On the other hand, more flexible query languages, such as SQL or QBE, are too complex. In search of a suitable query interface for webbases, we resurrected the idea of the *Universal Relation* (UR) [?].

The basic idea is simple and appealing. The user is presented with a list of all attributes that might be of interest for a particular application domain. To pose a query, the user simply points to a set of output attributes and imposes conditions on some other attributes. This is it: no joins, sheer simplicity. Of course, to realize such an agenda, the system (and the user) must know what such a query exactly means, and the understanding of that meaning by both the system and the user must coincide.

Simplistically, the semantics of a universal relation query is explained as a natural join of the underlying relations at the logical layer, which cover the output and the selection attributes specified in the query. Moreover, the join must be *lossless*. Losslessness is required because this is a formal analog of the common sense idea of connections between concepts that "make sense."

Underlying this idea are two basic assumptions:

1. *The unique relationship assumption:* The relationship between any given subset of attributes in the universal relation schema is unambiguous and unique; and
2. *The unique role assumption:* The name of an attribute unambiguously determines the role of that attribute.

The first problem arises even in very simple schemas that contain just four attributes. For instance, a customer and a bank might be connected because the customer has an account in the bank, a loan, or both. Which one did the user have in mind when she selected *Bank* and *Customer* as output attributes? A number of solutions were proposed to address the first problem, which range from restricting the topology of the underlying logical schema (*e.g.*, acyclicity [?]) to additional layers of semantics (*e.g.*, Maximal Objects, Window Functions [?, ?]).

Unfortunately, on the Web, we cannot assume the very basic lossless join semantics for UR, since we cannot even assume any dependencies (join, functional, or multivalued) on which the very idea of losslessness is based. Nor can we use most of the approaches to enforcing or relaxing the unique relationship assumption, because these approaches rely heavily on the use of constraints.

The second problem, the unique role assumption, was assumed to be solvable by simple renaming of attributes. However, this solution was never thought to be practical and may have been responsible for the general lack of enthusiasm for the UR approach.

In our attempt to adapt the UR as a web interface, we kept the basic idea of a simple query interface, but rejected the lossless join semantics and the two uniqueness assumptions. We call this approach *structured universal relation*. The

[4] Here we assume that the user wants *all* available answers to the query. If the user is willing to accept only *some* available answers because she does not want or care to fill out all the required attributes in a form, then we could define a *relaxed union*. In a relaxed union, both M_1 and M_2 (separately) would be acceptable bindings for α.

basic idea is to replace losslessness and constraints with *compatibility rules*. A compatibility rule has either the form $R_1, ..., R_k \rightarrow R$ or the form $R_1, ..., R_k \rightarrow \neg R$. In the first case, the rule says that if you already joined $R_1, ..., R_k$ then joining with R also "makes sense." This is our "poor man's lossless join requirement." The second rule is really a constraint. It says that if we have already joined $R_1, ..., R_k$, then joining with R would create an incorrect relationship (in the UR model, such connections are known as "navigation traps").

With these constraints, we can formulate the semantics of a query as follows: Let Q be a query that mentions the set of attributes $A = A_1, ..., A_m$. Then the semantics of this query is said to be the join $R = R_1 \bowtie ... \bowtie R_n$, where $R_1, ..., R_n$ is a minimal (with respect to inclusion) subset of logical relations that satisfy the compatibility rules, and R contains all attributes in A.[5] This is essentially our analogue of the Maximal Objects approach [**?**]. If there are several maximal objects covering the query attributes then we take the union of results obtained from each of the object. Depending on the exact structure of the compatibility rules, algorithms with various efficiency can be constructed. For instance, if the rules are of the form $R \rightarrow Q$, then we have a restricted join-ordering problem mentioned in Section 5.

To address the problem of unique name assumption, we propose to organize the attributes in the UR into a hierarchy of concepts. Each concept is a relation schema whose attributes are concepts of a lower layer. As shown in Figure 5, the top layer in this hierarchy is the universal relation itself, and the concepts are the attributes of that relation.

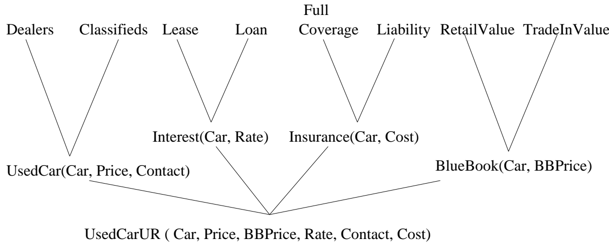

Figure 5: Concept Hierarchy for the Used Cars UR

Example 6.1 (Concept Hierarchy) The concept hierarchy describes the following: (1) A used car is either advertised at a dealer site or it is in the classified section of a newspaper site; (2) The blue book price of a car can either be its trade-in price or its selling price; (3) The interest rate for a used car depends on whether it will be financed or leased; and (4) the insurance rate depends on whether it provides full or liability coverage. □

The idea behind concept hierarchies is that the user starts by selecting top-level concepts and then proceeds to subconcepts. This makes it possible to build queries incrementally, by restricting the search to various sub-concepts and to specific ranges for attributes at the leaf level. The unique name assumption is not an issue here — for the user or the system —since both can see the entire concept hierarchy to which the attributes belong, and the relationships among concepts and attributes are definied by the compatibility rules.

