SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 9, Number 4, pp. 281–292. http://www.scpe.org

ISSN 1895-1767
© 2008 SCPE

DEEP WEB NAVIGATION BY EXAMPLE

YANG WANG AND THOMAS HORNUNG*

Abstract. Large portions of the Web are buried behind user-oriented interfaces, which can only be accessed by filling out forms. To make the therein contained information accessible to automatic processing, one of the major hurdles is to navigate to the actual result page. In this paper we present a framework for navigating these so-called Deep Web sites based on the page-keyword-action paradigm: the system fills out forms with provided input parameters and then submits the form. Afterwards it checks if it has already found a result page by looking for pre-specified keyword patterns in the current page. Based on the outcome either further actions to reach a result page are executed or the resulting URL is returned.

Key words: form analysis, deep Web navigation by page-keyword-actions

1. Introduction. The Web can be classified into two categories with respect to access patterns: the Surface Web and the Deep Web [7]. The Surface Web consists of static and publicly available Web pages, which contain links to other pages and can be represented as a directed graph. This Web graph can be traversed by crawlers (also known as spiders) and the found pages are then traditionally indexed by search engines.

The Deep Web in contrast consists of dynamically generated result pages of numerous databases, which can be queried via a Web form. These pages cannot be reached by following links from other pages and it is therefore challenging to index their content. Figure 1 depicts the general interaction pattern between the user and a Deep Web site. The user fills out the form field with the desired information (1) and the Web form is sent to the server where it is transformed in a database query. In this phase it is possible, that the system needs further user input due to ambiguity in the underlying data, e.g. there might be too many results for a query, and the user has to provide further information on intermediate pages (2). Finally, the Web server has gathered all necessary information to generate the result page and it is delivered to the user (3).

[12] discovered an exponential growth and great subject diversity of these Deep Web sites. Among others they arrived at the following conclusions:

- There are approximately 43.000—96.000 Web-accessible databases,
- The Deep Web is 400—500 times larger than the Surface Web,
- 95% of the available data and information on the Deep Web is freely available.

Taking into account this vast amount of high-quality data, which is geared towards human visitors, it is not surprising that many different research questions are actively pursued in this area at the moment, e.g. vertical search engines [13].

In this paper we present *DNavigator*, a framework to automate the necessary interaction steps to obtain data from Deep Web sites. The idea is to record the user interactions from the initial Web form to the desired result page. These interactions are generalized, where two different phases can be distinguished: the filling out and submission of the frontend form (cf. (1) in Figure 1) and the actions performed on intermediate pages (cf. (2) in Figure 1). Consequently, DNavigator consists of two components: Web form analysis and Deep Web navigation modeling.

This framework has been motivated by the FireSearch project [15], which is geared towards collecting and analyzing Deep Web data at query time. The ultimate extraction and labeling of data from the result page is done with the ViPER extraction system [23]. However, as the framework has been implemented in JavaScript and Java as a Firefox plugin it could be used with minor modifications in other projects, e.g. for a domain-specific Meta Search engine, where the relevant Deep Web sources could be integrated by an interested community, as well.

The paper is structured as follows: we start with a description of the two main components of our framework, namely the analysis of form fields in Section 2 and the navigation model in Section 3. Section 4 deals with the intricacies of implementing our research prototype in the Firefox browser. In Section 5 we present an evaluation of our system and in Section 6 we discuss related work. Finally, we give an outlook on future work in Section 7 and conclude with Section 8.

*Institute of Computer Science, Albert-Ludwigs University Freiburg, Germany, {wangy,hornungt}@informatik.uni-freiburg.de

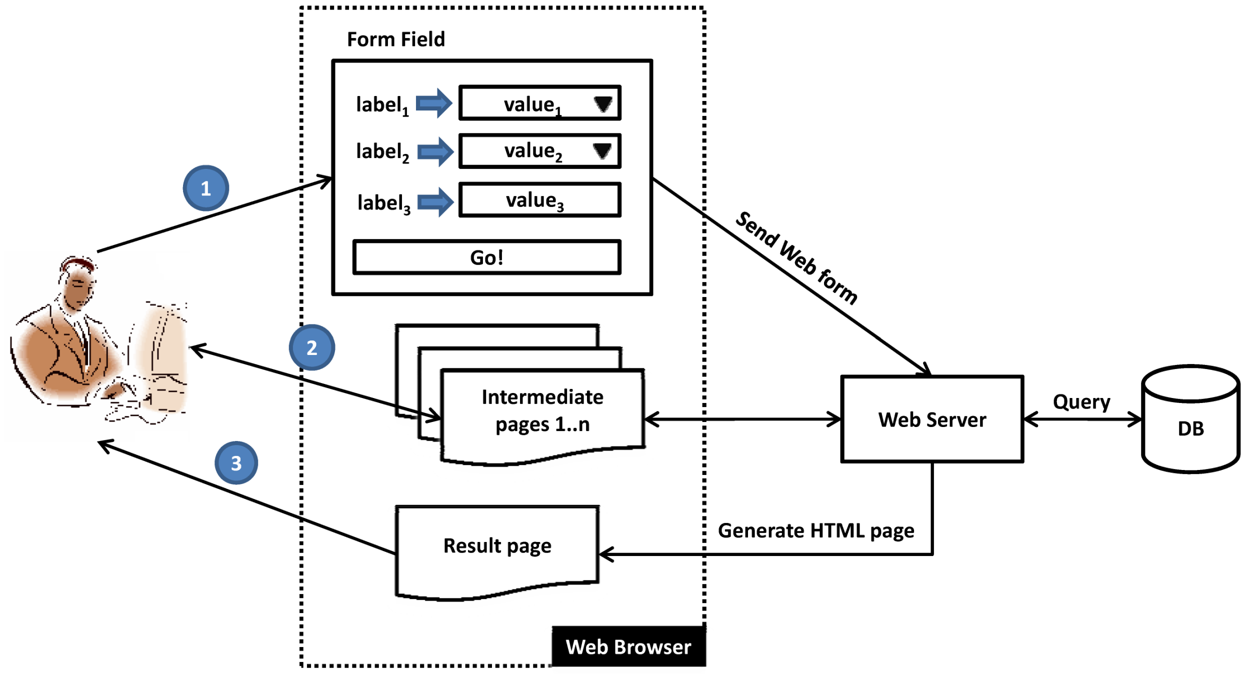
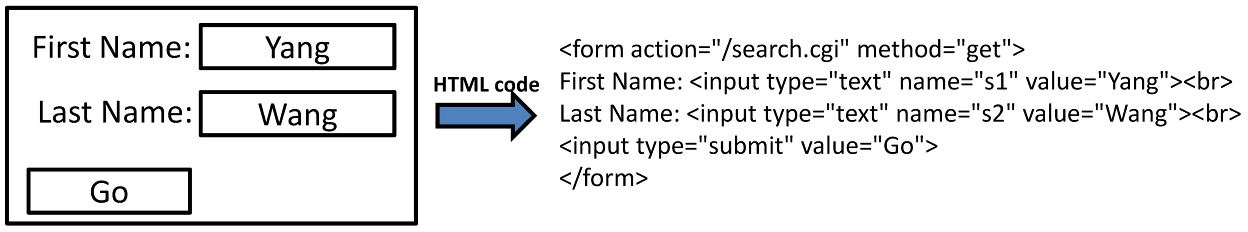


