ELSEVIER

DEQUE: querying the deep web

Denis Shestakov, Sourav S. Bhowmick *, Ee-Peng Lim

School of Computer Engineering, Nanyang Technological University, Singapore 639798, Singapore

Abstract

In this paper, we present a system called DEQUE (Deep WEb QUery SystEm) for modeling and querying the *deep Web*. We propose a data model for representing and storing HTML forms, and a web form query language called DEQUEL for retrieving data from the deep Web and storing them in the format convenient for additional processing. Our system is able to query forms (single and *consecutive*) with input values from relations as well as from result pages (results of querying web forms). We present a novel approach in modeling of *consecutive forms* and introduce the concept of the *super form*. A prototype system has been implemented on a SUN workstation working under Solaris 2.7 using Perl version 5.005_2 and employing MySQL (version 3.23.49) DBMS as the data storage.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Deep Web; Dynamic content; Consecutive forms; Super form; Form query language

1. Introduction

Current-day web crawlers retrieve content only from a portion of the Web, called the *publicly indexable Web* (PIW) [16]. This refers to the set of web pages reachable exclusively by following hypertext links, ignoring search forms and pages required authorization or registration. However, recent studies [12,17] observed that a significant fraction of Web content lies outside the PIW. A

---

* Corresponding author.
  E-mail addresses: assourav@ntu.edu.sg (S.S. Bhowmick), aseplim@ntu.edu.sg (E.-P. Lim).

great portion of the Web is hidden behind search forms (lots of databases are available only through HTML forms). This portion of the Web was called the *hidden Web* in [9] and the *deep Web* in [12]. Pages in the hidden Web are dynamically generated in response to queries submitted via the search forms.

The task of harvesting information from the deep Web can be roughly divided into three parts: (1) Formulate a query or search task description, (2) find sources that pertain to the task, and (3) for each potentially useful source, fill in the source's search form and extract and analyze the results. We will assume further that the task is formulated clearly. Step 2, source discovery, usually begins with a keyword search on one of the search engines or a query to one of the web directory services. The work in [5,8,19] addresses the resource discovery problem and describes the design of topic-specific PIW crawlers. In our study, we assume that a potential source has already been discovered. So we limit our discussion to Step 3.

Retrieving and analyzing relevant information from the deep Web autonomously is a challenging problem. At present, the user is required to manually provide input values to web forms, and extract data from the returned web pages. The manual filling out forms is not feasible and cumbersome in cases of complex queries but these queries are essential for many web-based applications. We illustrate this with an example given below.

Example 1. The `AutoTrader.com` is the largest car Web site with over 1.5 million used vehicles listed for sale by private owners, dealers, and manufacturers. The search page (Available at `http://www.autotrader.com/findacar/index.jtmpl?ac_afflt=none`) is shown in Fig. 1(a). The web page contains a form for searching new or used cars. The form consists of a text box, a "Next" button, a "Make" selection menu and two radio boxes with options "Used Cars" and "New Cars". The form returns a web page containing another form (called *child* form) shown in Fig. 1(b) with fields for additional car search options. The submission of the form on the second page generates a web page containing the results of the query (see Fig. 2).

Suppose the user wishes to find information about "used" Japanese cars made in 1997 within US$ 10,000 and available in "Chicago". Especially, he/she is interested in "black" colored cars. To formulate such a query manually, the user has to fill up the form shown in Fig. 1(a) by

Fig. 1. Form interfaces of Autotrader.com. (a) First search page; (b) second search page.
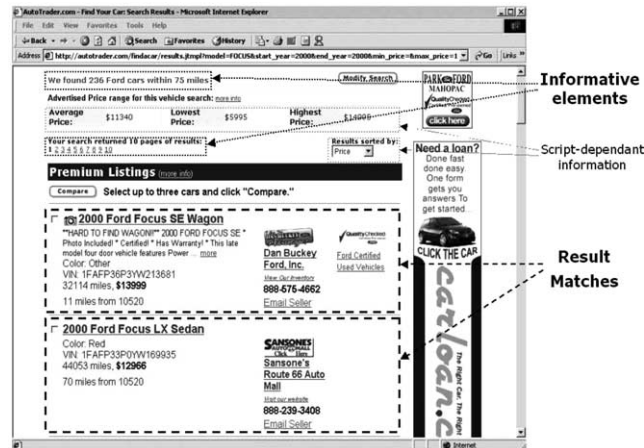
Fig. 2. Result page of Autotrader.com.

specifying the make of the Japanese cars (e.g., Honda, Toyota etc.) and the zip codes of Chicago in the fields labeled "The make I want" and "Near ZIP code" respectively. More importantly, the user has to formulate this query *repeatedly* for all different makes of Japanese car and at least one zip code of Chicago (that is, the user should fill in one of 88 zip codes corresponding to Chicago). To make matters worse, the user has to browse the results returned by each of this query to select all "black" colored Japanese cars. This is because the www.autotrader.com web site does not allow the user to specify the color of a car in the input forms (Fig. 1). Indeed, even for one city and a list containing a small number of car models, the process of filling out the AutoTrader forms, their submissions and looking through the returned results is a very tedious and time-consuming affair.

The above task can be efficiently accomplished by using an automatic form querying system supported by a robust data extraction technique. However, there are several challenges in designing such an automated query mechanism as discussed below.

Automatic filling of forms: The task of automatic filling of forms is a challenging problem in the first place because of the variety of interfaces provided by the web forms. Additionally, the user may not be aware of the values of all fields necessary to fill up the form. For example, the AutoTrader form requires zip code as input. However, it is natural to assume that the zip code(s) of a city is unknown to the user. Then, the query performing the specified task should retrieve the city zip codes from a zip database and substitute necessary input values for the corresponding form field.

Extraction of results: Another complex problem is to automatically extract the query results from the result pages since useful data is embedded into the HTML code. The search and the extraction of the required data from these pages are highly complicated tasks as each web form interface is designed for human consumption and, hence, has its own method of formatting and layout of elements on the page. For instance, Fig. 2 depicts the original AutoTrader result page with formatting and non-informative elements (such as banners, advertisements, etc.). Accordingly, the extraction tools must be able to filter out the relevant contents from the pages.

`Navigational complexity:` Dynamically generated web pages may contain links to other web pages containing relevant information and consequently it is necessary to navigate these links for evaluation of their relevances. Also, navigating such web sites requires repeated filling out of forms many of which themselves are dynamically generated by the server-side programs as a result of previous user inputs. For example, the `AutoTrader` web site produces a web page containing the results after at least two successful form submissions (Fig. 1(a) and (b)). These forms are collectively called *consecutive forms*.

`Client-side programs:` Lastly, the client-side programs may interact with forms in arbitrary ways to modify and constrain the form behavior. For instance, a text box control containing the total sales price in an order form might be automatically derived from the values in other text boxes by executing the client-side script whenever the form is changed or submitted to the server. Programs written in JavaScript are often used to alter the behavior of forms. Unfortunately, it is computationally hard to automatically analyze and understand such arbitrary programs.

In this paper, we discuss a query system for the deep Web called `DEQUE` [1] (Deep WEb QUery SystEm) and address some of these above challenges. There are three steps to be performed for querying a web form within the framework of `DEQUE`. Firstly, a web form to be queried or several forms (if multiple form submission is required to obtain the result pages) should be parsed and stored in a *form database* in accordance with the data model outlined in Sections 3 and 4. Storing is advisable as it quickens the query process. Nevertheless, non-stored forms can also be queried. In the second step, a form query specified by the user is passed to `DEQUE` for validation and submission. A query is formulated in a web form query language called `DEQUEL` that allows the user to assign more values to the form fields than it is possible when the form is manually filled out. Additionally, data from relational tables or data obtained from querying other web forms can be used as input values. The final step is retrieval of all result pages with query results and extraction of useful data from these pages according to the extraction conditions (if they are specified in the form query).

The rest of the paper is organized as follows: Section 2 discusses related research in this area. Sections 3 and 4 present a data model for representing web forms. In Section 5, we discuss how the result pages are represented. Section 6 unveils the syntax of `DEQUEL`. We discuss the implementation of `DEQUE` in Section 7 and highlight some experimental results. Finally, Section 8 concludes the paper.

2. Related work

The impetuous growth of the Web has stimulated considerable interest in the study of Web crawlers [5,6]. These works have addressed various issues in the design and implementation of PIW crawlers. Existing Web crawlers find it extremely difficult to maintain current indices using exhaustive crawling due to the increasing size and dynamic content of the Web. To overcome this, the focused crawlers [5,8] aim to search and retrieve only the subset of the PIW that pertains to a specific topic of relevance. The focused crawling approach is worth to mention in the context of

---

[1] Pronounced as "deck".

harvesting information from the deep Web since it can be used to identify target sites for the hidden Web crawlers [23].

Retrieving and querying web information have received considerable attention by the database research community [9]. W3QS (WWW Query System) [14,15] is a project to develop a flexible, declarative, and SQL-like Web query language called W3QL. The W3QS offers mechanisms to learn, store, and query forms. However, it supports only two approaches to complete a form: either using the knowledge of past form-filling or some specific values. No other form-filling methods, such as filling out a form using multiple sets of input values or values obtained from the queries to relational tables are provided. In addition, it does not support the modeling and querying of the consecutive forms. Furthermore, W3QS is not flexible enough to get all the result pages returned by a form.

ARANEUS introduced a set of tools and languages for managing and restructuring data coming form the Web [3,20]. The data model of the project, called the ARANEUS Data Model (ADM), models the internal structure of web pages as well as the structure of the web sites. Based on ADM, two languages, ULIXES and PENELOPE, are designed to support the querying and restructuring process. It is noted that ADM's form representation is rather simple as it has omitted many important field types, such as checkbox, select, and others. As a result, the ADM's form type may not be able to represent most of the existing forms and does not support the notion of consecutive forms. In our proposed web form model, we support all these features.

In [11], Davulcu et al. proposed a three-layered architecture for designing and implementing a database system for querying forms. The lowest layer, *virtual physical layer*, aims to provide navigation independence, making the steps of retrieving data from raw web sources transparent to the users. Data collected from different web sites may have semantic or representational discrepancies. These differences are resolved at the *logical layer* that supports site independence. The *external schema layer*, extending the Universal Relation (UR) concept [18], provides a query interface that can be used by naive web users. Unlike the other projects, the work by Davulcu et al. addresses the problem of navigating *consecutive forms*. Our work differs from the work by Davulcu et al. in two aspects. First, we propose more advanced web form representation and user-friendly language for defining form queries. Second, we do not treat the form query results as relations. For most web pages it is extremely difficult to transform automatically the web page content into relation tables. Hence, we limit our extraction process to the extraction of useful data. We represent the result pages as containers of the result matches, each of which containing informative text strings and hyperlinks.

Raghavan and Garcia-Molina [23] propose a way to extend the crawlers beyond the publicly indexable Web by giving them the capability to fill out Web forms automatically. Starting with a user-provided description of the search task, HiWE learns from successfully extracted information and updates the task description database as it crawls. There are some limitations of the HiWE design that if rectified, can significantly improve the performance. The first limitation is the inability to recognize and respond to simple *dependencies* between the form elements (e.g., given two form elements corresponding to states and cities, the values assigned to the "city" element must be cities that are located in the state assigned to the "state" element). This problem is closely connected with the querying of consecutive forms. Another limitation of the HiWE is the lack of support for data extraction from the result pages.

The Web wrapping problem, i.e., the problem of extracting structured information from HTML documents, has spurred a great amount of work that can be classified into two categories, depending on whether the HTML input is regarded as a sequential character string or a pre-parsed document tree. The STALKER approach for wrapper construction enables users to turn web pages into relational information sources [13,21]. STALKER is a machine learning based approach where *Embedded Catalog* ($\mathscr{EC}$) formalism is used to describe the content of a web page. The $\mathscr{EC}$ description of a page is a tree-like structure in which the leaves represent the relevant data. The internal nodes (elements) of the $\mathscr{EC}$ tree represent *lists* of k-tuples, where each item in the k-tuple can be either a leaf l or another list L (in which case L is called an embedded list). However, the STALKER was designed to extract data from a single web page and cannot handle a set of hyperlinked pages (the results generated by querying forms).

In [4], Baumgartner et al. presented techniques for supervised wrapper generation and automated web information extraction, and a system called Lixto implementing these techniques. Lixto assists the user to semi-automatically create wrapper programs by providing a fully visual and interactive user interface. Internally, this functionality is reflected by the logic-based declarative language Elog. In our study, we consider the wrapping problem to provide the user with more visual and clear presentation of the query results than an ordinary chain of web pages returned by a form submission. DEQUE does not require any interaction with the user during the wrapper generation process; that is, data from the result pages are extracted in a completely automatic way.

3. Modeling of a single HTML form

In this section, we discuss the data model for representing a single HTML form. Since for different HTML forms the nature and type of the layout markup are different, the web application should represent web forms in a uniform manner.

