Personalizing the Web Using Site Descriptions

Vinod Anupam Yuri Breitbart Juliana Freire Bharat Kumar
{anupam,yuri,juliana,bharat}@research.bell-labs.com
Bell Laboratories, Lucent Technologies
Murray Hill, NJ 07974

Abstract

*The information overload on the Web has created a great need for efficient filtering mechanisms. Many sites (e.g., CNN and Quicken) address this problem by allowing a user to create personalized pages that contain only information that is of interest to the user. We propose a new approach for personalization that improves on existing services in three significant ways: the user can create personalized pages with information from any site (without being restricted to sites that offer personalization); personalized pages may contain information from multiple Web sites (*e.g.*, a user can create a personalized page that contains not only news categories from her favorite news sources, but also information about the prices of all stocks whose names appear in the headlines of selected news, and weather information for a particular city); and users have more privacy since they are not required to sign up for the service. In order to build a personalization service that is general and easy to maintain, we make use of site descriptions that facilitate access to the data stored in and generated by Web sites. Site descriptions encode information about the contents, structure, and services offered by a Web site, and they can be created semi-automatically.*

1. Introduction

The information overload that we face today on the Web has created a great need for efficient filtering mechanisms that give better and faster access to data. Some sites (*e.g.*, CNN and Quicken) address this problem by allowing a user to create personalized pages that contain only information that is of interest to the user. Other sites send notifications when the underlying data changes and some condition is met. However, there are problems with these approaches to personalization: 1) personalization is limited to the information accessible from a single site; 2) personalization features are not offered by *all* the Web sites that might be of interest to a user; 3) since sites that provide personalization require users to sign up, there are privacy issues involved — the user needs to divulge personal information when signing up, and the sites can track the interests of the user.

In this paper we propose a new approach to personalization that allows a user to create personalized pages that contain information from *any* Web site. The user has full control over which information is retrieved, and thus she need not sign-up for any special service. In addition, personalized pages may include information from multiple sites (*e.g.*, one's favorite news categories from her preferred news source, weather information for her city, and the traffic report for her afternoon commute). This service can also be extended to customize change notifications based on conditions that may span multiple sites.

Our personalization system, *MyOwnWeb*, views a personalized page as a set of logical queries against Web sites, and the user can specify the conditions under which each query needs to be re-executed and the page refreshed. Each logical query is translated into a query to a specific Web site by translating logical attributes of the personalized system into the actual attributes of a specific Web site. Such an approach enables user to get information from several Web sites without requiring the user to know specific addressable attributes that the site may have.

Since Web languages such as W3QL [11], WebOQL [14], and WebL [9] provide mechanisms for querying the Web, a limited version of this service could be built using these languages — either allowing users to write a set of queries, or by providing a set of pre-defined queries for a user to choose from. Neither alternative is satisfactory: whereas the latter is too restrictive (and since Web sites change constantly, significant effort may be required to maintain such queries), the former is likely not to be widely acceptable as these languages can be too complex for an average Web user.

MyOwnWeb uses *site descriptions* as a simple and succinct way to represent the structure and contents of a Web site. These site descriptions are an extension of the navigation maps described in [5], and thus they can be generated semi-automatically. In addition users (or map builders) are not required to know specifics of the language that is used for site description specification. Given a set of such descriptions, a naive Web user can create queries by simply selecting from site descriptions the contents of interest without having to specify how they should be retrieved. A *point and click* user-interface is provided that lets users select attributes of interest from source pages, and prompts the user for required input values. In the absence of the description for a site, our service provides *SmartBookmarks* [2], a mechanism that transparently records a sequence of browsing actions that can be saved and replayed later.

The rest of the paper is organized as follows. In Section 2 we describe how site descriptions are used to model Web sites, and how they can be used to build personalized pages. An overview of the *MyOwnWeb* personalization system is given in Section 3. Related work and concluding remarks are presented in Sections 4 and 5.