FIG. 1.1. *Accessing a Deep Web site*

2. Form analysis. Web forms are omnipresent: whether the user searches for information on Google, participates in an online vote, or comments an entry in a blog, she always provides information via filling out and submitting a form. On a more technical level, each input element (in the context of this paper we refer to all elements in the form field that can be provided with a value, e.g. checkboxes, as input elements) of a Web form is associated with a unique ID and on submission of the form the value assignments are encoded as either GET or POST HTTP request [3].



```
<form action="/search.cgi" method="get">
First Name: <input type="text" name="s1" value="Yang"><br>
Last Name: <input type="text" name="s2" value="Wang"><br>
<input type="submit" value="Go">
</form>
```

FIG. 2.1. *Web form with HTML representation*

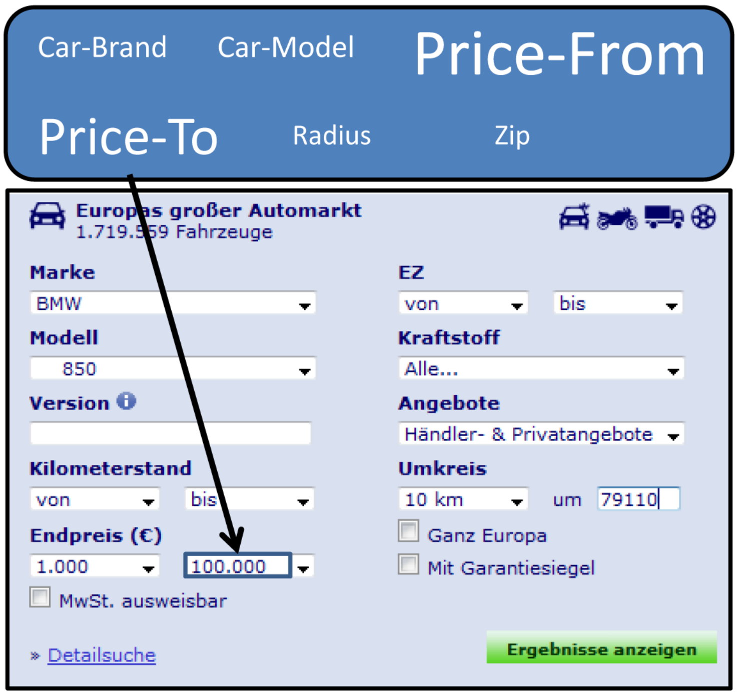
Figure 2.1 shows an example of a simple Web form. The unique ID for the input element labeled *First name* is s1 and thus the associated HTTP GET request looks as follows:

```
 GET /search.cgi?s1=Yang&s2=Wang HTTP/1.1
Host: www.example.org
User-Agent: Mozilla/4.0
Accept: image/gif, image/jpeg, */*
Connection: close
```

In Section 2.1 we discuss how to map user-defined labels onto input elements, while we deal in Section 2.2 with the problem of dependencies between different input elements. Finally, in Section 2.3 we show how to generate a valid HTTP POST/GET request based on the collected data.

2.1. Labeling of Input Elements. Initially for each new Web page we store all occurring forms with all input elements, IDs and the range of legal values (i. e. for dropdown menu lists, this would be the set of legal options), in a database for later analysis. Afterwards the user can load the desired form field and label the desired input elements, e.g. in Figure 2.2 the maximum desired price the visitor is willing to pay for a used car has been labeled Price-To. The labeling of the Web forms is inspired by the idea of social bookmarking [14]: each user has a personal, evolving vocabulary of tags. Here a tag is a combination of a string label with an XML datatype [15]. The example in Figure 2.2 shows the user vocabulary in the upper corner, where the size of the labels is determined by the frequency they have been used before.

Overall she has labeled six input elements, e.g. the desired brand and the make of the car. Now we check for each labeled input element, if they are static or if there are any dynamic dependencies, which might be due to Ajax interactions with the server. Note, that only these input elements of the form can be used later on for querying that have been labeled in this stage.



Marke: BMW	EZ: von / bis
Modell: 850	Kraftstoff: Alle...
Version:	Angebote: Händler- & Privatangebote
Kilometerstand: von / bis	Umkreis: 10 km um 79110
Endpreis (€): 1.000 / 100.000	☐ Ganz Europa
☐ MwSt. ausweisbar	☐ Mit Garantiesiegel

Fig. 2.2. *Labeling of input elements*

Our running example is the analysis of a Web search engine for used cars (`http://www.autoscout24.de`), where each car model depends on its car make. The other input elements are static, i. e. they do not change if one of the other input elements is changing.

2.2. Dependency Check of Input Elements. The dynamic and static combinations are determined automatically after the user has finished labeling the desired input elements based on the following idea: modify the first dropdown menu (only dropdown menus are currently considered as candidates for dynamic elements, all other input element types are assumed to be static by default) and check all other labeled dropdown menus, if the available options have changed. If this is the case, then modify the dependent dropdown menu to uncover layered dependencies and mark the dependent menu as dynamic. After all dropdown menus have been checked, we mark all menus that are not dynamic as static. To avoid loops, we only check possible dropdown menus that have not participated in a dependency in the current analysis cycle before, e.g. in the example shown in Figure 2.2 the car model would not be considered if we check for further dynamic dependencies for the car make input element. Figure 2.3 and Figure 2.4 show the resulting static and dynamic dependencies for our running example.