3.1. HTML forms

An HTML form is a section of a document containing normal content, markup, special elements called controls (checkboxes, radio buttons, menus, etc.), and labels on those controls. Users generally "complete" a form by modifying its controls (entering text, selecting menu items, etc.), before submitting the form to an agent for processing (e.g., to a Web server, to a mail server, etc.). These form controls are also known as *form fields*. An HTML form is embedded in its web page by a pair of "begin" and "end" <FORM> tags. Each HTML form contains a set of form fields and the URL of a server-side program (e.g., a CGI program) that processes the form fields' input values and returns a set of *result pages*.

There are three essential attributes of the FORM element that specify how the values submitted with the form are processed: the value of the action attribute corresponds to the URL of a form processing agent (server-side program), the HTTP method used to submit the form is defined by the method attribute, and the enctype attribute specifies the content type used for the form submission. For a web form F, we define the submission information of F (denoted as *subinfo*(F)) as three string elements specifying the form action, the HTTP method and the content type. That

is, *subinfo*(*F*) = {*action*, *method*, *enctype*}. For example, the submission information of the form shown in Fig. 1(a) is given by: *subinfo*(*FindCarForm*) = {"*http://autotrder.com/findcar/findcar_form2.jtmpl?ac_afflt=none*", "*get*", "*application/x-www-form-urlencoded*"}. Note that two forms F_1 and F_2 have the *same* submission information if *subinfo*(F_1) = *subinfo*(F_2).

3.2. Form fields

The user fills out a form by associating a value or piece of text with each field of the form. A form field can be any one of the standard input objects: selection lists, text boxes, text areas, checkboxes, or radio buttons. These objects are defined by the BUTTON, INPUT, SELECT, and TEXTAREA tags. The name attribute of the above tags defines the name of a form control (also called *fieldname*). In our study, we do not consider the *file select*, *object* and *reset button* controls as they have no use in queries to most of the searchable databases. The detailed description of each type is documented in the HTML 4.01 specification [1].

A form field can be represented by the following attributes:

- *Field domain*: The field domain is a set of values where each value is a character string which can be associated with the corresponding form field. Some form fields have predefined domains, where the set of values are embedded in the web page with a form. Other fields have undefined domains (e.g., set of all text strings with specified length) from which their values can be chosen.
- *Field label*: The form fields are usually associated with some descriptive text [2] to help the user understand the semantics of a field. The field label is a string containing the descriptive information about the corresponding form field.
- *Initial field set*: Each field has initial value(s) (defined or undefined) which can be submitted with a form if the user does not modify a field through "completing" a form. The initial field set is a set of the field's initial values. It is clear that for each form field an initial field set is a subset of the field domain.

Formally, given a field *f* of a web form *F*, *label*(f_F) denotes the field label of *f*. Similarly, the field domain and the initial field set of *f* is denoted by *domain*(f_F) and *iset*(f_F) respectively. Note that *iset*(f_F) ⊂ *domain*(f_F). We say that two form fields *f* and *f'* are the same if *formname*(*f*) = *formname*(*f'*), *type*(*f*) = *type*(*f'*), and *domain*(*f*) = *domain*(*f'*).

The field domain, field label and the initial field set are defined for each field in a web form. For example, the field label of the "*address*" field is equal to "*Near ZIP code*" (see Fig. 1(a)). Also, *domain*(*address*) = {*s*, *length*(*s*) ⩽ 5}, where *s* is a character string, and *iset*(*address*) = {∅}. The field label, domain and the initial set of the "*make*" field in Fig. 1(a) is given by the following: *label*(*make*) = {"*The make I want is*"}, *domain*(*make*) = {"*Acura*", "*Alfa Romeo*", "*AMC*", "*Audi*",...}, and *iset*(*make*) = {"*Acura*"}. Note that the values of the menu choices visible in a browser are not generally equal to the values actually submitted with a form. Since the visible values are more informative, we consider a field domain as a set of visible menu choices. In the same manner, an initial field set is a set of pre-selected choices visible in a browser.

---

[2] Besides description of a form field such texts often indicate if a field is optional or required.

4. Modeling of consecutive forms

In the preceding section, we discussed how to model a single form. We now elaborate on the modelling of *consecutive forms*. Consider an HTML page containing one or more HTML forms. Each form is called the *root form*. A response page is a page received in response to the root form submission. In some cases, after submitting a *root form*, the returned page (or response page) contains another form (called the *child form*) which needs to be filled out. Similarly, after submitting the child form, the returned page may contain another child form to be filled out and so on. All the child forms are collectively called *descendant forms* for the given root form. The root and its descendant forms are collectively called *consecutive forms*. The submitted form is also the *parent form* for the following child form. Each descendant form is completely defined by its parent form, the values filled-in and the time of the parent form submission. It also means that we always know the root form for each descendant form. A typical example of consecutive forms is the `Auto-Trader` car search interface. After submission of the form (root form) shown in Fig. 1(a), the returned page contains another form (child form) depicted in Fig. 1(b). The form in Fig. 1(a) is also the parent form for the form in Fig. 1(b). Similarly, the latter is the descendant form for its parent form.

4.1. Form type and location

There are several differences between the root and its descendant forms. One of them is the URL of a web page that contains a form. As a matter of fact, the main reason to have the URL of a page containing a form is that an HTML code related to a form often specifies only the relative URL of the server-side program. Thereby, the page URL must be known to compose the absolute URL of the server-side program. The page URL of a child form can always be defined based on the URL of the a parent form, the parent forms submission information, the values submitted with the parent form and the submission time [3]. Thus, we do not consider the URL of a page containing descendant forms as it can be easily constructed using the URL of the page containing the root form, and the processing information (submission information, submission data set, and submission time) of all forms beginning with the parent form and ending with the root form. In our study, we presume that the page URL of a root form is given by the user.

Formally, given a web form F, *formtype*(F) specifies whether a form is a root or descendant form. Ff F is a root form, then the *pageurl*(F) is the absolute URL of a page that contains F. Otherwise, it is equal to "*null*". For example, the `AutoTrader` form shown in Fig. 1(a) is the root form and *pageurl*(*AutoTrader*) = "*http://autotrader.com/findacar/index.jtmpl?ac_afflt=none*". Fig. 1(b) depicts the `AutoTrader` child form for which: *formtype*(*AutoTrader$_2$*) = "*descendant*" and *pageurl*(*AutoTrader$_2$*) = "*null*".

---

[3] The URL of a response page equals to the URI of the server-side program if the POST HTTP method is used to send the submission data set to the program and to the URI of the server-side program with appropriately appended submitted values in case the GET method is used.
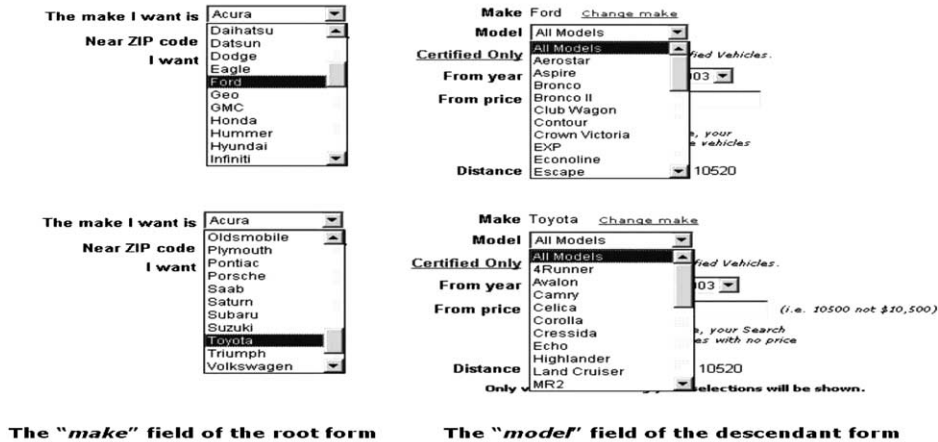
The "*make*" field of the root form   The "*model*" field of the descendant form

Fig. 3. Form field dependency.

4.2. Issues in modeling of consecutive forms

There are some non-trivial issues concerning the modeling of consecutive forms. First, most consecutive forms have *dependencies* (called *form dependencies*) between the root and the descendant forms. For example, the `AutoTrader` search interface (see Fig. 1) requires two consecutive forms to be fill-out, if the user searches for "*Used Cars*", and three forms, if the "*New Cars*" option is chosen in the root form. Also, a *dependency* (called the *form field dependency*) may exist between the form fields. For instance, if the user selects "*Toyota*" make in the "*make*" menu, then only "*Toyota*" models are available for the "*model*" select field of the `AutoTrader` child form in Fig. 1(b). The dependency between the "*make*" and the "*model*" fields is shown in Fig. 3. Choice of different car brands in the root form generates the child forms with different option values (corresponding to model names of the chosen car brand) for the "*model*" select field.

Second, each root form can generate a large number of different web pages containing the child forms. How these forms look like often depend not only on the values filled out in the root form but also on the submission time. In particular, it is irrational to store all possible child forms. For example, at least 48 different child forms (search for "*Used Cars*") [4] are generated by the server-side program related to the `AutoTrader` root form (see Fig. 1). All these forms are highly similar to one another. The key difference is in the "*model*" select field as the forms have different option values for the "*model*" field (Fig. 4).

4.3. Submission data set of consecutive forms

When the user submits an HTML form, a user agent (web browser, etc.) processes it as follows. First, a form data set (a sequence of control name/value pairs) based on the values filled-in by the user is created. Then, the form data set is encoded to the content type specified by the `enctype`

---

[4] 48 is the number of options in the "*make*" menu. Note that the submission of data sets containing different values for the text "*address*" field does not return different child forms.
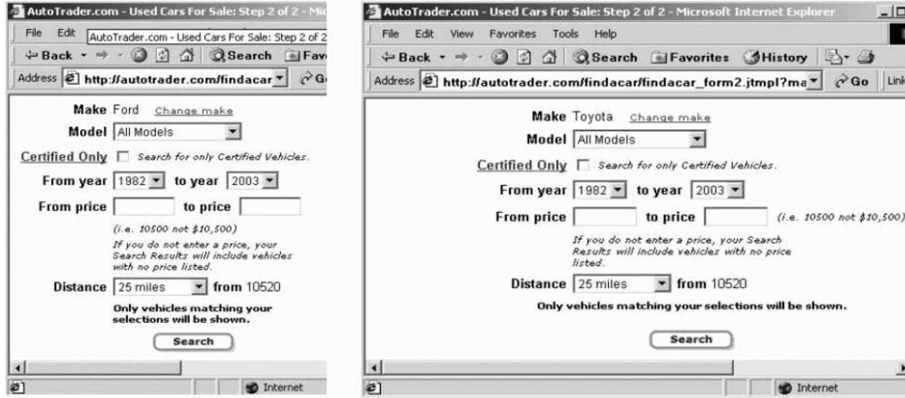
Fig. 4. AutoTrader child forms.

attribute of the FORM element. Finally, the encoded data is sent to a processing agent (server-side program) designated by the action attribute using the HTTP protocol (GET or POST) specified by the method attribute. The server-side program processes the form data set and returns a web page as the result of the form submission. Formally, let F be a web form with form fields $f_1, f_2, \ldots, f_n$. Let $domain(f_i)$ be the domain of the field f_i. Then, $S = \{s_1, s_2, \ldots, s_n\}$ is the *submission data set* of F, if $\forall i = \overline{1, n} : s_i \in domain(f_i)$. We say that two submission data sets $S = \{s_1, \ldots, s_n\}$ and $S' = \{s'_1, \ldots, s'_n\}$ of a web form F are *equal* to each other, if $\forall i = \overline{1, n} : s_i = s'_i$. The web page with a form $F^{(S,t)}$ returned by the server-side program as the result of submission of the form F_p at time t is expressed as $F_p(f_{p,1}, \ldots, f_{p,n}) \xrightarrow{S,t} F_c^{(S,t)}(f_{c,1}^{(S,t)}, \ldots, f_{c,k}^{(S,t)})$, where F_p, $F_c^{(S,t)}$ and S are the parent form with form fields $f_{p,1}, \ldots, f_{p,n}$, the child form with fields $f_{c,1}^{(S,t)}, \ldots, f_{c,k}^{(S,t)}$, and the submission data set of F_p respectively. Also, the set of form fields of the child form is defined as $fieldset(S, t) = \{fieldname(f_{c,1}^{(S,t)}), \ldots, fieldname(f_{c,k}^{(S,t)})\}$. We can describe several submissions of consecutive forms in the following way: $F_p \xrightarrow{S_1,t} F_{c_1} \xrightarrow{S_2,t} F_{c_2} \xrightarrow{S_3,t} \cdots \xrightarrow{S_{m-1},t} F_{c_{m-1}} \xrightarrow{S_m,t} R$. This expression shows that m forms ($F_p, F_{c_1}, \ldots, F_{c_{m-1}}$) were submitted to obtain the result web page R where t is the submission time of the last child form submission ($F_{c_{m-1}}$). We omit the superscript (S,t) as well as consecutive forms' fields to make the expression more compact.