We believe that webbases will be designed for application domains (such as cars, jobs, houses) by the experts in those domains, and designing concept hierarchies and compatibility constraints is a feasible task for them. We illustrate these ideas with an example, leaving out the details due to space limitation.

Example 6.2 (Structured UR in Action) The following compatibility constraints specify the meaningful connections for the UsedCarUR of Example 6.1.

Compatibility Constraints	*Semantics*
$\texttt{Classifieds} \rightarrow \neg\ \texttt{Lease}$	We cannot lease a car from its owner
$\texttt{Lease} \rightarrow \texttt{Full_Coverage}$	Leased cars have to be fully insured
$\texttt{Used_Cars} \rightarrow \neg\ \texttt{TradeIn_Value}$	Trade-in values are not applicable

Consider the following query: *make a list of used Jaguars advertised in New York City area sites such that each car's monthly payments are less than 1,000 dollars, and its selling price is less than its Blue Book price*. This query can be expressed as:

$$\texttt{Used_Car_UR(jaguar,Model,Year,Price,BBPrice,} \\ \texttt{Rate,Ins_Cost),Price} < \texttt{BBPrice,} \\ \texttt{(Price} \times \texttt{Rate} + \texttt{Ins_Cost)} \div \texttt{12} < \texttt{\$1000}$$

Using compatibility constraints, our algorithm generates the following maximal objects and the corresponding relational expressions:

$$\texttt{Dealers} \bowtie \texttt{Lease} \bowtie \texttt{Full} \bowtie \texttt{Retail_Val} \\ \cup\ \texttt{Dealers} \bowtie \texttt{Loan} \bowtie \texttt{Full} \bowtie \texttt{Retail_Val} \\ \cup\ \texttt{Dealers} \bowtie \texttt{Loan} \bowtie \texttt{Liability} \bowtie \texttt{Retail_Val} \\ \cup\ \texttt{Classifieds} \bowtie \texttt{Loan} \bowtie \texttt{Liability} \bowtie \texttt{Retail_Val} \\ \cup\ \texttt{Classifieds} \bowtie \texttt{Loan} \bowtie \texttt{Full} \bowtie \texttt{Retail_Val} \quad \square$$

Now assuming the existence of a mapping function from external schema relations to the logical level, maximal objects made up from the UR relations can be translated into conjunctive queries over logical level relations. Once translated, these queries can be optimized and evaluated by standard query evaluation techniques.

7 Implementation and Experiences

We have implemented the most essential components of two of the modules in our webbase architecture: the *navigation mapper* and the *query evaluator*. In what follows we describe the ideas underlying our implementation.

Navigation Mapper. We use the methodology of *mapping by example* to extract the *navigation maps* from Web sites. The main idea behind mapping by example is to discover the

[5] Compatible means that for every $1 < i \leq n$, there is a rule $Left \rightarrow R_i$ such that $Left \subseteq \{R_1, ..., R_{i-1}\}$; and there is no rule $Left \rightarrow \neg R$ such that $Left \cup \{R\} \subseteq \{R_1, ..., R_n\}$.

structure (or schema) of a site while the webbase designer moves from page to page, filling forms and following links. There are two key components to this methodology: (1) discovery of access paths to the data of interest; and (2) extraction of action objects (see Figure 2).

In order to build a practical tool, there are two important requirements: the mapping process should be as *transparent* as possible to the webbase designer (its operation should closely mimic the browsing experience); and the mapping tool must be portable (*e.g.*, it should not require modifications to the browser).

The navigation mapper achieves these goals by using JavaScript events to capture browsing actions. Actions are dynamically intercepted by JavaScript handlers (inserted into the retrieved pages by the mapper), and are added as edges of the navigation map. When a new page is loaded into the browser, it is parsed, and a new node corresponding to the page is inserted into the navigation map.[6] In order to guide the designer, an applet displays a graphical representation of the navigation map as it is being constructed, highlighting in the map the node corresponding to the page displayed in the browser.