2. Modeling Web Sources

Several data models to represent the Web have been proposed in the literature (*e.g.*, [4, 10, 5]). Site descriptions based on these data models provide a uniform representation of sites that is similar to a schema in traditional databases, and thus simplify querying of described sources. They are also focused, and need only reflect a set of pages that contain *relevant* information and services. Since these models contain not only information about the contents of pages, but also information required to navigate through a set of pages, they can be used for handling data retrieval and extraction in our personalization service.

The site descriptions used in our personalization service are an extension of the navigation maps described in [5]. A navigation map is a collection of Web objects that represent the portion of a Web site that is of interest to the map builder. It can be represented by a directed graph whose nodes correspond to *Web pages*, and an edge between nodes corresponds to the *action* that took the user from one page to another (see Figure 3). There are many advantages of using this model in a personalization service: (1) the user can be presented with a visual representation of the site and its contents, and once the desired information is selected, the actual navigation process to retrieve the selected pages can be generated automatically; (2) since these maps can be created semi-automatically (by example), one can potentially create descriptions for *many* Web sites, and when the underlying site changes, the descriptions can be easily updated.

As depicted in Figure 1 (modified from [5]), site descriptions are composed of common Web objects. For example, the `web_page` object has a url, contents, and a set of actions; the `action` object corresponds either to a form or to a link. In general, there is no restriction on what can constitute an action. A set of actions can be further extended with legitimate actions that each source page may offer. An

`web_page[`	Declaration of Class WebPage
`address⇒url;`	URL of page
`contents⇒string;`	HTML contents of page
`actions⇒⇒{action}`	List of actions found in the page
extracted_attr⇒⇒{*output_attr*}];	Attributes extractable from page
`action[`	Declaration of Class Action
`object⇒{link, form}`	Object that action applies to
`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
`link[`	Declaration of Class Link
`name⇒string;`	Name of link
`address⇒url]`	URL of link
`form[`	Declaration of Class Form
`cgi⇒url;`	URL for submission of this form
`method⇒meth;`	Submit Method of this form
`mandatory⇒⇒input_attr;`	Mandatory attributes of this form
`optional⇒⇒input_attr;`	Optional attributes of this form
`state⇒⇒attrValPair]`	State of form (name-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 1. Data Model for Navigation Maps

action has a source (the Web page where it appears), a set of possible targets (the Web pages it can lead to), and a method to execute the action (for example, follow link or fill form). Figure 4 shows an instance of a navigation map for `www.quicken.com`.

Even though navigation maps provide a good framework for retrieving Web pages, issues related to data extraction are not discussed in [5]. In what follows, we detail our extensions to handle data extraction.

Data Extraction In order to provide filtering in our personalization service, retrieved Web pages are further processed and attributes of interest extracted. For example, if one is interested in the prices of a set of stocks, he/she should be able to select from the page returned by `quicken.com` just the table with the prices (and omit all ads and other information that is returned together with the table). Sometimes though, fine-grained data extraction may be required. For example, the user may want to see only the `Bid` and `Ask` prices for a stock, and omit all the other information. Such an extraction can be done by selecting attributes of interest from the source pages using a *point-and-click* interface.

As is the case for data retrieval, data extraction should be easy to specify (so that it is feasible to cover a significant number of pages) and it should be robust, that is, it should not break (easily) when changes occur to Web pages. There is a vast literature and a number of systems have been proposed for performing data extraction on structured and unstructured text (see *e.g.*, [1, 3, 17]). Here, we restrict our discussion to data extraction from Web pages that contain *some* structure. We also assume that Web pages are either compliant with the HTML 4.0 specification, or are wellformed XML.*

Various formalisms have been proposed to specify data extraction functions. Some (*e.g.*, [17]) are based on extensions to the the Document Object Model (DOM) [6], which is an API for accessing elements in a page. The use of DOM API (or its extensions) has severe drawbacks. Since elements are accessed by their absolute position in the page structure, the specification is not robust, and minor changes to the page structure can cause such specification to break, or reference elements different from the intended ones.