After the dependency check, the form is submitted and either a POST or GET HTTP request [3] is generated, which encodes the value assignments for the input elements. Here we store the request URL, the action attribute of the form, and the specific value assignments, which are later used for building new requests offline.

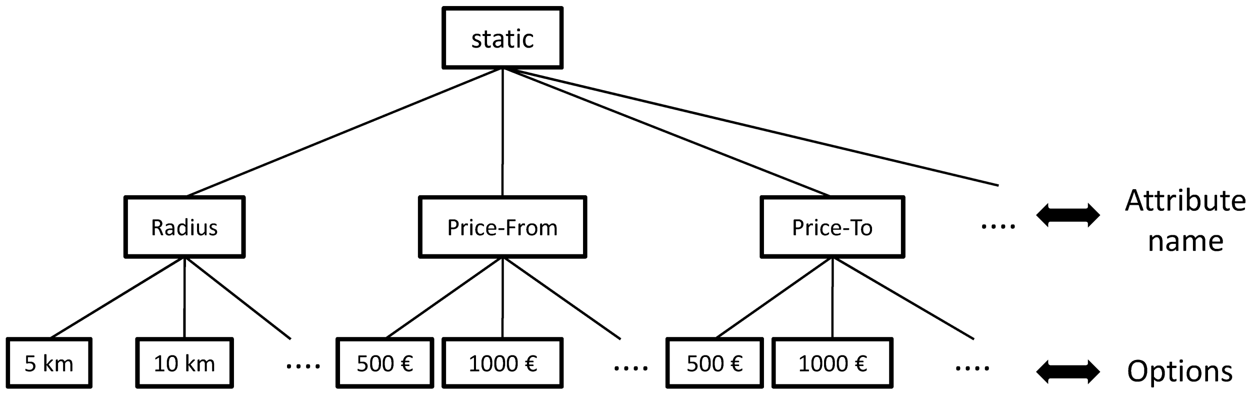


Fig. 2.3. *Relation tree for static input elements for* `http://www.autoscout24.de`

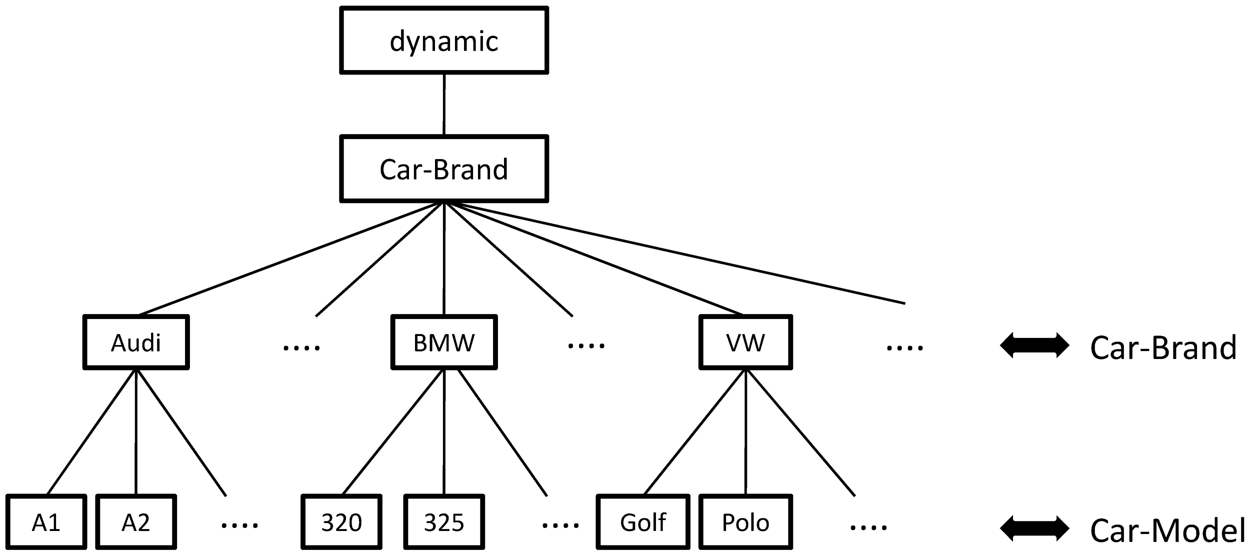


Fig. 2.4. *Relation tree for dynamic input elements for* `http://www.autoscout24.de`

2.3. Simulation of Web Form Behavior. Using the gathered data we now have two possible options to simulate the Web form behavior: we can either use the variable bindings for the user-defined tags to fill out and submit the Web form online, taking into consideration the dynamic and static dependencies or we can directly generate a POST/GET HTTP request offline. For obvious reasons, we usually prefer the offline generation, but as is discussed in Section 5.2 it is sometimes necessary to (automatically) fill out the Web form online.

Suppose the user provided the following variable bindings for our running car search example

```
Car-Brand=BMW
Car-Model=850
```

and the originally captured request URL was

```
http://www.autoscout24.de/List.aspx
```

with the following search part

```
vis=1&make=9&model=16581&...
```

Now we first match the tags to the corresponding URL field and the string representation to the associated value, yielding

```
make=13
model=1664
```

These two key-value pairs can then be inserted in the original search part, which gives us the new search part: `vis=1&make=13&model=1664&...`

Depending if a POST or GET request is required, the variable bindings are either encoded in the body of the message or directly in the URL.

After the HTTP request is send to the server, we either directly get back the result page, or alternatively an intermediate page. In the latter case we automatically navigate to the result page based on the *Page-Keyword-Action paradigm*, which is presented in the next section.

3. Deep Web navigation. The navigation model is a crucial part of our system. Based on the model the system can determine anytime, if it has already reached the result page or if it is on an intermediate page. Additionally the model determines the actions, which should be performed for a specific intermediate page, e.g. to click on a link or fill out a new form field. The key idea of our Page-Keyword-Action paradigm is that the system first determines its location (intermediate vs. result page) based on a page keyword and then invokes a series of associated actions if appropriate.