Example 2. Consider the submission of the AutoTrader root form (named as *Autotrader*) shown in Fig. 1. Suppose we intend to search for "*Used Cars*". Then *AutoTrader*(*make*, *address*, *search_type*, *field_1*, *ac_afflt*, *borschtid*) $\xrightarrow{S,t}$ *AutoTrader*₂, where $S = \{$"*Ford*", "*10520*", "*Used Cars*", "*submit*", "*none*", "*21532053581465403997*"$\}$ and $t = $ "17 *June* 2002/6:34*pm*". The result of submission is the page containing the AutoTrader child form (named as *AutoTrader*₂) as shown in Fig. 1(b). This form contains eight visible and five hidden (invisible) form fields. The set of fields of the *AutoTrader*₂ form is *fieldset*(*S*,*t*) = {*model*, *certified*, *start_year*, *end_year*, *min_price*, *max_price*, *distance*, *field_1*, *advanced**, *advcd_on**, *make**, *address**, *search_type**} [5]. Note that the hidden fields are often used to transmit the information about submitted values

---

[5] Hidden fields are marked by asterisk.

from one consecutive form to another. Thus, the initial values of the hidden fields *make*, *address*, *search_type* in the *AutoTrader$_2$* form are equal to the values assigned to the fields *make*, *address*, *search_type* of the *AutoTrader* form respectively ("*Ford*", "*10520*", "*Used Cars*").

4.4. Form union

We are now ready to discuss how to represent the child forms. According to the HTML specification, each form must have the `action` attribute in the `FORM` tag. This attribute specifies the URL of the server-side program to which the form contents will be submitted (if this attribute is absent, then the current document URL is used). Since web pages generated by the same server-side program are expected to be very similar, we shall combine forms with the same submission information. A *form union* allows us to represent multiple forms sharing the same server-side program as one form. At the same time, each form included in a form union is easily accessible.

Definition 1 (*Form Union*). Consider a form F (root or descendant) and its two child forms $F^{(S_1,t_1)}$ and $F^{(S_2,t_2)}$ such that $F \overset{S_1,t_1}{\to} F^{(S_1,t_1)}(f_1^{(S_1,t_1)}, \ldots, f_k^{(S_1,t_1)})$ and $F \overset{S_2,t_2}{\to} F^{(S_2,t_2)}(f_1^{(S_2,t_2)}, \ldots, f_m^{(S_2,t_2)})$. Suppose that the fields of $F^{(S_1,t_1)}(f_1^{(S_1,t_1)}, \ldots, f_p^{(S_1,t_1)})$ are the same as the fields of $F^{(S_2,t_2)}(f_1^{(S_2,t_2)}, \ldots, f_p^{(S_2,t_2)})$. Then, the **union** of $F^{(S_1,t_1)}$ and $F^{(S_2,t_2)}$, denoted by $W = F^{(S_1,t_1)} \cup F^{(S_2,t_2)}$, is defined if and only if $subinfo(F^{(S_1,t_1)}) = subinfo(F^{(S_2,t_2)})$. The forms $F^{(S_1,t_1)}$ and $F^{(S_2,t_2)}$ are called the *component forms* of W. The form union W is considered as the child form of F where (1) The form fields of W are $f_1^{(S_1,t_1)}, \ldots, f_p^{(S_1,t_1)}, f_{p+1}^{(S_1,t_1)}, \ldots, f_k^{(S_1,t_1)}, f_{p+1}^{(S_2,t_2)}, \ldots, f_m^{(S_2,t_2)}$ and (2) $subinfo(W) = subinfo(F^{(S_1,t_1)})$. Also, note that $F^{(S,t)} = F^{(S,t)} \cup F^{(S,t)}$.

Example 3. Let us consider the two child forms of the `AutoTrader` root form shown in Fig. 4. They are results of submission of the root form depicted in Fig. 1(a) with the submissions data sets $S_1 = \{$"**Ford**","10520","*Used Cars*", "*submit*", "*none*", "21532053581465403997"$\}$, $t_1 = $ "17 *June* 2002/6:34*pm*" and $S_2 = \{$"**Toyota**", "10520", "*Used Cars*", "*submit*", "*none*", "21532053581465403997"$\}$, $t_2 = t_1$, $AutoTrader \overset{S_1,t_1}{\to} AutoTrader_2^{(S_1,t_1)}$ and $AutoTrader \overset{S_2,t_2}{\to} AutoTrader_2^{(S_2,t_2)}$. These forms are the same except one select field with name "*model*" and one hidden field with name "*make*" for which $domain(model^{(S_1,t_1)}) = \{$"*All models*","*Aerostar*",…, "*Escort*",…$\}$ (i.e., all "**Ford**" models), $domain(make^{(S_1,t_1)}) = \{$"**Ford**"$\}$, $domain(model^{(S_2,t_2)}) = \{$"*All models*","*4Runner*",…,"*Corolla*",…$\}$ ("**Toyota**" models), and $domain(make^{(S_2,t_2)}) = \{$"**Toyota**"$\}$. Hence, the union form $AutoTrader_2^{union} = AutoTrader_2^{(S_1,t_1)} \cup AutoTrader_2^{(S_2,t_2)}$ is simply a copy of $AutoTrader_2^{(S_1,t_1)}$ (or $AutoTrader_2^{(S_2,t_2)}$) form and two additional form fields $model^{(S_2,t_2)}$ and $make^{(S_2,t_2)}$ (or $model^{(S_1,t_1)}$ and $make^{(S_1,t_1)}$). Fig. 5 depicts the representation of the $AutoTrader_2^{union}$ form.

Note that the form union described in the above example contains nine visible and six invisible form fields. Note that each child form contains eight visible and five invisible fields. Thus, form unions allows us to store similar (sharing the same submission information) forms more effectively
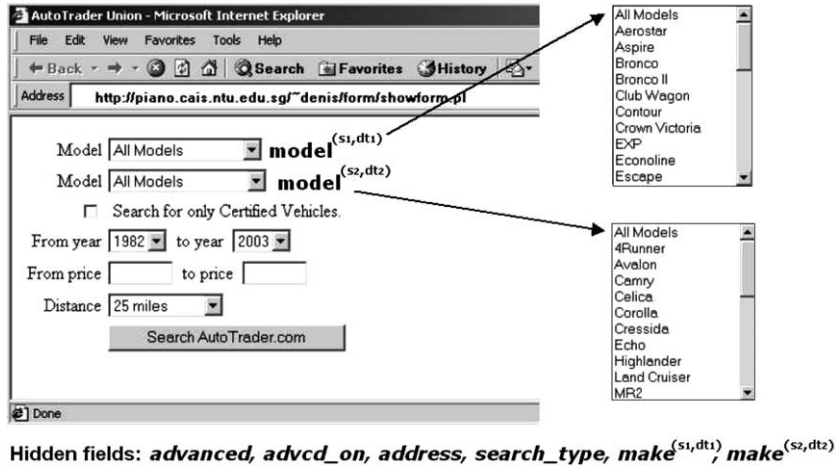
Fig. 5. Union of two `AutoTrader` child forms.

(15 fields instead 26 in this example). On the other side, if necessary, we can easily obtain any component of a form union as we also store the submission data set that generates the child form and a set of form fields of the child form.

4.5. Super form

In order to query the consecutive forms, our web form model must know the structure of all forms to be filled-out. As we noted above, most descendant web forms have very similar structure. This similarity motivates us to combine certain descendant forms into one form called the *super form*. In the previous section, we described how to construct a union of two forms. Note that a form union is also a form, and it shares the same submission information as its two component forms. Thus, a set of forms with the same submission information can be combined into one form. All the child forms sharing the same `action` attribute (*component forms*) are combined into one form that contains all the component forms' fields. At the same time, identical fields of the child forms are presented in the super form only once. Formally,

Definition 2 (*Super Form*). Consider a web form F with all possible submission data sets $S = \{(S_i,t)\}$, where t is an arbitrary date/time. Assume that a submission of S at different times results in the same child form on the response page. Let $F^{(S_k,t)}$ be a child form of F with submission data set (S_k,t), where $(S_k,t) \in S$. A super form F_{super} (or $F_{super}^{(S_k,t)}$) is given by:

$$F_{super} = F^{(S_k,t)} \cup \bigcup_{\forall S_j:(S_j,t)\in S:subinfo(F^{(S_j,t)})=subinfo(F^{(S_k,t)})} F^{(S_j,t)}$$

The form $F^{(S_k,t)}$ is called a *base form* of the super form $F_{super}^{(S_k,t)}$. Note that $F_{super}^{(S_k,t)} = F_{super}^{(S_m,t)}$, if $subinfo(F^{(S_k,t)}) = subinfo(F^{(S_m,t)})$.

Example 4. Consider the `AutoTrader` root form shown in Fig. 1(a). The submission of the `AutoTrader` form with the following submission data sets: $\{(S_i^{(used)}, t)\}$, where $S_i^{(used)} \in \{\langle v_1, v_2, \text{``Used Cars''}, v_4, v_5, v_6 \rangle \mid v_1 \in domain(make), \quad v_2 \in domain(address), v_4 \in domain(field\_1), v_5 \in domain(ac\_affl), v_6 \in domain(borschtid)\}$ returns web pages containing child forms (see forms in Fig. 4) with identical submission information. The generated child forms differ from each other in the visible "*model*" field, domain values of which correspond to the chosen "*make*" in the root form, and the invisible "*make*" field that simply contains the chosen "*make*". The super form $F_{super}^{(used\,cars)}$ is equal to its base form (see Fig. 4) plus 47 additional "*model*" select fields [6] and 47 "*make*" hidden fields.

The submission of the `AutoTrader` form with the following submission data sets: $\{(S_i^{(new)}, t)\}$, where $S_i^{(new)} \in \{\langle v_1, v_2, \text{``New Cars''}, v_4, v_5, v_6 \rangle\}$ returns the second series of child forms sharing the same submission information [7]. Thus, we can build the second super form $F_{super}^{(new\,cars)}$.

Among other things, dependencies between consecutive forms may be studied on the ground of a super form. Indeed, the analysis of submission data sets and generated child forms allows us to find existing dependencies. For instance, the constructed super form $F_{super}^{(used\,cars)}$ allows us to determine the dependence between the "*make*" field of the root form and the "*model*" field of any child form. This helps us to validate values in form queries. For instance, the query "*Find all Ford Corolla cars*" results in the web page with zero results as "*Corolla*" model does not belong to the "*Ford*" make. On the other side, the super form $F_{super}^{(used\,cars)}$ contains the names of all possible used car models. Similarly, the super form $F_{super}^{(new\,cars)}$ may be used to find the domain of all new car models. This domain does not necessarily be identical to the domain of used car models as some car models are not manufactured in recent years ("*Ford Bronco*" or "*Toyota Matrix*").

A super form significantly simplifies the querying and storage of consecutive forms. Indeed, to perform complex query on two consecutive forms we need to specify only two forms (root and super) instead of large number of combinations of a root and its child forms. For example, the simple search for "*Used Ford and Toyota cars within 10520 ZIP area*" requires four form submissions (two times of the root form in Fig. 1(a) and one time each of the child forms shown in Fig. 4). Hence, three forms should be stored and then specified in the query. However, the super form based on any of these child forms reduces the number of forms to be specified to only two.

Efficient storage is another advantage of our super form-based approach. The differences among the descendant forms are often minor since the same server-side program generates them. A super form allows us to store only one base form and the differences between the base form and all other forms with the same submission information. For instance, the form union presented in Example 3 requires storing only 15 fields instead of 26. Thus, combining even two forms significantly reduce the number of fields stored.

---

[6] 47 is the number of options in the "*make*" menu of the `AutoTrader` root form exclusive of one option that corresponds to the base form.

[7] Note that the submission information of child forms generated by $\{(S_i^{(new)}, t)\}$ is different from the submission information of child forms generated by $\{(S_i^{(used)}, t)\}$.

4.6. Form extraction

DEQUE presumes that all forms to be queried are stored in the form database according to the web form data model described earlier. This section describes the extraction of a web form from an HTML page.

An HTML page containing a form may be specified by the user through the GUI depicted in Fig. 6. Since DEQUE allows us to perform queries on consecutive forms, each response page must be examined for the presence of web forms. Thus, for each web page (specified through the GUI or retrieved from the Web) DEQUE constructs a logical tree representation of the structure of an HTML page based on the document object model (DOM) [2]. Next, the tree is forwarded to the following three extraction submodules that are responsible for the form extraction.

Form element extractor: The form element extractor analyzes a tree to find the nodes corresponding to the FORM elements. If one or more elements exist, then the *pruned tree* is constructed for each FORM element. For such tree construction we use only the subtree below the FORM element and the nodes on the path from the FORM to the root.

The extractor also retrieves data related to a form and its fields from the pruned tree in accordance with the web form model presented earlier. It should be noted that the visible values as well as the invisible values are retrieved. For example, consider the element <option value= "ALFA">Alfa Romeo</option>. "*Alfa Romeo*" and "*ALFA*" are stored as visible and invisible option values respectively. The situation is worse with radio and checkbox fields. The visible values related to these types of form fields are not embedded in the INPUT elements that define such form fields. In most cases, we can find one of the visible values of a radio/checkbox field

Fig. 6. Form storage GUI.

directly after the INPUT tag. For instance, "*New Cars*" as one of the visible values of the "*search_type*" field can be easily extracted from the following HTML code: `<input type="radio" name= "search_type" value="new" onClick="changeList(1)" > New Cars`. Unfortunately, there are web forms with more complex HTML markup in which the distance (in terms of the number of HTML tags) between a text element corresponding to a visible value and the INPUT element may be more than one tag.