Instead of inventing a new formalism for data extraction, we use the XPointer language [12]. Our motivation for choosing this language is the following. XPointer is a path expression language designed for referencing objects within Web pages. Since it has been designed for HTML and XML pages, it allows more sophisticated queries to be asked than permitted, for example, by WebL [9]. Moreover, it is also possible to create XPointers that can be tested for whether they identify the same object when followed, as they did when they were created. Thus it is easy to check when an XPointer breaks, as a result of changes in the page structure. Finally, XPointer is currently being reviewed as possible standard for referencing objects in Web pages. Acceptance of this language implies that we can ride the technology curve (with regards to parsers, tools, etc.) for free. XPointer expressions can be semi-automatically generated via a point-and-click interface in a spirit similar to that in W4F [17].

Extensions to Navigation Maps We extend the navigation maps of [5] with two new objects: input attributes (`input_attr`) and output attributes (`output_attr`) — see Figures 1 and 2. In our site descriptions, each form object has a set of input attributes, their corresponding types and information about whether the attribute is multi-valued.

Each Web page object has a set of output attributes that can be extracted from that page†. Associated with each attribute, is an XPointer specification of how the attribute is extracted from the page. Note that it is important to consider complex attributes. For example, consider a Web site that offers a form interface to retrieve a list of advertised used cars. Each item in this resulting list contains the price, year, and dealer information for a car. In order to extract this information, it is necessary to know the exact structure of the list of tuples returned.

Before presenting the model for attributes, we first discuss the different forms of attributes that we consider. We assume that input attributes can only be scalar (atomic), whereas output attributes can have complex types. Moreover, the type of attributes is always known (e.g., whether an input attribute is `string`, `char`, `integer`, `float`

*It is worth pointing out that even though many existing pages are ill-formed, tools are available (*e.g.*, Tidy [16]) to *fix* such pages.

†Note that some Web pages have no extractable attributes.

```
input_attr[
    type⇒{char, string, integer, float, date}
    multi_valued⇒boolean
]
simple_attr :: input_attr[
    attr_form_name⇒string
    binding⇒⇒[range_value, bind_value]
]
value_attr :: input_attr[
    binding⇒⇒[attr_form_name, range_value, bind_value]
]
output_attr[
    location⇒xpointer_ref
]
atomic_attr :: output_attr[
    type⇒{char, string, integer, float, date}
]
record_attr :: output_attr[
    fields⇒⇒output_attr
]
list_attr :: output_attr[
    item⇒output_attr
]
```

Figure 2. Attribute Specification in Site Descriptions

etc.). Input attributes can either be single-valued or multi-valued. For example, an auto classifieds Web site may allow multiple car models to be specified in a query. Multiple values specified for an attribute are assumed to return the union of the results obtained by specifying each of those values separately.

The complete attribute specification is presented in Figure 2. The input attributes are defined by their type (we only allow atomic input attributes), and a boolean flag specifying whether they are single or multi-valued. The specification also maintains a binding between their `range_value` (the value visible to the user while browsing), and their `bind_value` (the internal value actually submitted to the Web site). The form name used by the attribute is also stored (`simple_attr`). In certain cases (*e.g.*, `www.newsday.com`), some attributes have a different form name for each distinct attribute value (`value_attr`).

Output attributes are described by their type, and an XPointer expression. Each expression has a *locator source* which denotes the starting point of the tree traversal. For complex attributes (list or structures), the starting position of the entire attribute itself is identified (*e.g.*, the starting position of a record) and used as the implied locator source for the sub-attributes. For list attributes, the `span` construct [12] allows specification of the starting point of each item in the list.

Optimizing Site Descriptions Web sites have been designed for human use, and a lot of emphasis is placed on visual presentation and ease of use rather than rapid access to the underlying information. As a result, a user might need to traverse multiple links and fill out various forms in order to get to the information he/she needs, even though there might be a single cgi-script managing most of the interaction. For example, we examined ten Web sites that provide information about cars. All the sites require multiple interactions (to specify the filtering criteria such as the make and model of the car, price range etc.) to get to the desired information. However, in *nine* out of those ten sites, there is a single cgi-script that could be called by specifying all the filtering criteria as parameters to the script, resulting in the same information being retrieved. Since the actual information (*e.g.*, classified ads) is usually stored in a database, it is quite natural to have a single cgi-script that accesses the database using various filtering criteria.