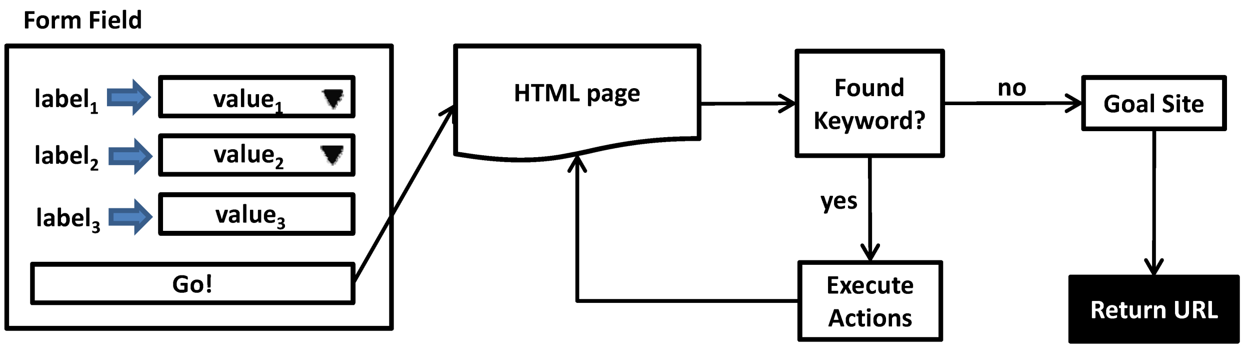
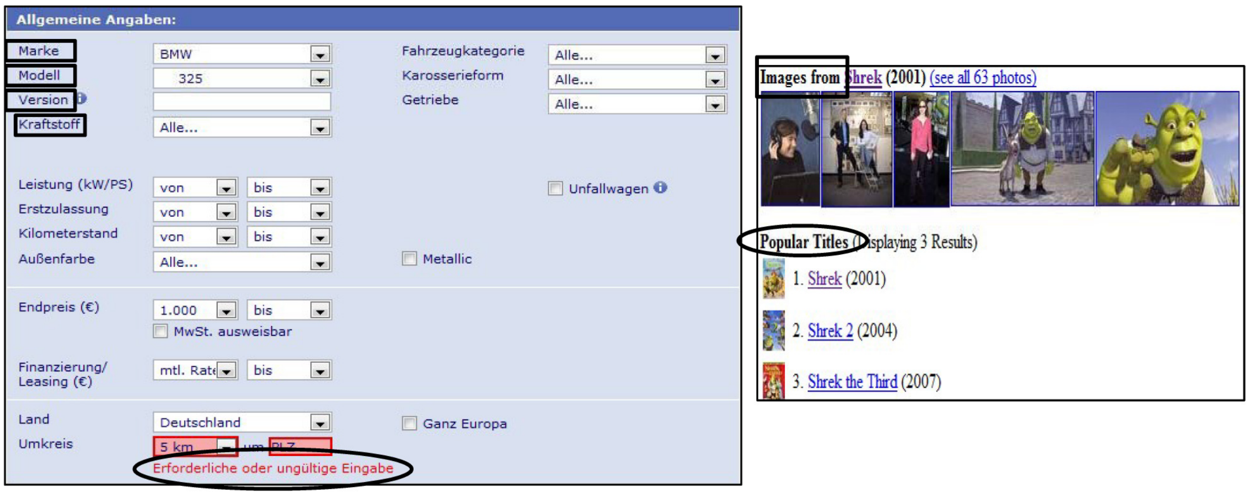


Fig. 3.1. *Navigation process*

3.1. Deep Web Navigation. The overall navigation process is illustrated in Figure 3.1: the user provides the system with a value map that contains for each desired input element label/value combinations. If the form field contains dynamic input elements for which she has provided input label/value combinations we check if they are legal. If so, we subsequently fill out and submit the form field with these combinations, which yields a new Web page (additionally, we use the information obtained during form analysis for directly generating the request POST/GET URL; thereby we can offline mimic the behavior of the form field). For this Web page we check, if we can find one of our defined keywords (cf. Section 3.2). If so, we perform the associated actions which result in a new Web page and check again if we are on a intermediate page. The cycle continues as long as we can find keywords on the Web page. To avoid an infinite loop, the user can specify an upper bound on the number of possible intermediate pages, after which an error message is returned. If we cannot find a keyword on the current Web page, we have found the goal page and return its URL.

3.2. Intermediate Page Keyword. Deep Web pages are typically created dynamically, i. e. data from a background database is filled into a predefined presentation template. Therefore, we can usually identify fixed elements, which are part of the template and are almost identical between different manifestations. After the form analysis is finished the user can iteratively submit the form with different options. If an input value combination leads her to an intermediate page, she can identify the relevant keyword as described in the following. If she has already reached a result page for a value combination no further user interactions are required. Note that as long as she is in the context of the currently active form field, she can also access a series of intermediate pages and for each page specify a series of actions. For the identification of a specific intermediate page we opted for a static text field. The reason is that it can be included in many HTML elements, e.g. the div, h2, or the span tag and given our template assumption they serve as a sufficient discriminatory factor. Other more advanced techniques based on visual markers on the page or more IR-related techniques, such as text classification approaches [19], could be used in this context as well and are planned as future work. In Figure 3.2 we have marked potential candidates for keywords with a rectangle.


Allgemeine Angaben:

Marke	BMW	Fahrzeugkategorie	Alle...
Modell	325	Karosserieform	Alle...
Version		Getriebe	Alle...
Kraftstoff	Alle...		
Leistung (kW/PS)	von bis	Unfallwagen	
Erstzulassung	von bis		
Kilometerstand	von bis		
Außenfarbe	Alle...	Metallic	
Endpreis (€)	1.000 bis		
	MwSt. ausweisbar		
Finanzierung/ Leasing (€)	mtl. Rate bis		
Land	Deutschland	Ganz Europa	
Umkreis	5 km um PLZ		
	Erforderliche oder ungültige Eingabe		

Images from Shrek (2001) (see all 63 photos)

Popular Titles (Displaying 3 Results)

1. Shrek (2001)
2. Shrek 2 (2004)
3. Shrek the Third (2007)

FIG. 3.2. *Intermediate pages for* `http://www.autoscout24.de` *(left) and* `http://www.imdb.com` *(right)*

The most likely candidates which are most characteristic are encircled with an ellipse, e.g. the error message for the car search service shown on the left. After the user has identified the keyword in the page, she can now specify actions that should be performed in order to reach the result page.