The map builder parses an HTML page and generates a set of F-logic objects (as detailed in Section 4). It extends PiLLoW[7], a publicly available Prolog-based system, to extract all necessary information for following links and submitting forms found inside the page. Since not all information is stored in the HTML object structure (*e.g.*, labels denoting the domain values of some attributes and attributes defined through a set of links) we take advantage of HTML tags and anchors and other structuring primitives (*e.g.*, tables, enumeration) to extract such information. For forms, the extractor is also able to infer which attributes are mandatory from their widget (*i.e.*, if an attribute is represented by a radio button we can safely assume it is mandatory), as well as other information such as the domain of attributes (*e.g.*, from the values of a selection list), maximum length (*e.g.*, for a text field), default value, to name a few. For data pages, as described in Figure 3, we assume that the designer provides an extraction script.

Of course there are instances where input from the designer is needed. For instance, the designer has to indicate whether a text field is mandatory. Also, it is not uncommon in forms for attributes to have rather cryptic symbolic names —in these cases (to facilitate subsequent querying) the user might want to provide a more informative name. There are also instances where attributes are implicitly defined through a set of links (*e.g.*, a list of links with car models). Since this kind of attributes is not part of a form, the designer has to specify a name as well as the set of links that relate to this attribute. It is worth pointing out that in many instances our parser is able to find these links by considering their HTML environment (*e.g.*, a table), or the user can provide additional hints. We are currently building a graphical user interface to simplify the input of such information by the designer.

We used our initial prototype of the map builder to map various sites. To give an idea of the degree of automation achieved, for the Newsday site depicted in Figure 2, all objects that describe the navigation map (85 objects with over 600 attributes in total) were automatically extracted. Less than 5% of the information in the map was added manually, which consisted of 10 to 12 facts to standardize attribute and domain value names. For other sites such as New York Times and Daily News, the ratio was similar. The process of mapping each of these sites took on average 30 minutes. It is worth pointing out that the main problem we face while mapping sites is the presence of faulty HTML, in which case the parser needs to be able to recover from the ill-formed documents.

Some points are worthy of note with respect to the maintenance of such maps. Modifications to Web sites can be automatically detected by periodically comparing the navigation map against its corresponding site, or when the corresponding navigation process fails. Whereas certain structural changes such as the addition of a new form attribute require manual intervention, others can be applied automatically (*e.g.*, the addition of a cell in a selection list). Since we first built navigation maps for car-related sites, we have noticed quite a few changes to these sites. For example, in Kelly's Blue Book (www.kbb.com) new links with information about 1999 cars have been added. In order to update navigation map, we only had to navigate through the modified pages, a process that took a few minutes.

Query Evaluator. As described in Section 4, once a map is built, navigation expressions are automatically generated. This process requires a simple traversal of the navigation map, and thus can be done in linear time in the size of the map.[8] Individually, each expression can be seen as a shortcut to retrieve data from a web site. Instead of filling forms and following links, one can simply specify a set of attributes and execute the appropriate navigation expression (*e.g.*, for the query *SELECT make,model,year,price,contact WHERE make=ford AND model=escort*), execute *newsday(ford,escort,Year,Price,Contact)* (described in Figure 4). It is worth pointing out that as a byproduct, the process of retrieving such data is made faster since during the execution of a navigation process no extraneous objects such as figures and Java animations are retrieved.

Navigation expressions are processed by the Transaction F-logic interpreter, which translates them into logic programs that are executed by a deductive engine, the XSB system.[9] On top of XSB, we use the HTTP library provided by PiLLoW to follow links, submit forms and retrieve documents from the Web.

[6] Since building maps is an incremental process, our tool checks whether actions and Web page objects are new before adding them to a map.

[7] http://www.clip.dia.fi .upm.es/Software/pillow/pillow.html

[8] The algorithm to generate navigation expressions is described in the full-version of this paper available at http://www-db.research.bell-labs.com/users/juliana.

[9] http://www.sunysb.edu/˜sbprolog

In order to combine information from different sites (or maps), the attribute names and their domains must be standardized. In our current implementation, one must manually specify these mappings. If a mapping is not provided for a certain attribute name, we employ fuzzy matching techniques, which evidently are not full-proof and may lead to errors. We intend to incorporate techniques from mediator systems such as [?, ?] to address this problem.

We have manually built an integration wrapper for 10 car-related sites, and below we show the number of pages navigated and (some of the best) evaluation times for the query *SELECT make,model,year,price WHERE make=ford AND model= escort* for each site. These times were collected on a Sun Ultra workstation, with dual 330 MHz processors, 1 GB of memory, and Solaris 5.6 operating system. These timings indicate that to ensure acceptable response times when querying a large number of sites, we may need to use techniques such as parallelization and caching.