Recall that site descriptions are created interactively, thus the navigation processes derived from them mimic user behavior. Even if there is a single underlying cgi-script, the derived process requires multiple links to be followed and multiple forms to be filled out. In most cases (as illustrated above), the maps can be *flattened* so that a single request-response interaction is required. Not only does this result in better performance, it also increases robustness since changes to the intermediate links/forms do not affect the data extraction process.

Another reason for flattening the navigation maps is that following multiple links/forms is sometimes difficult in the presence of client-side maps or dynamic content (Javascript, ActiveX controls, plugins, etc.), as that requires the personalization client to have a very tight integration with the Web browser. Moreover, generating the navigation map becomes more complex since additional action types (rather than simply following links or filling forms) are required. In the extreme case, it might be necessary to capture the exact mouse clicks made by the user. Since a flattened map makes use of only the standard HTTP protocol to interact with a Web site, it enables the creation of lightweight Web clients. Note that in some cases, multiple interactions might still be required. For example, the first access might generate a userid that is used in the remainder of the interaction. However, this does not prevent the rest of the navigation map to be flattened out.

Flattening navigation maps is made difficult by the fact that the Web server could be maintaining (and updating) some server state during the entire interaction. However, for sites where server state is not maintained, or it is not essential for the interaction, the corresponding navigation map can be flattened if all final GET/POST requests refer to the same cgi-script. A reasonable heuristic to flatten navigation maps is as follows.

- For each set of input attributes, choose values for these attributes that fall into their range (if a set of possible values for an attribute is known, then one of those values should be picked). The navigation map is then traversed using those values.
- If the final GET/POST method contains the attribute assignment $(fname = bname)$, where $fname$ is the attribute form name, and $bname$ is the binding value for the chosen range value, then the navigation map can be collapsed.

3. Creating Personalized Pages

In what follows we describe in detail *MyOwnWeb*, a site independent personalization service that allows users to combine information from multiple Web sites, as well as customize change notifications.

The basic idea The basic idea of *MyOwnWeb* is very simple: a personalized page is viewed as a set of Web queries. Queries can be as simple as a URL, which returns the Web page corresponding to the URL, or they can be complex as "find the prices of all stocks from the Quicken Web site that are also listed on CNN headlines", which requires information from multiple sites to be extracted and combined.

The system A straightforward implementation of a general personalization system can use existing Web query languages — an interface can be provided for users to input a set of queries. However, reported experiments [13] suggest that navigation languages are hard to master. Another alternative would be to let users choose from a set of fixed queries (written by an expert), but this is too restrictive and besides, since Web sites change constantly, maintaining such queries can be an arduous task.