3.3. Intermediate Page Actions. The above specified keywords can be used to identify intermediate pages. However, our ultimate goal is to find a result page given a set of input value combinations for the initial form field. Therefore some actions, such as clicking on a link or filling out and submitting a new (intermediate) form, have to be performed to access the next—preferably result—page. In order to uniquely identify the appropriate HTML elements on which the stored actions should be executed, we defined a path addressing language called KApath, which is a semantic subset of XPath [5]. In order to access the appropriate action element, the system first finds the common ancestor of the keyword element and the action element and then descends downwards in the action element branch. Afterwards, the registered actions are executed for the found action element. Thus, KApath supports the following path expressions:

- **/Node[@aname$_1$=avalue$_1$=] . . . [@aname$_n$=avalue$_n$]**: The element in the DOM tree that matches the specified attribute name-value combinations of type Node,
- **/P**: Immediate parent node of current node,
- **/P::P**: All (transitive) parent nodes of current node,
- **/P::P/Node[@aname$_1$=avalue$_1$] . . . [@aname$_n$=avalue$_n$]**: The first found parent node in the DOM tree that matches the specified attribute name-value combinations starting from the current node and is of type Node,
- **/Child**: Immediate child nodes of current node,
- **/Child::Child**: All (transitive) child nodes of the current node,
- **/Child::Child/Node[@aname$_1$=avalue$_1$] . . . [@aname$_n$=avalue$_n$]**: The first found child node in the DOM tree that matches the specified attribute name-value combinations starting from the current node and is of type Node.

Figure 3.3 shows an example how the associated action element in a page can be referenced with respect to the page keyword with a KApath expression. Here, the TBODY node is the first common parent node for both (keyword and action) elements. Therefore the system automatically generates a KApath expression which allows optional intermediate elements between the keyword and the first common parent node. For finding the correct action element it is crucial to consider its attributes as well.

If the desired action elements have no (e.g. links) or dynamic attributes (e.g. visibility), we additionally store the absolute path from keyword to action element and the tree structure starting from the common parent. Another situation where we can make use of the absolute path is when the HTML page structure has changed and the common parent node is still on the same level in the DOM tree but in another branch. The tree structure is helpful if there are changes on the way downwards from the common parent node.

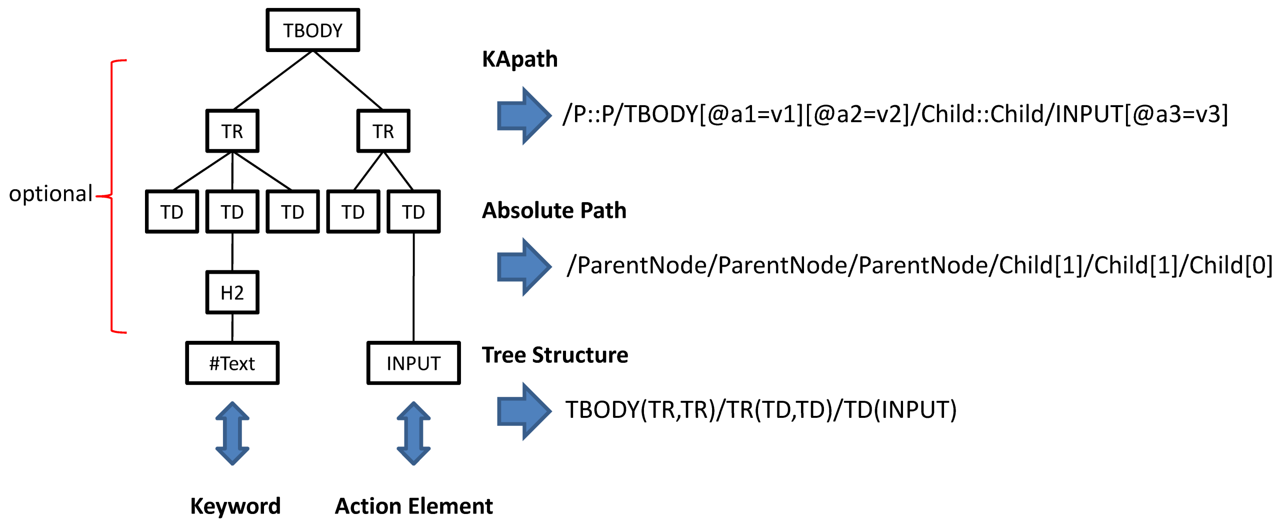

optional
TBODY
TR
TR
TD
TD
TD
TD
TD
H2
#Text
INPUT
Keyword
Action Element
KApath
/P::P/TBODY[@a1=v1][@a2=v2]/Child::Child/INPUT[@a3=v3]
Absolute Path
/ParentNode/ParentNode/ParentNode/Child[1]/Child[1]/Child[0]
Tree Structure
TBODY(TR,TR)/TR(TD,TD)/TD(INPUT)

FIG. 3.3. *Example KAPath expression, which allows optional HTML elements in the intermediate page*

3.4. Recording User Actions. Based on the user's browsing behavior, the system can generate the complete navigation model. First, she identifies the keyword for an intermediate page by clicking on the relevant text in the Web page. Then, the system determines the closest surrounding HTML element and stores the relevant context information. Afterwards, the system monitors the user behavior and stores each action she performs until she reaches a new page. Based on this action log, the system can automatically determine the paths and tree structures for each action.

To ease the recording of the user actions we have implemented *WScript*, a HTML action language similar to Chickenfoot [8]. This intermediate script language is convenient, because in order to find the HTML elements on which the actions have to be invoked we have to rely on the navigation structures defined in Section 3.3. Therefore, the provided actions have a navigation and (if applicable) an input part.