Label extractor: In DEQUE, the label extractor submodule is responsible for extraction of the field labels. The label extractor (by default) begins with ignoring font sizes, typefaces and any styling information, so the corresponding pruned tree is simplified. We use the *visual adjacency* approach introduced in [22] to extract labels. The key feature of this approach is that when the HTML code is rendered by the browser, the relationships between the fields and their labels or fields and their visible values must be obvious to the user. In other words, irrespective of how the page with the form is formatted, the phrase "*The make I want is*" (form label) or "*Used Cars*" (visual value) in Fig. 1(a) must be visually adjacent to the select menu or one of the radio field choices respectively. Similarly, the text "*Near ZIP code*" must be visually adjacent to the corresponding textbox widget. We use the following heuristic for identifying the label of a given form field (an analogous heuristic is used for domain values of radio/checkbox fields).

- Identify the pieces of text, if any, that are visually adjacent to the form field. If a text piece contains less than eight words [8] and the distance (in terms of HTML tags) between a text and a widget is less than eight, then we consider each piece of text as possible candidate to be a label of the form field. For example, consider Fig. 7. This piece of HTML code is a part of the Amadeus form at www.amadeus.net/home/en/home_en.htm. Observe that the tag distance between the text "*Departing from:*" and the text box is seven tags. For each candidate we compute the actual pixel distances between the form widget and the candidate text pieces. We use two JavaScript functions that returns the element's real X and Y coordinates. The distance between the text element and the form field can easily be computed using these coordinates.
- If there are candidates to the left and/or above the form field, then we drop the candidates to the right and below. Note that the visible values of the radio/checkbox fields are usually to the right of the form field. Thus, we prefer the candidates to the right when extract domain values of the radio/checkboxes fields.
- If there are still two candidates remaining, the text piece rendered in bold or using a larger font size is chosen. Apparently, this step is omitted if the label extractor ignores the styling information.
- If two candidates are still not resolved, then one of them is picked at random.

JS function extractor: This module [9] is responsible for extracting the JavaScript functions from the web pages. The main reason to extract such kind of functions is that the values before submis-

---

[8] Most labels are either short words or short phrases.
[9] In our implementation we consider only JavaScript as the client–server script language.

```
   <td class=small valign="bottom" bgcolor="#FEF0DE">Departing from :</td>
1. <td valign=bottom bgcolor="#FEF0DE" class="small">Depart Date :</td>
2. <td valign=bottom bgcolor="#FEF0DE" class="small"> Search for: </td>
3. <td bgcolor="#FEF0DE" valign="bottom" colspan="-1">
4. <p align="left"> </p></td></tr>
5. <tr bgcolor=#d2e1ff>
6. <td valign="top" class="small" bgcolor="#FEF0DE"> </td>
7. <td valign="top" class="small" bgcolor="#FEF0DE">
   <input maxlength=20 name=D_City size=12  style="width:110px">
```

Fig. 7. Tag distance between form label and field.

sion to a server-side program would be checked in the same manner as in the case of the manual form filling-out. Additionally, nowadays web pages often contain scripts that define the domain of values for some form field based on the chosen value of another field. In our work, we consider only extraction of the client-side scripts. The content of the nodes related to the SCRIPT tags is considered for the function codes that are triggered if some form or field events occur. For example, after extraction of the form event information by the form element extractor *formevent*(*Auto-Trader*) = {"*onsubmit*", "*return validateData*();"}, the JS function extractor searches the content of the SCRIPT tags or it searches the content of the file that may be linked to the web page as a container of the client-side scripts for this page for the function named "*validateData*" and retrieves it.

5. Representation of result pages

In this section, we first discuss how the results returned by a deep Web query are represented in DEQUE and then describe an approach used to extract data from the result web pages.

5.1. Result navigation

Perhaps the most common case is that a web server returns results a bit at a time, showing 10 or 20 result matches per page. Usually there is a hyperlink or a button to get to the next page with results until the last page is reached. We treat all such pages as part of one single document by concatenating all result pages into one single page. Specifically, we will consider all the result web pages as one web page containing the search status string and N result matches, where, N may be specified in the web form query by one of the following special keywords: (1) ALL (default keyword)—all the result matches from each page; (2) FIRST(x)—the first x matches starting from the first result page; (3) FIRSTP(y)—all matches from the first y result pages. These keywords should be specified in the extraction part of the SELECT operator of DEQUEL (discussed later).

5.2. Result matches

The next step to complete our result page representation is to examine the result matches. An HTML code related to a match is often organized as tables using different web styles, fonts, co-

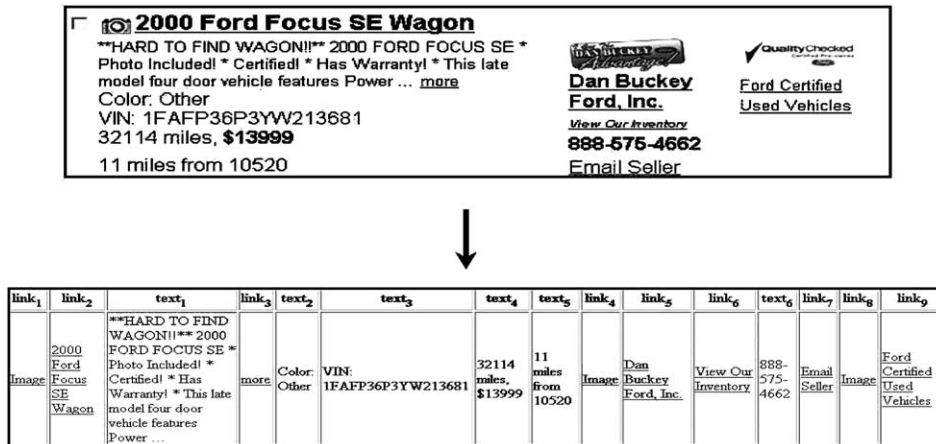| link$_1$	link$_2$	text$_1$	link$_3$	text$_2$	text$_3$	text$_4$	text$_5$	link$_4$	link$_5$	link$_6$	text$_6$	link$_7$	link$_8$	link$_9$
Image	2000 Ford Focus SE Wagon	**HARD TO FIND WAGON!!** 2000 FORD FOCUS SE * Photo Included! * Certified! * Has Warranty! * This late model four door vehicle features Power ...	more	Color: Other	VIN: 1FAFP36P3YW213681	32114 miles, $13999	11 miles from 10520	Image	Dan Buckey Ford, Inc.	View Our Inventory	888-575-4662	Email Seller	Image	Ford Certified Used Vehicles

Fig. 8. Result match representation.

lors, images and so on. As a matter of fact, only text elements and hypertext links of a result match are informative. Thereby, we ignore an HTML layout of the result match and focus on the text strings and the hyperlinks embedded in its HTML code. Thus, each result match is represented as a set of text strings and links. Links have their own internal structure similar to the structure of the HTML hyperlink, that is, the link label and the URL of the link. Note that if an HTML hyperlink label is an image, the corresponding link label is the text string defined by the `alt` attribute of the `IMAGE` tag or simply the word "*Image*". Fig. 8 shows the first result match from the `AutoTrader` result page (see Fig. 2), and the text strings and links corresponding to this match (such strings and links may be stored in HTML or XML format [10]). In this way, each result match is considered as a single row in a table with attributes of two types: *text* and *link*. Any value corresponding to the link type attribute consist of a hyperlink label and the URL of the hyperlink. The default attribute names are $text_i$, where i corresponds to the number of occurrence of the text element in the HTML code related to the result match, and $link_j$ where j is the number of occurrence of the hyperlink in the code.

Since result matches even from the same result page may have different structure (in particular, different number of text strings or links), the representation of several matches in one table is ambiguous. For example, the second result match from the `AutoTrader` result page (see Fig. 2) has five text elements and five hyperlinks, and hence, has ten attributes in the table representation. However, it is easy to see in Fig. 8 that there are 15 attributes (six text and nine link attributes) for the first match. Actually, the search for the common attributes for all result matches extracted from the result pages is a complicated problem. We give a brief description of our approach to this problem in Section 5.4.

For the time being, we assume that it is possible to build a *Result Table* (RT) for several result matches. Fig. 9 depicts a partial view of the result table corresponding to the result matches from all `AutoTrader` result pages.

---

[10] Currently, we implement the *result database* capable to store an HTML code related to a result match. Thus, text strings and links can be easily extracted from the code.

link$_1$	link$_2$	text$_1$	link$_3$	text$_2$	text$_3$	text$_4$	text$_5$	link$_4$	link$_5$	link$_6$	text$_6$	link$_7$	link$_8$	link$_9$
Image	2000 Ford Focus SE Wagon	**HARD TO FIND WAGON!!** 2000 FORD FOCUS SE * Photo Included! * Certified! * Has Warranty! * This late model four door vehicle features Power ...	more	Color: Other	VIN: 1FAFP36P3YW213681	32114 miles, $13999	11 miles from 10520	Image	Dan Buckey Ford, Inc.	View Our Inventory	888-575-4662	Email Seller	Image	Ford Certified Used Vehicles
	2000 Ford Focus LX Sedan			Color: Red	VIN: 1FAFP33P0YW169935	44053 miles, $12966	70 miles from 10520	Image	Sansone's Route 66 Auto Mall	Visit our website	888-239-3408	Email Seller		
Image	2000 Ford Focus SE Wagon				VIN: 1FAFP36P5YW375828	32532 miles, $13490	66 miles from 10520		Ray Price Motors			Email Seller	Image	Mercury Certified Pre-Owned Vehicles

236 rows

Fig. 9. Result table for AutoTrader result pages.

Initially, a result table is built with default attribute names: $text_i$ and $link_j$, where i and j are some indexes and $1 \leqslant i \leqslant k$, $1 \leqslant j \leqslant m$, where k and m are the numbers of text and link attributes of a result table respectively. For the AutoTrader example shown in Fig. 9, $k = 6$, $m = 9$ and the number of rows is 236. The operator DEFINE described in the next section allows the user to give the attributes of the result table more meaningful names.

5.3. DEFINE operator

The operator DEFINE with keyword ATTRIBUTE is used to define more suitable attribute names for the result table. Two types of syntax are available. If the result table has already been created (with default attribute names), then the syntax (also see Appendix A.1.1) is as follows.

```
DEFINE ATTRIBUTE <default attribute label> <new attribute name>
            FOR <form label>
```

The above statement defines a new attribute name for the specified default attribute name of the result table. Since the result table is created for the representation of the result pages generated by some server-side program, we specify the form name in the **FOR** clause. Thus, a result table corresponds to a form. Note that in the case of querying consecutive forms, the result table is defined for the last submitted form. For example, the submission of the form depicted in Fig. 1(b) produces the page shown in Fig. 2. If we denote this form as "*AutoTrader2*" then it is possible to define the column names in the result table using the DEFINE operator as follows:

DEFINE ATTRIBUTE *link$_2$* **carpage** FOR AutoTrader2;

DEFINE ATTRIBUTE *link$_5$* **dealerpage** FOR AutoTrader2;

DEFINE ATTRIBUTE *text$_2$* **color**FOR AutoTrader2.

These statements rename the specified attributes of the result table for the AutoTrader result pages. The modified result table is shown in Fig. 10.

The second type of syntax of the DEFINE operator is used to specify attribute name(s) if the result table was not generated before. In this form the DEFINE operator may be used to define the extraction conditions for result pages generated by a particular server-side program. The syntax (see Appendix A.1.2) of the operator is as follows:

```
DEFINEATTRIBUTE <type> <set of attribute names>
        CONDITION <condition on text> | <condition on label>
              FOR <form label>
```

First the operator specifies the type of attribute(s) (TEXT and LINK correspond to the text and link types respectively). Since more than one link or text may satisfy the extraction conditions, several attribute names may be specified. However, the operator requires at least one attribute name to be specified. Then, if necessary, the attribute names will be given by subindexing of the specified attribute name. The **CONDITION** clause specifies conditions on the text strings or hyperlinks respectively of each result match. The satisfied text string or hyperlink will be presented in the result table as the value of the column specified by the attribute name. Note that the syntax assumes that each web form has its own set of the result table's attributes. As the *result database* stores an HTML code of each result match, we can define data for extraction from the stored results anytime using the DEFINE operator.