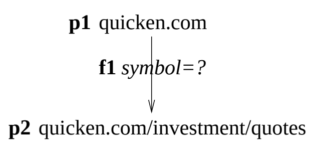


Figure 3. Simplified Graph of Navigation Map for `www.quicken.com`

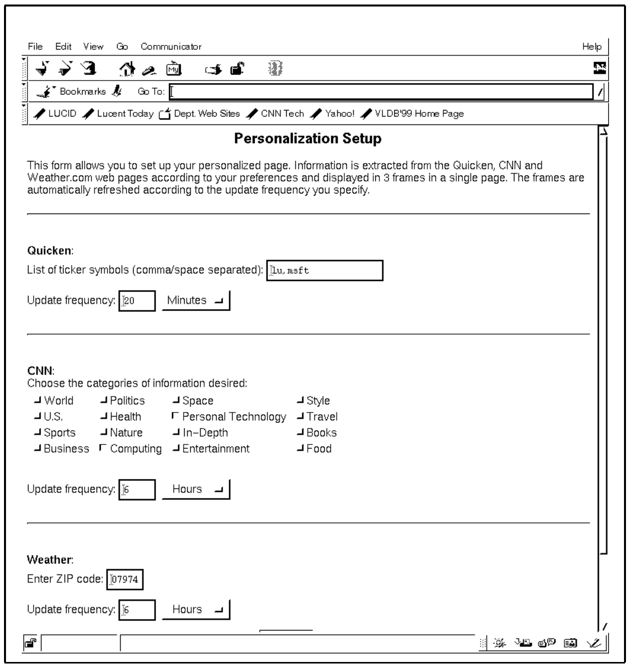
MyOwnWeb uses site descriptions to guide the user in choosing the contents to be included in a personalized page. As described in Section 2, site descriptions encode, in a succinct way, information about the contents, structure, and services offered by a site. Figure 3 shows a (simplified) graph of the navigation map of `www.quicken.com`, and Figure 4 its corresponding site description. Using the description in Figure 4, one is allowed to enter one or more stock symbols to retrieve a table with the stock prices.

In order to build a personalized page, a user is presented with descriptions of Web sites (see Section 2 for details), and may select the nodes (corresponding to Web pages) that contain information relevant to him/her, provide bindings for the queryable attributes, and select specific output attributes to be displayed.[‡] Alternatively, a graphi-

[‡]More complex queries that combine information from multiple sources are also possible (*e.g.*, joins between two sources).

```
quicken : webpage[
   address→"quicken.com"
   title→"QuickenMainPage"
   contents→"quicken1.contents"
   actions↠{a1}
]
a1 : action[
   form→f1[
      cgi→get_quotes
      method→"GET"
      mandatory↠{attr1}
   ]
   source→quicken
   target→p2
]
p2 : webpage[
   address→"quicken.com/investment/quotes"
   title→"Quickenquotes"
   contents→"quicken2.contents"
   extracted_attributes↠ticker_table
] attr1 : simple_attr[
   type→string
   multi_valued→true
   attr_form_name→"symbol"
]
ticker_table : list_attr[
   location→root().string(1, "Bid").ancestor(1, TABLE).
      span(child(2, TR), child(−1, TR))
   item→ticker_item
]
ticker_item : record_attr[
   location→
   fields↠symbol, last, change, volume, bid, ask
]
symbol : atomic_attr[
   location→child(1, A).child(1, #text)
   type→string
]
last : atomic_attr[
   location→child(1, STRONG).child(1, #text)
   type→string
]
change : atomic_attr[
   location→child(1, STRONG).child(1, #text)
   type→string
]
volume : atomic_attr[
   location→child(1, FONT).child(1, #text)
   type→string
]
bid : atomic_attr[
   location→child(1, FONT).child(1, #text)
   type→string
]
ask : atomic_attr[
   location→child(1, FONT).child(1, #text)
   type→string
]
```

Figure 4. Web Objects for Quicken Site


File Edit View Go Communicator Help
Bookmarks Go To:
LUCID Lucent Today Dept. Web Sites CNN Tech Yahoo! VLDB'99 Home Page

Personalization Setup

This form allows you to set up your personalized page. Information is extracted from the Quicken, CNN and Weather.com web pages according to your preferences and displayed in 3 frames in a single page. The frames are automatically refreshed according to the update frequency you specify.

Quicken:

List of ticker symbols (comma/space separated): lu, msft

Update frequency: 20 Minutes

CNN:

Choose the categories of information desired:

World	Politics	Space	Style
U.S.	Health	[✓ Personal Technology]	Travel
Sports	Nature	In–Depth	Books
Business	[✓ Computing]	Entertainment	Food