The following types of actions are supported by our system:

- Clicking on links: *link(absolute path, KApath, tree structure)*,
- Entering text in input fields: *enter(absolute path, KApath, tree structure, element name, element ID, input value)*,
- Selecting a checkbox or radio button: *click(absolute path, KApath, tree structure, element name, element ID)*,
- Selecting an option from a dropdown menu: *dropdown(absolute path, KApath, tree structure, option text, element name, element ID)*, and
- Submitting forms: *click(absolute path, KApath, tree structure, element name, element ID)*.

The element name and ID that are present for some actions are identical to the name and ID attributes of the underlying HTML element and are used first to find the relevant HTML element. If the lookup by ID and name fails, the search for the action element continues with the KApath as described in Sections 3.3 and 4.2. For example the following action expression would enter *Hallo World* into the text input field of the HTML tree in Figure 3.3:

```
enter(/ParentNode/ParentNode/ParentNode/Child[1]/Child[1]/Child[0]
,/P::P/TBODY[@a1=v1][@a2=v2]/Child::Child/INPUT[@a3=v3],
TBODY(TR,TR)/TR(TD,TD)/TD(INPUT),,,Hallo World)
```

Together, keyword and the associated actions form the navigation model for this intermediate page (cf. Figure 3.3).

4. Implementation. In this section, we describe in detail the implementation of DNavigator. Because the framework is geared towards casual Web users, important requirements must be met, most notably the tool must be easy to use. The DNavigator functionality is implemented as a Firefox extension in Java and JavaScript running a MySQL database for storing the necessary metadata (cf. Figure 4.1). LiveConnect [11] provides JavaScript with the ability to create and manipulate standard Java objects so that the system can connect to the database, e.g. to store the extracted dynamic dependencies and the navigation model, and fetch the predefined navigation models from the database to manipulate an intermediate page.

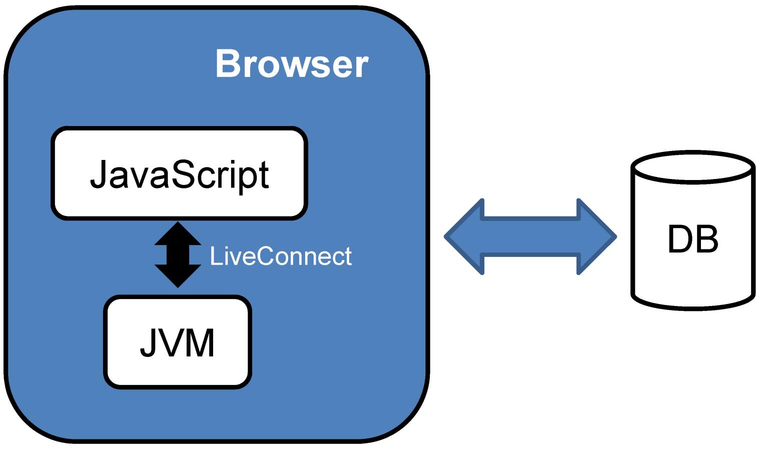


Fig. 4.1. *System architecture*

The rest of this section describes the implementation as well as the main issues we solved while implementing the system.

4.1. Navigation Model Creation. The system tracks a user's navigation actions on an intermediate page by adding JavaScript event handlers to Web pages before recording. These event handlers are invoked when certain user actions occur (e.g., clicking on text, clicking on a link, changing the selected option in a dropdown menu, etc.), which are supported by our system. The recording process is as follows: when the user presses the analysis button in the Firefox plugin window, the system sets event handlers on all clickable elements in the page displayed in the browser (i. e., handlers for links, handlers for forms, etc.). When an event fires, the system records all the necessary information for the event, e.g. KApath, absolute path, tree structure etc. It must then wait until the following page is loaded to repeat the process of adding handlers and waiting for events.

In order to determine the KApath, absolute path and tree structure with respect to the keyword and action element the system traverses the Document Object Model [1] tree starting from both elements.

4.2. Deep Web Navigation. After submitting the offline generated HTTP request, the first web page is returned from the target server. The system inserts an *onload* handler in the Web page to detect when the page has been completely loaded. Then, after the page has been downloaded, the navigation is invoked, i. e. the system will check whether one of our predefined keyword elements exist in this returned page. If this is the case, it is an intermediate page. Because for each keyword element we have saved its HTML type, attributes sequence and contained text, the check process was realized in JavaScript using the document object and node object based on the DOM tree, i. e. with the method *getElementsByTagName()*. The system first finds all HTML elements that have identical HTML types as keyword element. After the comparison between the attributes of these found elements and the stored attributes of the keyword element, and between the saved keyword text and the found keyword text, the system can determine whether a saved keyword (keyword element) exists in this intermediate page.

For any intermediate page a number of related actions must be performed, so that the system is able to navigate in the direction of the result page. Before such actions are executed, the system must first find the action-related elements, i. e. we must find all HTML elements, on which the action-events have to be activated. For this goal we use WScript that was presented in Section 3.4. The associated script will be fetched from the database using the Website ID and the identified keywords. Afterwards, an interpreter function is invoked to parse and execute every WScript expression step by step. Here, we iteratively use the following approaches:

1. When the corresponding element's attributes *id* or *name* are available, the action element can be easily found with the method *getElementById()* and *getElementsByName()*.

2. Otherwise, we try to find the action element based on the KApath-expression.
3. Finally, if the action element after executing the first two strategies cannot be found, the system uses the absolute path and the tree structure to locate the action element.

The execution of related actions is simulated using DOM Level 2 events [1, 2], i. e. fake event objects are created using the *document.createEvent()* method. Afterwards, they are activated on the desired action element using the *element.dispatchEvent()* method.

5. Evaluation. In our experiments, we evaluated the following aspects for our two major components: accuracy and runtime. For this, we selected 100 Deep Web sites from different domains, e.g. car search and video search. 60 of them were directly adopted from the website table in [7], because they contain a large amount of data. The others were selected by a focused search on Google on Deep Web repositories. For a full list of the tested Web sites we refer the interested reader to [25].