Site	*# of pages*	*cpu time*	*elapsed time*
AutoWeb	2	2.84	8.00
WWWheels	2	1.29	5.82
NYTimes	3	0.92	5.08
CarReviews	3	3.59	10.10
NewYorkDaily	3	3.43	11.70
Car&Driver	3	3.54	12.14
AutoConnect	3	7.26	14.70
Newsday	4	1.01	4.77
YahooCars	5	3.12	10.48
Kelly's	5	2.54	7.63

It is worth pointing out that a significant portion of the time in querying is spent in fetching and parsing the Web pages. We believe these times can be greatly improved if a faster parser is used.

8 Related Work

The problem of retrieving data from and querying Web sources has received considerable attention in the database literature (see [?] for a survey). Managing information on the Web encompasses several tasks that include locating interesting data, modeling Web sites, extracting and integrating related information from multiple sites.

Web query languages such as W3QL [?], WebSQL [?], WebLog [?], and Florid [?] address the problem of finding and retrieving data from the Web. They improve on search engines by combining textual retrieval with structure and topology-based queries. These languages view the Web as a collection of unstructured documents organized as a graph, and users can declaratively express how to navigate portions of the Web to find documents with certain features. Conceptually, these languages are equivalent to various subsets of our navigation calculus. More importantly, however, these are fairly sophisticated query interfaces designed to be used by a fairly sophisticated user. In contrast, even though our navigation calculus requires an even greater degree of programming expertise, it is not designed to be used by a programmer. Instead, navigation expressions are generated automatically from the map.

Web information integration systems [?, ?, ?, ?] are more closely related to our work in that they try to present the Web through a unified database interface. The Araneus project [?] provides a rich model (ADM) to describe both the topology and the contents of Web sites. Their concept of the *ADM scheme* is analogous in many respects to our navigation maps. Navigation processes to populate database views are expressed in a newly developed declarative algebra, called *Ulixes*. Ulixes is intended to be used by a database designer to create Web views for the end user. In contrast to this, we use a well-known, existing formalism (Transaction F-logic [?]), which functionally is a superset of Ulixes. However, as mentioned earlier, it is not intended to be used by a designer or an end user. Instead, navigation expressions that use this language are generated *automatically* from the navigation map. The interpreter than simply executes these expressions when user queries need to be evaluated. This is possible in our architecture due to the clear separation between the VPS and the logical layers of the database and also due to the use of our process-oriented object model. It is worth pointing out that maintenance of navigation expressions in our approach is much simpler, since the navigation maps from which the processes are generated, can be updated semi-automatically (through mapping by example).

Ariadne [?] is a system for extracting and integrating data from semi-structured Web sources. Ariadne has two foci: data extraction from unstructured Web pages and what in our architecture amounts to mapping from the logical layer to the virtual physical layer. Both of these issues are orthogonal to our work. For instance, Ariadne's data extraction facilities as well as the body of techniques for extracting information from semi-structured data [?] could be used in our system.

From the perspective of our architecture, the focus of the work in the Information Manifold (IM) [?, ?] project can be viewed as mapping the logical layer to VPS. IM approaches the problem by first specifying the *reverse* (physical-to-logical) mapping, which they call *source description*. The required logical-to-physical mapping is then generated automatically. The benefit of this indirect approach is claimed to be the ease of maintenance of the logical-to-physical mapping in view of adding or deleting the Web sources. In this way, IM is complementary to our work, since our focus is on building the VPS and the conceptual layers of the webbase.

There is a large body of work on information mediators, such as TSIMMIS [?], Hermes [?] and Garlic [?], which help smooth the semantic and syntactic differences between heterogeneous information sources. Techniques developed for information integration systems such as these can be used in our architecture for semantic integration of VPS relations that come from different sources. On the other hand, these systems could use our VPS automation techniques to gain

access to dynamic Web content.

Finally, we should note the growing commercial interest in integration of information from diverse Web sources (*e.g.*, Junglee, Center Stage [**?**, **?**]). Techniques described in this paper can facilitate rapid development of such services.

9 Conclusions and Future Directions

In this paper we described a layered architecture for designing webbases. The separation of layers, which is analogous to traditional databases, simplifies the creation, maintenance, and use of webbases for retrieving information available on the web. We have implemented the main components of a prototype implementation of our architecture and reported on some preliminary experimental results.

We have shown that navigation maps can be created semi-automatically as the webbase designer browses sites, and that navigation expressions can be automatically derived from these maps. These expressions are executed when evaluating a query, and thus optimizing such expressions is an important problem that needs to be studied.

Our experiments suggest that parallelization of query evaluation is crucial for obtaining acceptable response times. Finally, while the idea of structured UR as a query interface seems to be promising in the context of webbases, more experimental work needs to be done to evaluate the practicality of the idea.

Acknowledgements: We would like to thank various people that contributed to this work: Vinod Anupam for suggestions on how to implement navigation by example; Daniel Lieuwen and C.R. Ramakrishnan, for carefully reading this manuscript; and Narain Gehani, David Warren and Guizhen Yang for valuable discussions.