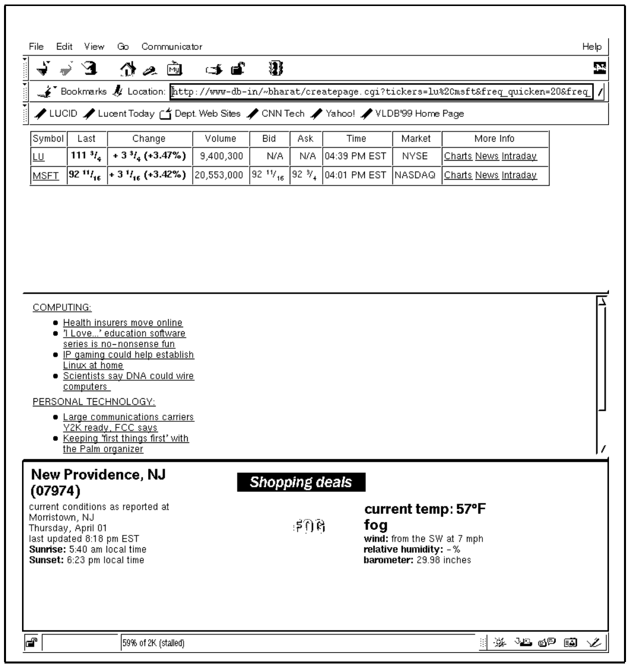
Update frequency: 6 Hours

Weather:

Enter ZIP code: 07974

Update frequency: 6 Hours

Figure 5. Personalization Setup


File Edit View Go Communicator Help
Bookmarks Location:
LUCID Lucent Today Dept. Web Sites CNN Tech Yahoo! VLDB'99 Home Page
Shopping deals

http://www-db-in/~bharat/createpage.cgi?tickers=lu%2Cmsft&freq_quicken=20&freq_

Symbol	Last	Change	Volume	Bid	Ask	Time	Market	More Info
LU	**111 3/4**	**+ 3 3/4 (+3.47%)**	9,400,300	N/A	N/A	04:39 PM EST	NYSE	Charts News Intraday
MSFT	**92 11/16**	**+ 3 1/16 (+3.42%)**	20,553,000	92 11/16	92 3/4	04:01 PM EST	NASDAQ	Charts News Intraday

COMPUTING:

- Health insurers move online
- 'I Love...' education software series is no–nonsense fun
- IP gaming could help establish Linux at home
- Scientists say DNA could wire computers

PERSONAL TECHNOLOGY:

- Large communications carriers Y2K ready, FCC says
- Keeping 'first things first' with the Palm organizer

New Providence, NJ (07974)

current conditions as reported at Morristown, NJ
Thursday, April 01
last updated 8:18 pm EST
Sunrise: 5:40 am local time
Sunset: 6:23 pm local time

current temp: 57°F
fog
wind: from the SW at 7 mph
relative humidity: – %
barometer: 29.98 inches

59% of 2K (stalled)

Figure 6. Personalized Page

cal user interface can be generated from the site description — see Figure 5 for an example.

One important characteristic of Web information is that it may change frequently. News sites such as CNN may be updated every hour, financial sites usually update stock prices every 20 minutes, weather updates come every 3 hours, online classified ads change daily, and so on. In order to accommodate this feature, our system lets users specify for each query the refresh interval, and in case that is not provided, the default is set to whatever smaller refresh frequency appears in the set of pages that contribute to the query.

An advantage of *MyOwnWeb* is that unlike existing personalization services, it does not require a user to *sign up* or provide any personal information. As a result, it is harder for Web sites to track the user's usage patterns. Note that there are anonymization services available on the Web [8] that protect users' privacy, however, users are still required to go through the sign-up process for the various Web sites.

One might argue that the use of site descriptions can be restrictive, since the user is limited to the descriptions provided by the service. Indeed, we do not expect an average Web user to create such descriptions, since even though it is a simple process, it does require some expertise. And although site descriptions can be created semi-automatically by example, it is not feasible to cover the whole Web. In the case a description is not available for a site or service, a user can still include information from that site in a personalized page by simply bookmarking the desired page, using a mechanism we call *smart bookmarking* [2]. Currently, bookmarking dynamically generated pages is not always feasible. For example, if the page has been generated by a POST request, referencing the page again (after the page has expired from the cache) will result in a logical or server error. Moreover, certain sites maintain information about the current session via dynamically assigned session-ids. In that case, all the pages accessed (even via the GET request) will have different URLs if they are accessed at different times. In most cases, such session-ids are not essential, in the sense that removing them from the URL still gives the correct page. However in other circumstances, not specifying them, or using an older session-id, results in an error. Smart bookmarking circumvents these problems by providing a record/replay facility that transparently tracks the user's actions, and saves the sequence of actions to be replayed later.