The following example is also based on the results generated after submission of the Auto-Trader form (see Fig. 1(b)). Suppose that the user is issuing query on the AutoTrader forms for the first time. The user may specify the links and text elements that may be on the result pages. In particular, it is clear that some links or text elements would contain the text string "*Ford Focus*" if the user searches for "*Ford Focus*" cars using the AutoTrader web interface. Then, the DEFINE operator may be specified as follows:

link$_1$	carpage	text$_1$	link$_3$	color	text$_3$	text$_4$	text$_5$	link$_4$	dealerpage	link$_6$	text$_6$	link$_7$	link$_8$	link$_9$
Image	2000 Ford Focus SE Wagon	**HARD TO FIND WAGON!!** 2000 FORD FOCUS SE * Photo Included! * Certified! * Has Warranty! * This late model four door vehicle features Power …	more	Color: Other	VIN: 1FAFP36P3YW213681	32114 miles, $13999	11 miles from 10520	Image	Dan Buckey Ford, Inc.	View Our Inventory	888-575-4662	Email Seller	Image	Ford Certified Used Vehicles

Fig. 10. Modified result table.

```
DEFINE ATTRIBUTE LINK carpage
CONDITION (label contain "Ford Focus", url contain http://autotrader.com)
FOR Autotrader2
```

If the result table is not created for the `AutoTrader2` form, then the execution of this statement defines a table with one column called *carpage*. Then, the query (*find used* "*Ford Focus*" *cars made in 2000*) returns the results (the first result page is shown in Fig. 2) that are represented as shown in Fig. 11.

5.4. Result extraction

Currently, `DEQUE` extracts the pieces of HTML code that correspond to the result matches. The main idea of such extraction is the regularity of HTML patterns related to the result matches. We can find them by searching an HTML tree for the number of sibling sub-trees. Fig. 12 shows an example of the HTML tree. In this example, the HTML code contains twelve siblings subtrees representing tables (defined by `TABLE` tags) with identical structure. These tables correspond to the 12 result matches laid out on the result page. We additionally require that the subtrees to be extracted must contain several hyperlinks and text strings to distinguish them from the subtrees related to different navigation menus available on the page.

The result table for a set of result pages is built by `DEQUE` using the `DEFINE`-statements or conditions specified in a form query (see Section 6). If no information on the result table is provided, then a modified version of the approach in [7] is used. Pattern discovery in `RoadRunner` [7] is based on the study of similarities and dissimilarities between two HTML pages at a time; mismatches are used to identify relevant inner structures of the result matches. In our approach, we compare the first two result pages to determine a common inner structure of the result matches and then extract common matches' attributes from all result pages using the discovered pattern. However, this can be done effectively if there are at least two result pages both containing sufficient number (more than ten according to our experiments) of result matches. Otherwise, when only one result page is returned (or second result page contains only few result matches), `DEQUE` analyzes the two HTML pages that are artificially constructed based on the available result matches by separating them into two pages. The reader may refer to [7] for further details about the identification of relevant inner structures within the pages generated by the web forms.

carpage
2000 Ford Focus SE Wagon
2000 Ford Focus LX Sedan
....
....
2000 Ford Focus SE Wagon
....
....

Fig. 11. Result table based on extraction conditions.

Fig. 12. Example of HTML tree.

6. Deep web query language (DEQUEL)

In this section, we discuss the query language for the deep Web called DEQUEL. The DEQUEL is aimed at providing applications such as automated web agents searching for specific domain information, hidden Web crawlers, etc. with an expressive query interface to query the data in the deep Web. The proposed language, designed specifically to query web forms, is an expressive web query language that permits queries on topology (filling single or consecutive forms) and document structure (within result pages). However, in the context of data extraction, the DEQUEL is less expressive than up-to-date wrapping languages [10]. The complete syntax of DEQUEL is given in Appendix A.

6.1. Value assignment

The result pages are generated by a server-side program on the basis of the values submitted via web forms. In DEQUEL, a value or a set of values are assigned to a form field in the following way: *form_name.field_name = value* or *form_name.field_name = {set_of_values}*. For example, *Auto-Trader.make* = {"*Ford*", "*Toyota*"} assigns the values "*Ford*" and "*Toyota*" to the "*make*" field of the AutoTrader form. Corresponding initial field sets are assigned if values for some fields of a form are not specified in the query [11]. The value assignment for the button and hidden type fields are not necessary. Thus, these types of fields are omitted in the SELECT statement.

---

[11] Null string is assigned to an unspecified text field without predefined value.

The following type of value assignment is especially useful for the text type fields: *form_name.field_name* = {LABEL,*n*}, where *n* is a number. The keyword LABEL specifies that the domain values of some field with similar descriptive information (label) shall be assigned to the specified form field. The variable *n* specifies the number of domain values that are assigned. For instance, if our form database stores a form with the field "*zip*" for which *label(zip)* = "*ZIP code*" and *domain(zip)* = {"60601", "60602",..., "60612"} then the assignment *AutoTrader. address* = {LABEL,2} specifies that the values "60601" and "60602" shall be assigned to the "*address*" field. This is based on the assumption that the label of the "*zip*" field is semantically closest to the label of the "*address*" field (*label(address)* = "*Near ZIP Code*").

The DEQUEL also allows us to assign values from relational tables and results of previous form queries (these results must be presented as result tables). The syntax is as follows: *form_name.field_name* = {*relational_table_name.attribute_name,k*} or *form_name.field_name* = {*query_name.attribute_name,m*}, where *k* and *m* specify the number of values that should be assigned to the form field; *relational_table_name.attribute_name* defines the column of the specified relational table; and *query_name.attribute_name* specifies the column of the result table that stores the results of the form query. For example, suppose we need to send links to web pages containing information about used "*Ford Focus*" cars made in 2000 to a friend via SMS service. Suppose we use the text field named "*mes*" in the form called SendMessage for sending the SMS. Also, assume that our search using the AutoTrader web interface was stored in the result table (we call the table "*Focus2000*") with one column called "carpage" (as shown in Fig. 11). Then, we can assign two links from this stored result table to the "*mes*" field (that contains the text of SMS message) as follows: *SendMessage.mes* = {*Focus2000.carpage,2*}.

Note that web forms impose some restrictions on values. Only values pertaining to the form fielddomains are processed. The rest of the values are ignored. Thus the assignment *AutoTrader.make* = {"*Ford*", "*Microsoft*", "*DELL*"} will be transformed into *AutoTrader.make* = {"*Ford*"} as "*Microsoft*" and "*DELL*" are not valid in the domain.

6.2. DEQUEL syntax

The DEQUEL is intended to provide more convenient and efficient way to fill out web forms. The syntax of the DEQUEL is as follows:

```
<query>::=SELECT          [<number of results>]
                          <set of result table attributes>
                          [<set of assigned values>]
                          [AS <query label>]
          [ FROM          <source set> ]
          [ WHERE         <assignment set> ]
          [ CONDITION     <condition set> ]
```

The SELECT, retrieval operator of the DEQUEL, consists of four parts: *extraction*, *source specification*, *assignment*, and *condition* [24]. The first part of the SELECT statement is related to the query results returned by the DEQUEL query processor. The number of results defines the

number of result matches that should be extracted from the result pages for each submission data set defined in the assignment part. The `set of the result table attributes` can be specified if such table has been created before. Otherwise, the default attribute names *link*$_i$, *text*$_j$ may be used in case the result table has not been created. The **AS** clause specifies that the results of this query shall be stored and defines the reference to these results. Form(s) to be queried, relational table(s) used as a source of input data, the form URL(s) if form(s) is not pre-stored in the form database, and names of the stored query results must be specified after the **FROM** clause in the "`source set`". The **WHERE** clause defines the values to be assigned to the form fields (the fields must pertain to the forms specified in the **FROM** clause). Lastly, conditions on the data extracted from the result pages are specified in the **CONDITION** clause. In the current implementation, parentheses are not allowed and the priority of AND/OR in the condition set is based on the occurrence of the operators from left to right. We illustrate the syntax of the DEQUEL with some examples.

6.3. Examples of DEQUEL queries

Example 5. Suppose that we wish to find the ZIP codes of Chicago, USA. The web form at `http://zipfind.net` (called ZIPFind) is used to find the ZIP codes. Assume that this form has been stored in our form database. Then the query is formulated as follows:

SELECT ALL AS *Chicagozips*
FROM *zipfind*
WHERE *zipfind.104* = "*Chicago*"

The result database stores the query results and creates the Chicagozips reference to them. Since we do not specify the result table attributes the default attribute names are used. The results of the query are shown in Fig. 13(a).

Example 6. Suppose we wish to find the "best" [12] flights from Singapore to London on the following dates: October 28, 2002; November 14, 2002; and January 24, 2003 with available seats in business or economy classes. The Amadeus [13] web form is used for this query. To retrieve the relevant results we formulate the following query:

SELECT FIRST(3) *flight, depart, arrive, stops_aircraft, duration, business_seat, economy_seat, amadeus.D_Month, amadeus.D_Day* AS *Seatsaval*
FROM *amadeus*
WHERE *amadeus.D_City* = "*Singapore*" AND *amadeus.A_City* = "*London*" AND (*amadeus. D_Month, amadeus.D_Day*) = ("*October*", "*28*" "*November*", "*14*", "*January 2003*", "*24*")
CONDITION *business_seat* = (*text contains* "*Yes*") OR *economy_seat* = (*text equal* "*Yes*")

---

[12] The flight duration is minimum.
[13] Available at http://www.amadeus.net/home/index.htm.

text$_1$	text$_2$	text$_3$	text$_4$	text$_5$	text$_6$	text$_7$	text$_8$
1	60601	Chicago	IL	5,865	Cook	312	Central
2	60602	Chicago	IL	0	Cook	312	Central
3	60603	Chicago	IL	0	Cook	312	Central
4	60604	Chicago	IL	84	Cook	312	Central
5	60605	Chicago	IL	12,847	Cook	312	Central
6	60606	Chicago	IL	1,662	Cook	312	Central
7	60607	Chicago	IL	18,162	Cook	312	Central
8	60608	Chicago	IL	87,815	Cook	773	Central
9	60609	Chicago	IL	79,763	Cook	773	Central
10	60610	Chicago	IL	49,470	Cook	312	Central
11	60611	Chicago	IL	25,733	Cook	312	Central
12	60612	Chicago	IL	35,879	Cook	312	Central
.....	.....	.....	.....	.....	.....	.....	.....
.....	.....	.....	.....	.....	.....	.....	.....
87	60699	Chicago	IL	0	Cook	312	Central
88	60701	Chicago	IL	0	Cook	773	Central

(a)

carpage	dealerpage	color
1997 Toyota Corolla DX	Toyota on Western	Color: Black
1997 Toyota Camry LE	Clean Cars	Color: Black
1997 Toyota Avalon	Car Outlet	Color: Black
1997 Nissan 200SX SE-R	Continental Nissan	Color: Black
1997 Nissan Maxima	Dodge City of Countryside	Color: Black
1997 Nissan PICKUP King Cab XE 4x4	Woodmar Auto Sales	Color: Black
...	...	...
...	...	...
1997 Honda Civic EX Coupe	Jacobs Twin Auto Plaza	Color: Black
1997 Honda Passport 4 Door 4x4	Car Mart USA	Color: Black

(b)

Fig. 13. DEQUEL query results. (a) Chicagozips Query Results. (b) Japancars97 Query Results.

Note that the assignment (*form.field*$_1$, *form.field*$_2$) = (v_1, v_2, v_3, v_4) differs from the assignment *form.field*$_1$ = {v_1, v_3} *AND form.field*$_2$ = {v_2, v_4}. The former indicates that the value v_1 is assigned to the field *field*$_1$ only if v_2 is assigned to *field*$_2$. While the latter presumes that all possible combination of {v_1, v_3} and {v_2, v_4} may be assigned to *field*$_1$ and *field*$_2$. Thus, in our example, the `amadeus` form is not queried for dates such as *October* 14, 2002 or *November* 28, 2002 or *November* 24, 2002.

In the above query we specify using the keyword `FIRST(3)` that the first three result matches are extracted from the result page for each set submitted to the server-side program related to the `Amadeus` form. Furthermore, we specify the attribute names of the result table (suppose that the result table has been created and the table attributes have been given names such as *flight*, *depart* and so on) and the assigned values. Note that the assigned values are not part of the result table. We specify them to make the results easier to understand (in this example, to distinguish the flights on different dates). The **CONDITION** part of the query defines conditions on the values of the result table related to *business_seat* and *economy_seat* attributes. We require that the result match must have a value "*Yes*" corresponding to its *economy_seat* attribute or the same value corresponding to its *business_seat* attribute. Fig. 14 depicts the `Seatsaval` query results.

Example 7. Given a list of researchers from a graduate school related to natural sciences [14], suppose that we wish to find all works published by these researchers in 2002. Assume that the names of the researchers are stored in the relational table `researcher(id, name)` as shown in Fig. 15. The `PubMed` [15] form is used to search for published works. The query is formulated as follows.