5.1. Experimental Results. All experiments have been conducted on a Thinkpad T60 (Intel Core Duo 2 Processor T7200 2,00Ghz with a 667MHz front side bus and 2GB of main memory) running Windows Vista, MySQL Server 5.0, Java JDK 1.6 and Firefox 2.0.0.12. The maximal download rate of the internet connection was 2048 Kbit/s and the maximum upload rate 256 Kbit/s.

5.1.1. Frontend Analysis. For 99% of the tested Web sites the frontend analysis was successful, finding the correct static and dynamic dependencies. Depending on the number of items in the dropdown menus of the form fields, the time needed for analysis took from 0.5 to 30 seconds, i. e. 4.28 seconds on average. Since this analysis has only to be performed once, we feel that performance optimizations for this analysis are of limited benefit, because our major focus is on correctly identifying hidden dependencies between the dropdown menus.

Table 5.1
Time (in seconds) for navigation experiments.

# Int. Pages	# Web Sites	Page Load	1 Model	6 Models
0	58	2.25	2.26	2.31
1	22	-	4.60	4.66
2	14	-	6.47	6.55
3	4	-	8.12	8.23
4	1	-	9.70	9.83
5	1	-	11.06	11.22

5.1.2. Deep Web Navigation. For 96% of the tested Web sites we were able to successfully find a keyword and to navigate to the desired result page. The navigation process took from 2.26 to 11.22 seconds, i. e. 3.79 seconds on average. As shown in Table 5.1 most of the time was spend for loading pages, i. e. 2.25 seconds on average. The columns labeled *1 Model* and *6 Models* indicate the number of registered navigation models for each page. As can be seen, the overhead for checking multiple models was marginal in contrast to the time spent for loading pages. This is due to the fact that the execution of the actions is performed by the browser on the client side and since no computationally intensive algorithm is required to identify intermediate pages.

5.2. Open Issues. Our evaluation revealed the following open issues of our system.

5.2.1. Frontend Analysis.

- Delayed AJAX interactions: For one Web site we were unable to correctly detect the dynamic dependencies because the server took longer than our specified threshold to change the items in the respective dropdown menu.

This could be remedied by increasing our threshold value to some extent, but further investigation is needed to find a general solution for this problem.

5.2.2. Deep Web Navigation.

- Dynamic request URLs: Usually, different request URLs only differ in the searchpart, i.e. the part of the URL after ?, due to different variable bindings, which are transferred to the server. Two Web sites in our test bed used different paths as well, which our system converts into illegal request URLs.

- Hidden form elements: Since the user can only drag labels to visible form elements, values in hidden form elements that have to be correlated with visible elements cannot be detected by our system.
- Session IDs: Session IDs are often used to track user interactions with Web pages and are only valid for a certain period. Because we are not able to produce a new (fake) session ID for each service, the offline generated URL becomes invalid over time.

All of the abovementioned issues could be solved by filling out the frontend form at runtime and skipping the offline generation of the URL for such resources.

- Static URLs: Our system determines, if a new Web page has been loaded based on the current URL. If the URL does not change after a form has been submitted, we are not able to initiate the navigation process and add the required event handler as described in Section 4.2.

This can be solved by using another metric for determining if a new Web page has been loaded, e.g. a checksum of the Web page.

6. Related Work. [22] presents a framework called DEQUE for querying Web forms where input values are allowed from relations as well as from result pages. As a part of their system they also model Web form interfaces, but their focus is more on the modeling of consecutive forms and they did not consider the dependencies between form input elements.

A number of navigation concepts have been proposed for accessing Deep Web sources. [10] and [18] proposed process-oriented navigation maps, which describe a set of paths from a start page to a result page. But these maps rely on consecutive state transitions and fixed interactions between them. In [16] the user actions from a specified start page over possibly multiple intermediate pages to an end page are recorded in a navigation map. The actions that link two adjacent pages are strongly connected as well. A sophisticated Deep Web navigation strategy based on the branched navigation model is proposed in [6]. The navigation is represented as a sequence of pages, with envisioned future support for standard process-flow languages such as WS-BPEL [4]. In [21] a navigation sequence was specified in NESQL [20]. The NESQL expression contains metadata about action elements, for instance, their specified names and types. Each expression will be interpreted based on these element properties. By storing historical information from previous accesses of a Deep Web resource and utilizing browser pools, their system tries to reuse the current state of a browser. [24] describe a system called WebVCR, which is able to record and replay a series of browser steps as a smart bookmark, but they do not consider optional intermediate pages.

Our framework is not dependent on a rigid sequence of intermediate pages, because for each new page all keyword patterns are checked and therefore the previous state of the system is not important for our page-oriented navigation model. Besides, we do not need a complex navigation algebra or calculus for the navigation process because we just save the above described navigation model for each intermediate page. For instance, the framework proposed by [10] relies on a subset of serial-Horn Transaction F-Logic [17]. As discussed in Section 3.4, the saved action sequences are just macro procedures, which are interpreted by our JavaScript macro engine.

7. Future Work. At the moment we only perform a hard string match between user inputs and the options in a dropdown menu. If the strings do not match exactly an error is returned. At the moment we are investigating approximate string matching techniques [9] to alleviate this problem to some extent. An alternative would be to use semantic similarity metrics, such as proposed in [27], which would also be able to capture the similarity between the two car companies *Toyota* and *Lexus* (a division of Toyota). The work by [26] tries to automate the extraction of query capabilities, such as labeling form input elements and finding legal ranges of input values. This could be interesting to combine with our approach to suggest tags to the user, or to try to match the labels on the Web form with the tags in the user vocabulary and thus easing the labeling of the Web forms.

Our experiments suggest that the determination of a suitable keyword is crucial for the successful identification of an intermediate page, and that for some cases it might be better to skip the offline generation of the start URL. Currently, we are extending our research prototype to accept a list of keywords and work on an algorithm to automatically suggest meaningful and discriminatory keywords. Ultimately, we are interested in generalizing the concept of immediate page identification to more elaborate techniques, such as the visual appearance of the Web page.