Implementation issues We now provide an illustrative example of a possible architecture for *MyOwnWeb* in the context of a user browsing on the web with a Java-capable browser[§].

[§] Variations of this architecture are possible, for example, for thin Web clients such as PalmPilot, Internet ready phones etc.

We assume that there is a Web site that acts as a repository of site descriptions. When a user visits that site, a signed applet is loaded into his browser, that lists the site descriptions available in the repository organized hierarchically according to some ontology chosen by the repository designer. The user can choose one or more descriptions from that list, enter a URL for a static page, or create a smart bookmark. When a user chooses a description and the desired output attributes, she is presented with a set of input attributes that need to be specified in order to execute the query against the underlying site. For simplicity, we assume that personalization queries are against individual sites. Queries spanning multiple sites can be built using a variation of query-by-example, which we term *web-by-example*.

On specifying the parameters for each query, along with a refresh frequency, the applet creates a connection with the browser, through which it is able to modify the contents of the page displayed in the browser. The applet then generates a frameset, with each frame in the frameset corresponding to a personalization query specified by the user. The URL specified for each frame contains the complete description of the corresponding query.

When the frameset is initially created, the URL specified by each frame generates a request to the applet. The applet reformulates the request into one or more requests to the corresponding target site, based on its site description. Once a requested page is retrieved by the applet, post-processing of the page is performed to extract the desired information. Before sending a response back to each frame, the applet adds an HTTP META tag to the document with the appropriate frequency at which the the page should be refreshed. An example of a personalized page is given in Figure 6.

Note that variations of this architecture are possible. Even though a small lightweight applet that does not allow any query reformulation is sufficient in many situations, in general, query reformulation and optimization capabilities are necessary. Adding this functionality to the applet can considerably increase its size, making it impractical be loaded every time a user wants to change his personalization settings. An alternative architecture can make use of a Java-based proxy, which the end-user would have installed locally, to perform query formulation and data extraction. The applet described above would simply generate the frameset, with the URLs in each frame referencing a special server address. The proxy, upon receiving an HTTP request containing that server address, performs the query reformulation along with the data access and extraction, and returns the contents to the browser.

It is worth pointing out that *MyOwnWeb* can be easily extended to provide change notifications to the underlying information. This can be achieved by simply comparing the results of a query against the results previously saved for that query. Also note that the ability to perform data extraction allows us to specify complex notification conditions. For example, the user could specify that the frame containing his portfolio should only be refreshed if a specific stock price increased by 10% (or that an email should be sent to that effect).

4. Related Work

Languages such as WebL [9], W3QL [11] and WebOQL [14] have been proposed for retrieving data from Web sources. Users can write programs in these languages to access Web sites and extract their contents. In contrast, our site descriptions are created semi-automatically and users are not required to know specifics of a language. As a byproduct, our wrappers are easier to create and maintain.

W4F [17] provides a human-assisted interface for creating wrappers. Since it uses extensions to the Document Object Model (DOM) [6] to access elements in a page, the generated wrappers are not robust to changes to the structure of the underlying page. In contrast, our system uses XPointer [12] descriptions, which less likely to break when pages are modified. Other systems such as [1, 3] try to infer the structure of a document by combining automatic analysis with user input. These systems assume they are given a Web page and ignore the fact that a complex navigation process might be required to retrieve such page. However, their extraction techniques could be used in our system.