SELECT *authors*, *work*, *published*, *pubmed.TEXT*
FROM *pubmed*, *researcher*
WHERE *pubmed.db* = "*PubMed*" AND *pubmed.TEXT* = {*reseacher.name,all*}
CONDITION *published* = (*text contains* "*2002*")

---

[14] We use data available at http://www.abo.fi/isb/research_groups.html.
[15] Available at http://www.ncbi.nlm.nih.gov/entrez/query.fcgi.

flight	depart	arrive	stops_aircraft	duration	business_seat	economy_seat	amadeus.D_month	amadeus.D_Day
Singapore Airlines SQ 318	09:00	15:10	Non-stop 744	14h10min	Yes	Yes	October	28
Singapore Airlines SQ 320	12:40	18:50	Non-stop 744	14h10min	Yes	Yes	October	28
British Airways BA 016	22:40	04:55 + 1 day(s)	Non-stop 744	14h15min	Yes	Yes	October	28
British Airways BA 7371	08:05	14:05	Non-stop 747	14h00min	N/A	Yes	November	14
Qantas Airways Ltd QF 031	08:05	14:05	Non-stop 744	14h00min	Yes	Yes	November	14
Virgin Atlantic VS 7318	09:00	15:10	Non-stop 744	14h10min	N/A	Yes	November	14
British Airways BA 7371	05:00	11:00	Non-stop 747	14h00min	N/A	Yes	January 2003	24
Qantas Airways Ltd QF 031	05:00	11:00	Non-stop 744	14h00min	Yes	Yes	January 2003	24
Virgin Atlantic VS 7318	09:00	15:10	Non-stop 744	14h10min	N/A	Yes	January 2003	24

Fig. 14. `Seatsaval` query results.

researcher:

id	name
1	Coffey ET
2	Kulomaa MS
3	Lahti R
4	Lahesmaa R
5	Kilpelainen I

japancars:

Price	make
10000	Toyota
…	…
15000	Nissan
…	…
12000	Mazda
…	…

Fig. 15. Tables `researcher` and `japancars`.

According to the query, results are presented in a four-column table with attributes *authors*, *work*, *published*, *pubmed.TEXT*. Similar to the previous query, the values assigned to the *pubmed.TEXT* field specifies the values of the fourth column. These values are taken from the table `researcher`. The `all` keyword specifies that all corresponding values of the relational table are assigned to the "*TEXT*" field. We can also specify the number of values that are used as input to the "*TEXT*" field of the `pubmed` form. For example, *pubmed.TEXT* = {*reseacher. name*,2} specifies the following assignment: *pubmed.TEXT* = {"*Coffey ET*", "*Kulomaa MS*"}. Fig. 16 shows the results of the query.

Example 8. Reconsider the query in Example 1. Let the forms in Fig. 1(a) and (b) be called *Autotrader* and *Autotrader*2 respectively. Assume that the Japanese makes are stored in the relational table japancars (price,make) shown in Fig. 15. Suppose that the attributes *carpage*, *dealerpage*, and *color* are defined for the result table (see Section 5.3). Then the query is formulated as follows:

```
SELECT FIRSTP(2) carpage, dealerpage, color
AS Japancars97
FROM autotrader, autotrader2, japancars, zipfind
```

authors	work	published	pubmed.TEXT
Coffey ET, Smiciene G, Hongisto V, Cao J, Brecht S, Herdegen T, Courtney MJ.	c-Jun N-terminal protein kinase (JNK) 2/3 ...	J Neurosci. 2002 Jun 1;22(11):4335-45.	Coffey ET
Karisola P, Alenius H, Mikkola J, Kalkkinen N, Helin J, Pentikainen OT, Repo S, Reunala T, Turjanmaa K, Johnson MS, Palosuo T, Kulomaa MS.	The major conformational IgE-binding epitopes ...	J Biol Chem. 2002 Jun 21;277(25):22656-61.	Kulomaa MS
Laitinen OH, Hytonen VP, Ahlroth MK, Pentikainen OT, Gallagher C, Nordlund HR, Ovod V, Marttila AT, Porkka E, Heino S, Johnson MS, Airenne KJ, Kulomaa MS.	Chicken avidin-related proteins show altered biotin-binding and ...	Biochem J. 2002 May 1;363(Pt 3):609-17.	Kulomaa MS
Tiirola MA, Mannisto MK, Puhakka JA, Kulomaa MS.	Isolation and characterization of Novosphingobium ...	Appl Environ Microbiol. 2002 Jan;68(1):173-80.	Kulomaa MS
...	...	...	...
...	...	...	...
Karkonen A, Koutaniemi S, Mustonen M, Syrjanen K, Brunow G, Kilpelainen I, Teeri TH, Simola LK.	Lignification related enzymes in Picea abies suspension cultures.	Physiol Plant. 2002 Mar;114(3):343-353.	Kilpelainen I

Fig. 16. PubMed query results.

WHERE *zipfind.*104 = {"*Chicago*"} AND *autotrader.make* = {*japancars.make*, *all*} AND *autotrader. search_type* = "*Used Cars*" AND *autotrader.zip* = {*zipfind.rt.text$_2$*,1} AND *autotrader*2.*max_price* = "10000" AND *autotrader*2.*from_year* = "1997" AND *autotrader*2.*end_year* = "1997" CONDITION *color* = (*text contains* "*Black*")

The FIRSTP(2) keyword specifies that the result matches are extracted from the first two result pages. The zipfind form is used to find the ZIP codes of Chicago. The submission of the zipfind returns 88 ZIP codes corresponding to "*Chicago*" (see Fig. 13(a)). The expression {*zipfind.rt.text$_2$*, 1} defines that only one "*text$_2$*" value from the zipfind result table (named *zipfind.rt*) is used for providing ZIP code to the related AutoTrader form field. If the query presented in Example 5 was executed, then we can formulate this query by removing *zipfind* in the third and fourth lines of the query and changing {*zipfind.rt.text$_2$*,1} in the sixth line to {*Chicagozips.text$_2$*, 1}. The query results are shown in Fig. 13(b).

Example 9. Suppose we wish to find used Toyota black cars manufactured in 1997 within US$ 10,000 and available in Chicago. We can formulate a query similar to the previous one but it is a good idea to reuse the results of the query described in Example 8 as shown below:

SELECT *carpage,dealerpage*
FROM *Japancars97*
WHERE *Japancars97.carpage* = (*label contains* "**Toyota**")

Here we used the Japancars97 result table specified in Example 8. The results of the query are a two-column result table result(carpage, dealerpage).

6.4. DEQUEL query execution

This section describes the execution of the DEQUEL query formulated by the user through the form query UI. In the preceding sections, we introduced two operators: DEFINE and SELECT.

The DEFINE operator specifies the extraction conditions on the data from the result pages. The query processing is defined exclusively by the SELECT operator.

Firstly, the *Query Processor* of DEQUE checks whether the same query was processed before. If yes, it returns the results of the processed query and indicates the date of processing. This definitely may lead to outdated results. However, in this paper we do not focus on the issues related to outdated information here. Secondly, the specified DEQUEL query is parsed by the *Query Parser*. Currently, a query on two or more forms can be composed only if the forms to be queried are consecutive. The SELECT statement envisages that a query may contain the URL of the forms. Such forms are extracted before query evaluation.

All relevant forms and relational tables (these tables must be also pre-stored in the relational database) are retrieved from the form and the relational database by the *Storage/Retrieval Manager*. For each form field specified in the query, the *Query Evaluation Module* of DEQUE considers the form field domain, initial field set, label, and, the values indicated in the query. For example, consider querying two consecutive forms: *autotrader*(*make, address, search_type, field_1, ac_afflt, borschtid*) (see Fig. 1(a)) and *autotrader2*(*model, certified, start_year, end_year, min_price, max_price, distance, advanced, advcd_on, make, address, search_type*) (see Fig. 1(b)). Suppose 24 values as relational input are assigned to the field *make*, five values are assigned to the field *max_price*, and the initial field sets of all fields of both forms consist of one value. Then, 120 possible submission data sets will be validated. The potential submission data sets with the form field information such as field domains, labels, initial field sets for each form specified in the query are passed for the validation of the values.

For each form involving in the query, DEQUE considers the domain constraints corresponding to the form. Any potential submission data set must satisfy these domain constraints. Since we perform queries on consecutive forms, two or more groups of the domain constraints may be considered. The steps given below validate the potential submission data set.

Consider a form F with fields $f_1, \ldots, f_k$, its child form F_c with fields $f_{k+1}, \ldots, f_n$ and a potential submission data set $S = \{s_1, \ldots, s_k, s_{k+1}, \ldots, s_n\}$, where $\forall i = \overline{1, n} : s_i$ is assigned to the field f_i. A data set S can be submitted if the following is true: (1) The data set $S' = \{s_1, \ldots, s_k\}$ satisfies the domain constraints of F. (2) The data set $\{s_{k+1}, \ldots, s_n\}$ satisfies the domain constraints of the form $F_c^{(S')}$ where $F_c^{(S')}$ is the result of submission of F with submission data set S'.

The *HTTP Request Module* of DEQUE transforms the validated submission data sets into the HTTP GET or POST requests. If a query on consecutive forms is executed, only the last child form is submitted to its server-side program. Values assigned to the fields of all other consecutive forms are assigned to the hidden fields of the last child form. Indeed, the majority of descendant forms contains the hidden fields whose initial values are equal to the values assigned to the fields of the parent form. For example, the AutoTrader root form is submitted with values "*Ford*", "*10520*" and "*Used Cars*" assigned to the "*make*", "*address*" and "*search_type*" fields respectively then the child form will contain the following three hidden fields: "*make*", "*address*" and "*search_type*" with values "*FORD*", "*10520*" and "*used*" respectively.

7. Implementation

We have created a prototype system that allows the user to formulate DEQUEL-queries on the web forms that can be extracted from the web, and extract and store useful data from the result pages. The implementation was conducted on a SUN workstation working under Solaris 2.7 operational system using Perl version 5.005_2 and employing MySQL (version 3.23.49) DBMS as the data storage. We also used ActiveState Perl 5.6.1 on a Pentium III workstation under Windows 2000 OS for our experiments related to HTML parsing. A web-based graphical user interface (GUI) was implemented using CGI programs written on Perl under control of the Apache Web Server (version 1.3.22). In this section, we present the architecture of the prototype system and summarize the significant results from our experiments.

7.1. System architecture

As shown in Fig. 17, our prototype system consists of the following components: the *User Interface*, the *Web Document Loader*, the *HTML Parser*, the *Query Processor*, the *Extraction Module*, and the *Storage/Retrieval Manager*.
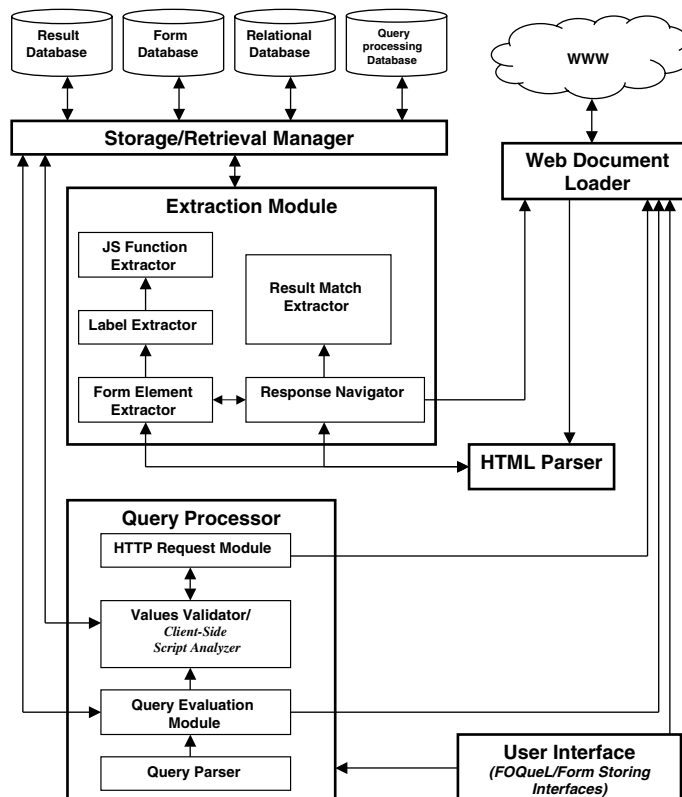

Fig. 17. Architecture of DEQUE.

The user interface shown in Fig. 6 allows the user to specify the URLs or local filenames of web pages containing one or more web forms. It calls the Web Document Loader to fetch the web page from the specified URL. The returned web page is parsed by the HTML Parser that constructs a logical tree representation of the downloaded web page, based on the document object model (DOM) [2], and passes it to the extraction module. We use the Perl collection of modules *HTML-Parser* (version 3.26) and *HTML-Tree* (version 3.11) to parse HTML documents, to create HTML syntax trees and to extract information from them.

The extraction module consists of five submodules: the Form Element Extractor, the Label Extractor, the JS Function Extractor, the *Response Navigator*, and the *Result Match Extractor*. The first three submodules (discussed in Section 4.6) are responsible for extracting HTML forms from a given web page. The form element extractor retrieves form data based on the HTML syntax related to forms. The label extractor and the JS (**J**ava**S**cript) function extractor extract additional form data (form labels and related client-side scripts). The extracted data is stored in the Form Database. The Response Navigator and the Result Match Extractor are responsible for result extraction. The response navigator retrieves all web pages linked to the returned page. If a result page contains an HTML form, then the form is extracted by the form extraction submodules. According to the DEQUEL syntax, all forms to be submitted to obtain a result page should be specified in a query. Thus, we assume that each data set submitted by the HTTP Request Module [16] generates a web page with results. However, web forms that may be contained in the result pages are extracted by the extraction module and stored in the form database.