8. Conclusion. In this paper we presented DNavigator, a framework for accessing result pages of Deep Web sites, which contributions are twofold: first, a frontend analysis has been described, which needs only to be performed once, and afterwards the system can simulate the behavior of the Web form offline. Second, we have proposed a simple but effective Deep Web navigation strategy, which replaces a heavy-weight navigation calculus with an intermediate page identification procedure and a set of actions that navigate to the next page. The proposed navigation strategy has the following benefits:

1. *It is stateless.* Because for each page, we check all available navigation models, we are not dependent on a specific navigation order.
2. *Simple extensibility.* If the system encounters a new and so far unknown immediate page, the user can easily extend the existing navigation model with only a few steps.
3. *Simple presentation of the model.* Each navigation model has an intuitive textual representation which is easier to understand and use than a complicated navigation calculus.

To sum up, DNavigator offers a simple user interface, but successfully deals with most of the problems that are posed by real-world Deep Web sites as our evaluation has shown.

REFERENCES

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

[2] *Document object model (dom) level 2 events specification.* `http://www.w3.org/TR/DOM-Level-2-Events/`

[3] *Hypertext transfer protocol—http/1.1 (rfc 2616).* `http://tools.ietf.org/html/rfc2616/`

[4] *Web services business process execution language version 2.0.* `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`

[5] *Xml path language (xpath) version 1.0.* `http://www.w3.org/TR/xpath`

[6] R. Baumgartner, M. Ceresna, and G. Ledermüller, *Deep web navigation in web data extraction*, in Proceedings of the 2005 International Conference on Computational Intelligence for Modelling, Control and Automation, and International Conference on Intelligent Agents, Web Technologies and Internet Commerce., Vienna, Austria, 2005, pp. 698–703.

[7] M. K. Bergman, *The deep web: Surfacing hidden value, white paper.* `http://www.brightplanet.com/images/stories/pdf/deepwebwhitepaper.pdf` 2001.

[8] M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller, *Automation and customization of rendered web pages*, in Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology, Seattle, WA, USA, 2005, pp. 163–172.

[9] W. W. Cohen, P. Ravikumar, and S. E. Fienberg, *A comparison of string distance metrics for name-matching tasks*, in Proceedings of the Workshop on Information Integration on the Web, Acapulco, Mexico, pp. 73–78.

[10] H. Davulcu, J. Freire, M. Kifer, and I. V. Ramakrishnan, *A layered architecture for querying dynamic web content*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, Philadelphia, Pennsylvania, USA, 19999, pp. 491–502.

[11] D. Flanagan, *JavaScript: The Definitive Guide, Fourth Edition*, OŠReilly, Sebastopol, CA, USA, 2001.

[12] B. He, M. Patel, Z. Zhang, and K. C. C. Chang, *Accessing the deep web*, Communications of the ACM, 50 (2007), pp. 94–101.

[13] H. He, W. Meng, C. T. Yu, and Z. Wu, *Wise-integrator: A system for extracting and integrating complex web search interfaces of the deep web*, in Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, 2005, pp. 1314–1317.

[14] P. Heymann, G. Koutrika, and H. Garcia-Molina, *Can social bookmarking improve web search?*, in Proceedings of the International Conference on Web Search and Web Data Mining, Palo Alto, California, USA, 2008, pp. 195–206.

[15] T. Hornung, K. Simon, and G. Lausen, *Mashing up the deep web—research in progress*, in Proceedings of the 4th International Conference on Web Information Systems and Technologies, Funchal, Madeira—Portugal, 2008, pp. 58–66.

[16] N. Julasana, A. Khandelwal, A. Lolage, P. Singh, P. Vasudevan, H. Davulcu, and I. V. Ramakrishnan, *Winagent: A system for creating and executing personal information assistants using a web browser*, in Proceedings of the 2004 International Conference on Intelligent User Interfaces, Funchal, Madeira, Portugal, 2004, pp. 356–357.

[17] M. Kifer, *Deductive and object-oriented data languages: A quest for integration*, in Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases, Singapore, 1995, pp. 187–212.

[18] J. P. Lage, A. S. da Silva, P. B. Golgher, and A. H. F. Laender, *Collecting hidden web pages for data extraction*, in Proceedings of the 4th ACM CIKM International Workshop on Web Information and Data Management, Virginia, USA, 2002, pp. 69–75.

[19] K. Nigam, A. McCallum, S. Thrun, and T. M. Mitchell, *Learning to classify text from labeled and unlabeled documents*, in Proceedings of the 15th National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, Wisconsin, USA, 1998, pp. 792–799.

[20] A. Pan, J. Raposo, M. Alvarez, J. Hidalgo, and A. Vinaet, *Semi-automatic wrapper generation for commercial web sources*, in Proceedings of the Working Conference on Engineering information Systems in the Internet Context, Kanazawa, Japan, 2002, pp. 265–283.

[21] J. Raposo, M. Alvarez, J. Losada, and A. Pan, *Maintaining web navigation flows for wrappers*, in Proceedings of the 2nd International Workshop on Data Engineering Issues in E-Commerce and Services, San Francisco, CA, USA, 2006, pp. 100–114.

[22] D. Shestakov, S. S. Bhowmick, and E. P. Lim, *Deque: Querying the deep web*, Data & Knowledge Engineering, 52 (2005), pp. 273–311.

[23] K. Simon and G. Lausen, *Viper: Augmenting automatic information extraction with visual perception*, in Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, 2005, pp. 381–388.

[24] A. Vinod, J. Freire, B. Kumar, and D. F. Lieuwenet, *Automating web navigation with the webvcr*, in Proceedings of the 9th International World Wide Web Conference, Amsterdam, The Netherlands, 2000, pp. 503–517.

[25] Y. Wang, *Deep web navigation by example*, master's thesis, Institute of Computer Science, Albert-Ludwigs University Freiburg, 2008.

[26] Z. Zhang, B. He, and K. C. C. Chang, *Understanding web query interfaces: Best-effort parsing with hidden syntax*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, 2004, pp. 107–118.

[27] C. N. Ziegler, K. Simon, and G. Lausen, *Automatic computation of semantic proximity using taxonomic knowledge*, in Proceedings of the ACM CIKM International Conference on Information and Knowledge Management, Arlington, Virginia, USA, 2006, pp. 465–474.

Edited by: Dominik Flejter, Tomasz Kaczmarek, Marek Kowalkiewicz
Received: January 11th, 2008
Accepted: March 19th, 2008
Extended version received: June 25th, 2008