A number of languages have been proposed to capture service descriptions and personalization preferences. WIDL [15] is a proposal pending with the W3C which attempts to describe services offered by Web sites. Unlike site descriptions, WIDL does not have the concept of mandatory/optional attributes, and it has no support for specifying data extraction. Channel definition format (CDF) [7] is another proposal for personalizing information received by a user, by allowing the user to specify channels (essentially, news feeds) that are periodically refreshed. However, it requires changes at the server side to support channels, browser support to display channel information (only Internet Explorer allows this), and the information the user gets is restricted to what the channel has to offer. In contrast, site descriptions require no server support, they are browser independent, and the user can choose whatever information available on the Web.

5. Conclusions and Future Directions

In this paper, we use site descriptions as the basis of a novel approach for creating personalized web pages that improves on existing services in three significant ways: the user can create personalized pages with information from any site; personalized pages may contain information from multiple Web sites; and users are entitled to more privacy as they are not required to sign up or provide any personal information. We also describe a simple extension of the service to support change notifications based on conditions that may span multiple sites.

We are currently investigating possible interfaces for for creating complex Web queries (*i.e.*, allows information from multiple sources to be combined). The main challenge here is to design an interface that is powerful and expressive, yet easy-to-use for the average Web user. We are also looking into high-level declarative language for specifying the layout of personalized pages.

Acknowledgments: We would like to thank Bob Arlein, Narain Gehani, Daniel Lieuwen and Kenneth Rodemann for useful discussions and suggestions on how to improve our system.

References

[1] B. Adelberg. NoDoSe - a tool for semi-automatically extracting structured and semi-structured data from text documents. In *Proc. of SIGMOD*, pages 283–294, 1998.

[2] V. Anupam, J. Freire, B. Kumar, and D. Lieuwen. Smart bookmarks: Saving time by recording web traversals. Technical report, Bell Laboratories, 1999.

[3] N. Ashish and C. Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8–15, 1997.

[4] P. Atzeni, G. Mecca, and P. Merialdo. To weave the web. In *Proc. of VLDB*, pages 206–215, 1997.

[5] H. Davulcu, J. Freire, M. Kifer, and I. Ramakrishnan. A layered architecture for querying dynamic web content. In *Proc. of SIGMOD*, pages 491–502, 1999.

[6] Document object model (dom). http://www.w3.org/DOM.

[7] C. Ellerman. Channel definition format (CDF), Mar. 1997. http://www.w3.org/TR/NOTE-CDFsubmit.html.

[8] E. Gabber, P. Gibbons, Y. Matias, and A. Mayer. How to make personalized web browsing simple, secure and anonymous. In *Proc. Financial Cryptography*, 1997.

[9] T. Kistler and H. Marais. WebL - a programming language for the web. In *Computer Networks and ISDN Systems*, volume 30, pages 259–270, Apr. 1998.

[10] C. Knoblock, S. Minton, J. Ambite, N. Ashish, P. Modi, I. Muslea, A. Philpot, and S. Tejada. Modeling web sources for information integration. In *Proc. of AAAI*, 1998.

[11] D. Konopnicki and O. Shmueli. Information gathering in the World-Wide Web: The W3QL query language and the W3QS. *ACM TODS*, 23(4):369–410, 1998.

[12] E. Maler and S. DeRose. XML pointer language, Mar. 1998. http://www.w3.org/TR/WD-xptr.

[13] G. Mecca, P. Atzeni, A. Masci, P. Merialdo, and G. Sindoni. From databases to web-bases: The araneus experience. Technical Report n. 34-1998, May 1998.

[14] A. Mendelzon, G. Mihaila, and T. Milo. Querying the World Wide Web. *International Journal on Digital Libraries*, 1(1):54–67, 1997.

[15] P. Merrick and C. Allen. Web interface definition language (WIDL), 1997. http://www.w3.org/TR/NOTE-widl.

[16] D. Raggett. HTML Tidy. http://www.w3.org/People/-Raggett/tidy/.

[17] A. Sahuguet and F. Azavant. W4F: a wysiwyg web wrapper factory. Technical report, Univ. of Pennsylvania, 1998.