The response navigator analyzes the HTML tree for hyperlinks whose label contains words such as "1", "2", "next" and so on. The URLs of these hyperlinks are very similar. For most cases, these URLs overlap the URL of the request. The resultant URLs are given to the Web Document Loader. The latter returns the result pages linked to the first returned page. Finally, all result pages are passed to the Result Match Extractor that extracts the result matches from the result pages and builds a result table using the technique described in Section 5.4.

The DEQUEL user interface (see Fig. 18) allows the user to specify queries on web forms using the DEQUEL. The formulated query is passed to the Query Processor (described in Section 6.4) consisting of the following components: the Query Parser, the Query Evaluation Module, the *Values Validator*, and the HTTP Request Module. The query parser parses the query and determines the steps of query execution to be given to the Query Evaluation Module. The latter is responsible for deriving different sets of input values from the *Relational Database*, *Result Database*, and the Form Database. The Result Database stores the results of form queries. The query evaluation method also calls the web document loader if some form specified in the query is not pre-stored. The goal of the Query Evaluation Module is to prepare the submission data set and pass them to the Values Validator for checking. The client-side script analyzer as a part of the validation component of the query processor allows the Values Validator to more accurately restrict the submission data set on the basis of the client-side scripts. The information about successful submission data set is stored in the *Query Processing Database* and given to the HTTP request module. This module simply submits the different form requests to the remote server-side program. The

---

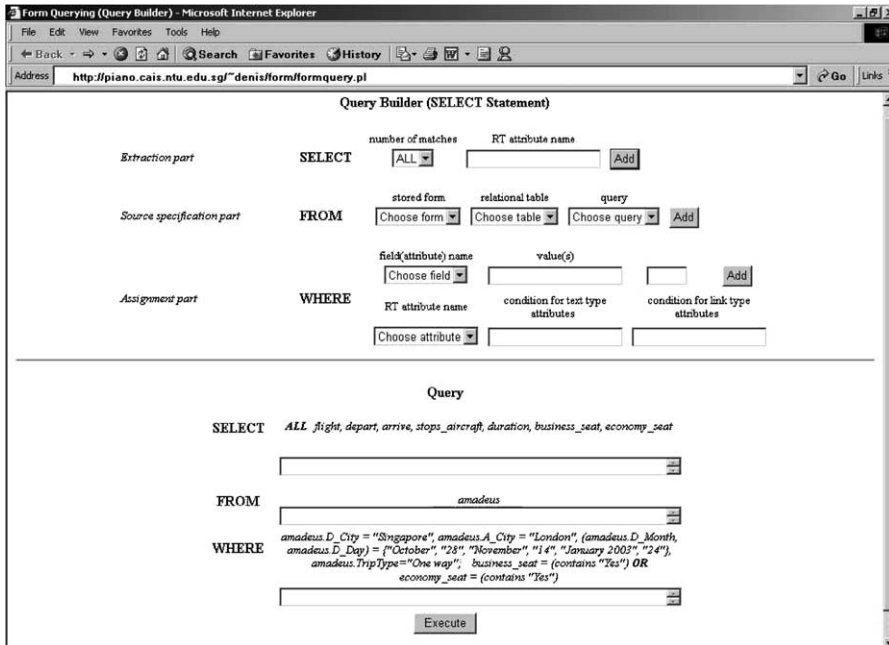[16] Except when the request is constructed for the values' validation.

Fig. 18. DEQUEL GUI.

Table 1
Form database schema

Table name	Table attributes
Form	**fid**, faction, fenctype, fmethod, rfid, pfid, rfurl, fname, fevent_id, fdesc, fdt
Field	**ffid**, form_id, ffname, fftype, session_id, ffevent_id, fflabel
Value	field_id, visval, invisval, vselected, valtext, vlength
Label	**lid**, lstr, lvalues
Event	**eid**, estr, ecode

returned web pages are parsed by the HTML Parser and forwarded to the Extraction Module described above.

7.2. Form and result storage

The prototype form and result databases are implemented using MySQL. The form database consists of five tables: *form*, *field*, *value*, *label*, and *event* (shown in Table 1). The semantics of relevant attributes is given in Table 2. Each record of the *form* table describes a form. A super form is also described by one record in the *form* table. Each record of the *field* describes a form field pertaining to a some form. If a form field belongs to a descendant form, then the information about the parent form submission (i.e., parent form identifier, submission data set, date and time of submission) is linked to the corresponding record. Each record of the *value* describes a value (visible and invisible) of a form field. A record in the *label* table describes the label and domain

Table 2
Attributes of form database

| Attribute name	Description
faction	URL of a server-side program handling a form
rfid, pfid	Identifiers of a root and a parent form correspondingly
rfurl	URL of a page containing a form (*null* for descendant forms)
fevent_id	Identifier of an event code for a form event
fdt	Date and time of the form extraction
form_id	Identifier of a form that contains this field
fftype	A type of a form field
session_id	Identifier of a session (for form fields pertaining to descendant forms)
ffevent_id	Identifier of an event code for a form field event
fflabel	A text string containing descriptive information about a field
field_id	Identifier of a form field
visval, invisval	A visible and an invisible value correspondingly
lstr	A text string containing descriptive information (that is, label itself)
ecode	A code of a JavaScript function
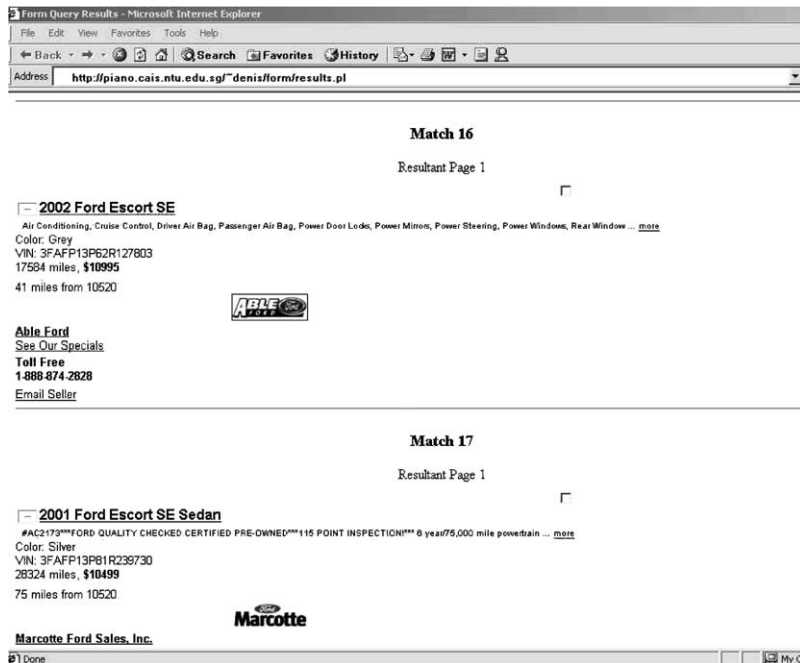
Fig. 19. Example of query results.

of values related to the label. A record in the *event* stores the JavaScript function related to a form or a form field event.

The Extraction Module stores the query results. If a result table has been built and the result table attributes have been specified in a query, then the result of a form query is represented as the

table. Otherwise, the query result is a single page containing all the result matches. Fig. 19 shows an example of the results returned by the system after performing the following query on the `AutoTrader` forms: *Find used Ford Escort cars not older than 4 years and price is within 10,520*. Currently, the extracted result matches are stored in the result database that consists of two tables: *resmatch* (describing a result match) and *rtname* (describing an attribute of some result table). Thus, more powerful extraction tool can easily analyze the results of form queries stored as pieces of HTML code (each related to one result match) to extract the data of interest.

7.3. Experimental results

We have performed queries on 29 single and 8 consecutive forms. Totally, the prototype system stored 66 different forms (that is, the *form* table of the form database contains 66 records), including 36 root forms and 16 super forms. All forms having the same submission information, location and form type are represented by one ''form'' record in the form database. Several forms are not interesting as they only sort the results returned by the other forms. Table 3 shows some of the form URL, the form type, the number of visible and invisible fields, and the number of performed queries for several forms that were used to test our technique.

Table 3
Queried forms

Form name	URL	No. of visible fields	No. of invisible fields	Type	No. of queries
AutoTrader	http://autotrader.com /findacar/index.jtmpl?ac_afflt=none	4	2	Consecutive	240
ZIPFind	http://zipfind.net	3	–	Single	40
Amadeus	http://www.amadeus.net/home/ en/home_en.htm	9	15	Consecutive	100
PubMed	http://www.ncbi.nlm.nih.gov/ PubMed/	4	4	Single	200
Google	http://google.com	3	3	Single	170
ClassicCar	http://classiccar.carfrenzy.com /autos/index2.html	2	–	Consecutive	110
Amazon	http://amazon.com	3	–	Single	30
CiteSeer	http://citeseer.nj.nec.com/cs	3	1	Single	60
Lycos Companies Online	http://business.lycos.com/ companyresearch/crtop.asp	3	–	Single	40
Powell's Books	http://www.powells.com/search /DTSearch/search.cgi	17	–	Single	20
AA Flight Search	http://www.aa.com	13	20	Consecutive	70
Phuket Hotel Guide	http://www.phuket-hotels.com /indexprices.htm	53	1	Single	10
Froogle	http://froogle.google.com/	2	–	Single	50
Yahoo!Autos	http://autos.yahoo.com/	3	5	Consecutive	100
Carsearch.com	http://www.carsearch.com/	3	1	Consecutive	100
Mobile.de	http://www.mobile.de/	40	1	Single	110
Yahoo! RealEstate	http://realestate.yahoo.com/	7	7	Single	80
SmartBargains	http://www.smartbargains.com/	3	2	Single	100

Table 4
Label extraction results

Form name	No. of labels for text fields	No. of extracted labels	No. of labels for select fields	No. of extracted labels	No. of labels for radio and checkbox fields	No. of extracted labels
AutoTrader	1	1	1	1	1	1
ZIPFind	1	1	–	–	–	–
Amadeus	2	2	4	4	–	–
PubMed	1	1	1	1	–	–
ClassicCar	–	–	1	1	–	–
Lycos Companies Online	1	0	–	–	2	0
Powell's Books	6	6	6	6	3	0
AA Flight Search	2	0	6	4	2	1
Phuket Hotel Guide	–	–	–	–	53	44
Mobile.de	2	2	17	11	20	15
Yahoo! RealEstate	1	1	4	4	1	1
Total	30	24	48	37	82	62

We have been successful with submission of 58 forms. That is, the system was able to process the issued query so that the pages containing the query results or pages containing the child forms were generated. The automatic submission of the remaining forms mainly failed due to the number of built-in JavaScript functions or HTML frames.

The advantage of our query system depends on the type (single or consecutive) of the form to be queried. The system is faster than manual filling-out if at least two data sets are submitted with the consecutive forms and more than three data sets with a single form. This is based on the following evaluation. Our system provides the web-based interface (in other words, it is also a web form) to formulate form query. To submit one data set with a single form the user needs to interact with approximately x fields. The DEQUEL interface also has similar requirements. However, submission of n data sets using a single form requires about $n*x$ interactions. With our system the user specifies n sets at about $x + n*2$ interactions, where two is an empirical number that shows how many different values are in submitted data sets. Note that $x \geqslant 2$ as for any form the user must specify at least one value and press a button. After the comparison of the number of interactions we are able to obtain the above estimation for a single form. Similarly, we believe that submission of more than two data sets with consecutive forms (say, a parent and a child form) is faster using the DEQUEL interface.

Table 4 contains the results of the label extraction for different forms. We observed that the implemented label extraction technique was able to achieve 80% and 77% accuracy in extracting the labels of the text and select fields respectively, and about 75% in the label extraction for the radio and checkboxes fields. The result pages connected by hyperlinks were correctly retrieved with 89% accuracy. For the most part, links of the "unsuccessful" result pages are images ignored by the extraction module.

The main disadvantage of the result extraction technique is that each result page is analyzed apart from other connected pages. Thus, the system is sometimes unsuccessful in extracting results from pages containing only few result matches. For example, if a query returns 53 results laid out

Table 5
Result match extraction

Form name	No. of queries	No. of query results	No. of extracted matches	Accuracy (%)	Precision (%)
AutoTrader	240	12,340	15,570	95	75
ZIPFind	40	1020	980	96	96
Amadeus	100	1200	1440	100	83
PubMed	200	1640	1390	97	82
Google	170	1700	2240	100	76
ClassicCar	110	790	610	76	98
Amazon	30	470	730	99	64
CiteSeer	60	870	710	80	98
Froogle	50	1000	1280	100	78
Yahoo!Autos	100	6920	6310	85	93
Carsearch.com	100	4350	5010	94	82
Mobile.de	110	4860	4030	79	95
Yahoo! RealEstate	80	960	880	80	87
SmartBargains	100	1540	1960	91	71

on three pages then the extraction from the third page (containing three result matches) is sometimes unsuccessful. However, the extraction module successfully extracts all matches from all connected result pages that contain more than 10–20 result matches. Additionally, for many analyzed result pages the result match extractor returns more matches than really presented on the page as some irrelevant subtrees are considered as result matches' sub-trees.

Table 5 shows the *accuracy* and *precision* of the result matches' extraction for different forms. We define accuracy as: $\text{Accuracy} = \frac{\text{Number of relevant matches}}{\text{Number of query results}} \times 100\%$, where the number of query results is the total number of actual results returned by the server-side program, and the number of relevant matches is the number of extracted matches corresponding to the actual results. The precision is given by the following ratio: $\text{Precision} = \frac{\text{Number of relevant matches}}{\text{Number of extracted matches}} \times 100\%$, where the number of extracted matches is the total number of matches extracted by the extraction module from the returned pages. For example, 80% accuracy means that system extracts only 80% of all the matches that really presented on the returned pages. Additionally, 70% precision shows that 30% of the result matches extracted by the system do not contain any useful data. In summary, our experiments show that automatic form querying is feasible, and that relatively few forms are queried incorrectly.

8. Conclusions and future work

This paper describes our work done in modeling and querying the deep Web. In particular, we have proposed a data model for representing and storing HTML forms, and a web form query language for retrieving data from the deep Web and storing them in the format convenient for additional processing. We presented a novel approach in modeling of consecutive forms and introduced the concept of the super form. The proposed web form query language DEQUEL is able

to query forms (single and consecutive) with input values from relational tables as well as from the result pages (results of querying web forms). Finally, we have implemented a prototype system based on the discussed methodologies.

In the process of DEQUE's design and implementation we have addressed several challenges in building a query system for the deep Web. First, we introduced the concept of super form to simplify the process of querying and storing consecutive forms. Secondly, we introduced the DEQUEL to provide more convenient and efficient way to fill out web forms. Furthermore, we described our approach to extraction of query results from result web pages.

As part of future work we shall focus on the following issues: (1) *Support of client-side scripts*: Currently, we only store form or field events and related functions' codes in our form database. The next step is to perform queries on web forms considering the client-side scripts. Then, we can validate values to be submitted with a form using the client-side functions. The support of built-in functions is also very desirable to identify the dependencies between the form fields (in many current web forms, the chosen value of some field triggers off the function that specifies the domain values of another field). (2) *Dependencies between consecutive forms*: The modeling of dependencies between the forms can significantly improve the performance of queries on consecutive forms as it helps to eliminate irrelevant submission data sets before submitting them to the server-side program. (3) *Intermingling of forms and results*: At present, DEQUE strictly separates forms and results. However, it is possible to have result pages which themselves contain *subforms*. We used to develop techniques to address these types of intermingled forms in the future. (4) *Transformation to XML*: We intend to investigate more robust algorithms to extract data from the result pages and storing them. In particular, we intend to extract the hidden web query results and transform them to XML using machine learning techniques. Our motivation to transform the deep Web query results to XML is the following. Deep Web data is HTML-formatted and every site generates it in its own fashion. Thus it becomes extremely difficult and cumbersome to develop generalized techniques that can be used for deep Web data integration, query processing, etc. Consequently, it is important to develop a technique for transforming deep Web data to more structured format (e.g., XML) so that we can develop such generalized techniques for the deep Web. By doing so, we can use powerful XML query languages to query the deep Web.

Appendix A. DEQUEL grammar

A.1. DEFINE operator

A.1.1. First type of syntax

```
<define> ::= DEFINE ATTRIBUTE <default attribute label> <new attribute
             name>
                        FOR <form label>
<default attribute label> ::= text<number> | link<number>
<new attribute name> ::= <value>
```

A.1.2. Second type of syntax

```
<define> : : = DEFINE ATTRIBUTE <type> <set of attribute names>
                   CONDITION <condition on text> | <condition on label>
                        FOR <form label>
<type> : : = TEXT | LINK
<set of attribute names> : : = <new attribute name> |
                             <set of attribute names>, <new attribute
                              name>
<condition on text> : : = ("text" <cond op> <value>)
<condition on label> : : = ("label" <cond op> <value>, "url" <cond op>
                          <value>) |
                          ("label" <cond op> <value>) |
                          ("url" <cond op> <value>)
<cond op> : : = "equal" | "contain"
```

A.2. SELECT operator

```
<query> : : = SELECT [<number of results>] <set of RT attributes>
                   [<set of assigned values>] [AS <query label>]
            [FROM <source set> ]
            [ WHERE <assignment set> ]
            [ CONDITION <condition set> ]
<number of results> : : = ALL | FIRST(<number>) | FIRSTP(<number>)
<set of RT attributes> : : = "*" | <RT attribute> |
                                <set of RT attributes>, <RT attribute>
<source set> : : = <source> | <source set>, <source>
<source> : : = <form source> | <alternate source> <form source> : : =
              <form label> |
<url> AS <new form name> |
              (<url>, <number>) AS <new form name>
<url> : : = <value> <new form name>: : =<value> <alternate source> : : =
<relational table label> | <query result label>
<RT attribute> : : = <attribute label> |
                   <alternate source>.<attribute label>
<assignment set> : : = <assignment clause> |
                      <assignment set> AND <assignment clause>
<assignment clause> : : = <form label>.<field label> <eq> <predicate> |
                         (<set of fields>) <eq> <predicate>
<set of assigned values> : : = <assigned value> |
                              <set of assigned values>, <assigned value>
<assigned value> : : = <form label>.<field label>
```

```
<set of fields> ::= <form label>.<field label> |
                    <set of fields>, <form label>.<field label>
<predicate> ::= <value> | {<set of values>} |
                {<alternate source>.<attribute label>,<card number>} |
                {LABEL,<card number>}
<card number> ::= <number> | <all>
<condition set> ::= <condition on attribute> |
                    <condition set> AND <condition on attribute> |
                    <condition set> OR <condition on attribute>
<condition on attribute> ::= <RT attribute> <eq> <condition on text> |
                             <RT attribute> <eq> <condition on label>
<all> ::= "all"
<eq> ::= "="
```

The intuitive meaning of the remaining nonterminals is the following: "`<relational table label>`" a name of relational table, "`<query result label>`" a reference to stored query results, "`<attribute label>`" an attribute name, "`<query label>`" a new reference to results of this query, "`<form label>`" a form name, "`<number>`" a number, "`<value>`" a text string.

References

[1] HTML 4.01 Specification—W3C Recommendation, December 1999. http://www.w3.org/TR/1999/REC-html401-19991224/.

[2] Document Object Model (DOM) Level 2 HTML Specification Version 1.0—W3C Candidate Recommendation, June 2002. http://www.w3.org/TR/DOM-Level-2-HTML/.

[3] P. Atzeni, G. Mecca, P. Merialdo. To Weave the Web, in: Proc. of the Int. Conf. on Very Large Data Bases (VLDB), 1997.

[4] R. Baumgartner, S. Flesca, G. Gottlob. Visual Web Information Extraction with Lixto, in: Proc. of the 27th VLDB Conf., Roma, Italy, 2001.

[5] S. Chakrabarti, M. van den Berg, B. Dom. Focused Crawling: A New Approach to Topic-Specific Web Resource Discovery, in: 8th World Wide Web Conf., May 1999.

[6] J. Cho, H. Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler, in: Proc. 26th Int. Conf. Very Large Data Bases, VLDB, 2000.

[7] V. Crescenzi, G. Mecca, P. Merialdo, RoadRunner: towards automatic data extraction from large web sites, VLDB J. (2001) 109–118.

[8] M. Diligenti, F. Coetzee, S. Lawrence, C.L. Giles, M. Gori. Focused Crawling using Context Graphs, in: 26th Int. Conf. on Very Large Databases, VLDB 2000, September 2000.

[9] D. Florescu, A.Y. Levy, A.O. Mendelzon, Database techniques for the World-Wide Web: a survey, SIGMOD Record 27 (3) (1998) 59–74.

[10] G. Gottlob, C. Koch. Monadic Datalog and the Expressive Power of Languages for Web Information Extraction, in: Symp. on Principles of Database Systems (PODS), 2002, pp. 17–28.

[11] H. Davulku, J. Freire, M. Kifer, I.V. Ramakrishnan. A Layered Architecture for Querying Dynamic Web Content, in: ACM Conf. on Management of Data (SIGMOD), June 1999.

[12] M.K. Bergman. The Deep Web: Surfacing Hidden Value, September 2001.http://www.brightplanet.com/deepcontent/tutorials/DeepWeb/deepwebwhitepaper.pdf.

[13] C.A. Knoblock, K. Lerman, S. Minton, I. Muslea, Accurately and reliably extracting data from the web: a machine learning approach, IEEE Data Eng. Bull. 23 (4) (2000) 33–41.
[14] D. Konopnicki, O. Shmueli. W3QS: A Query System for the World Wide Web, in: 21st Conference on Very Large Databases, Zurich, Switzerland, 1995, pp. 54–65.
[15] D. Konopnicki, O. Shmueli, Information gathering in the World-Wide Web: The W3QL query language and the W3QS System, ACM Trans. Database Systems 23 (4) (1998) 369–410.
[16] S. Lawrence, C.L. Giles, Searching the World Wide Web, Science 280 (5360) (1998) 98–100.
[17] S. Lawrence, C.L. Giles, Accessibility of information on the web, Nature 400 (1999) 107–109.
[18] D. Maier, J. Ullman, M. Vardi, On the foundations of the universal relation model, ACM Trans. Database Systems 9 (2) (1984) 283–308.
[19] A. McCallum, K. Nigam, J. Rennie, K. Seymore, Building Domain-specific Search Engines with Machine Learning Techniques, in: Proc. AAAI-99 Spring Symp. on Intell. Agents in Cyberspace, 1999.
[20] G. Mecca, P. Atzeni, A. Masci, P. Merialdo, G. Sindoni. The ARANEUS Web-base Management System, in: Proc. of the Int. Conf. on Management of Data, 1998, pp. 544–546.
[21] I. Muslea, S. Minton, C.A. Knoblock, Hierarchical wrapper induction for semistructured information sources, Autonomous Agents Multi-Agent Systems 4 (1/2) (2001) 93–114.
[22] S. Raghavan, H. Garcia-Molina. Crawling the Hidden Web. Technical Report 2000-36, Computer Science Dept., Stanford University, December 2000. Available at http://dbpubs.stanford.edu/pub/2000-36.
[23] S. Raghavan, H. Garcia-Molina. Crawling the Hidden Web, in: Proc. of the 27th Int. Conf. on Very Large Data Bases (VLDB 2001), September 2001.
[24] D. Shestakov. On Modeling And Querying Web Forms. Master's thesis, School of Computer Engineering, Nanyang Technological University (Singapore), 2002.

Denis Shestakov received his Master's degree in Computer Engineering from Nanyang Technological University, Singapore in 2002. He is currently pursuing his Ph.D. in Finland.

Sourav S. Bhowmick received his Ph.D. in computer engineering in 2001. He is currently an Assistant Professor in the School of Computer Engineering, Nanyang Technological University. His current research interests include XML data management, data integration, secured data management, web warehousing, web mining, and biological data integration. He has published more than 60 papers in major international database conferences and journals such as VLDB, IEEE ICDE, ACM CIKM, ICDCS, DEXA, IEEE Transactions on Knowledge and Data Engineering, ACM Computing Survey, and Data and Knowledge Engineering Journal. He is serving as a PC member of various database conferences and workshops and reviewer for various database journals. He is the program chair of the International Workshop on Biological Data Management (BIDM) since 2003. He is also program co-chair of 2nd International Workshop on XML Schema and Data Management (in conjunction with IEEE ICDE 2005). He is the Guest Editor of a Special Issue on Biological Data Management for the Data and Knowledge Journal. He also serve in the editorial boards of International Journal of Digital Information Management (JDIM) and International Journal of Data Warehousing and Mining (JDWM). He has co-authored a book entitled "Web Data Management: A Warehouse Approach" (Springers Verlag, October 2003). He is member of ACM and IEEE.

Ee-Peng Lim is an Associate Professor with the School of Computer Engineering, NTU, Singapore. He received his Ph.D. from the University of Minnesota, Minneapolis in 1994. He is currently the Head of Division of Information Systems at the School of Computer Engineering. He has published more than 130 referred journal and conference articles in the area of data mining, web databases, digital libraries, and database integration. His papers appeared at ACM Transactions on Information Systems (TOIS), IEEE Transactions on Knowledge and Data Engineering (TKDE), Decision Support Systems (DSS), and other major journals. He is currently an Associate Editor of the ACM Transactions on Information Systems (TOIS), and a member of the Editorial Review Board of the Journal of Database Management (JDM). He is also a Guest Editor of a Special Issue on Web Information and Data Management for the Data and Knowledge Journal. He actively participates in conference activities. He has also served as chair, co-chair and as a program committee member of numerous international conferences. He has recently been invited to join the IEEE Intelligent Transportation System Council (ITSC)'s Technical Committee on Homeland Security: Information, Communication and Transportation. Dr. Lim is a Senior Member of IEEE and a Member of ACM.