# Deep Web Navigation by Example

Yang Wang and Thomas Hornung

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

**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

A recent study by He et. al [5] has found an exponential growth and great subject diversity of Deep Web [2] sites. Taking into account the 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 [4].
In this paper we present a framework which bridges the gap between the front page and the desired result page which actually contains the relevant data. In a user-assisted acquisition step we first analyze the relevant form fields on the Web page we are interested in and then build a navigation model based on the page-keyword-action paradigm. The main idea is twofold: first, the user has to identify and label the relevant input form fields. For these we pre-compute and store the dependencies in a database so that we can check for illegal combinations offline. Second, we use the user-provided input values at runtime to fill out the appropriate form fields and then check after submittal if we have already reached the result page. The check is based on a keyword sequence, which gives us a hint if we are on an intermediate, or bridge, page. If so, a series of actions, which are associated with this bridge page, is performed, which yields us to a new page. Here again, we check if we have reached a result page. If this is the case, we return the URL, otherwise we (again) perform the associated actions or return an error message.
This framework has been developed for use in the FireSearch project [6] whose

aim it is to organize Deep Web sources in a mashup graph, where it is used in conjunction with the ViPER [13] wrapper tool to convert Deep Web sources into machine-processable query interfaces. However, as it 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. In Section 4 we present an evaluation of our system and in Section 5 we discuss related work. Finally, we conclude in Section 6 with an outlook on future work.

## 2 Form Analysis

Initially for each new Web page we store all occurring forms in a database for later analysis. Afterwards the user can load the desired form field and label the desired input elements[1], e.g. in Figure 1 the maximum desired price the visitor is willing to pay for a used car has been labeled *Price-To*. 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. Our running example is the analysis of a Web search engine for used cars[2], 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. 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[3] 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 1 the car model would not be considered if we check for further dynamic dependencies for the car make input element.
Figure 2 and Figure 3 show the resulting static and dynamic dependencies for our running example. After the *frontend* analysis is finished, we continue with

[1] 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.
[2] `http://www.autoscout24.de`
[3] Only dropdown menus are currently considered as candidates for dynamic elements, all other input element types are assumed to be static by default.

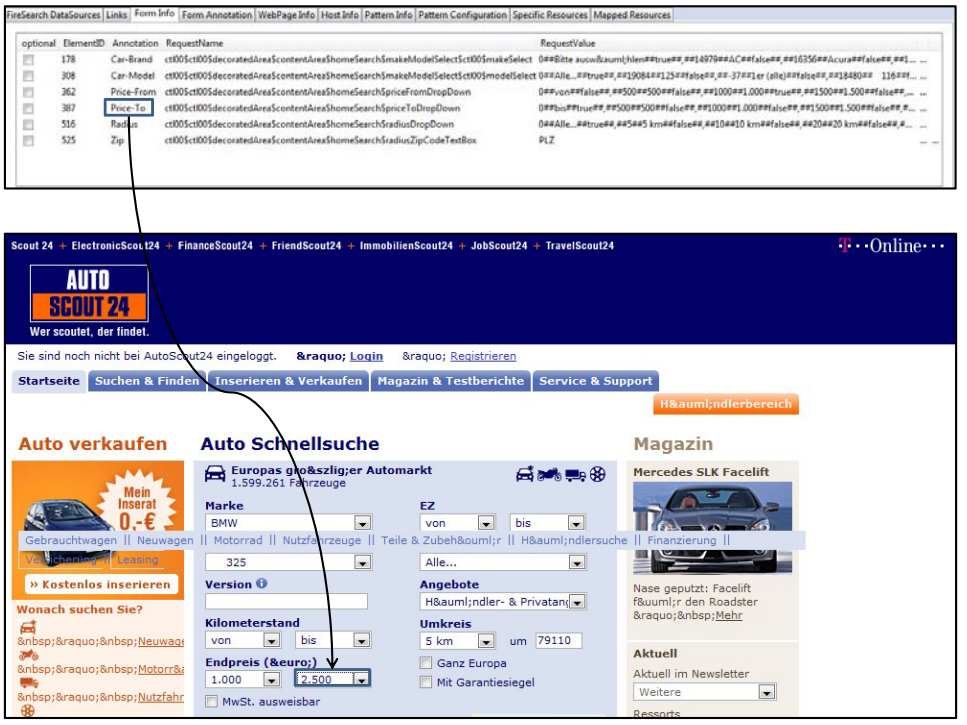


**Fig. 1.** Annotation of form attributes

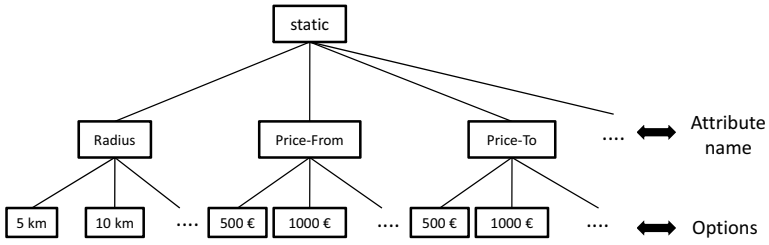


**Fig. 2.** Relation tree for static input elements for `http://www.autoscout24.de`

the analysis of the possible navigation patterns for this page described 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 anytime determine, 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 perfomed for a specific intermediate page, e.g. to click on a link or fill

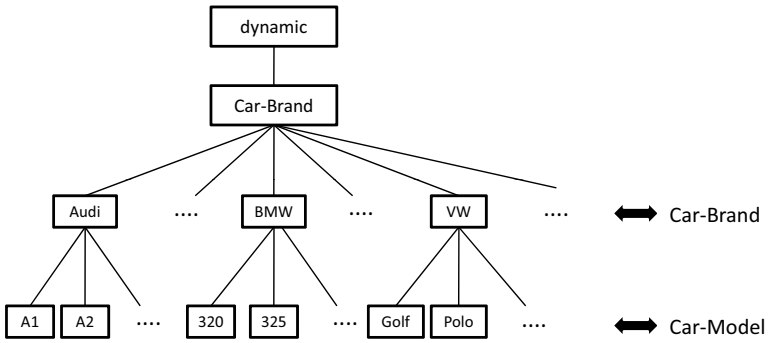


**Fig. 3.** Relation tree for dynamic input elements for `http://www.autoscout24.de`

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.

### 3.1 Deep Web Navigation

The overall navigation process is illustrated in Figure 4: 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[4]. 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, which 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

[4] 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.

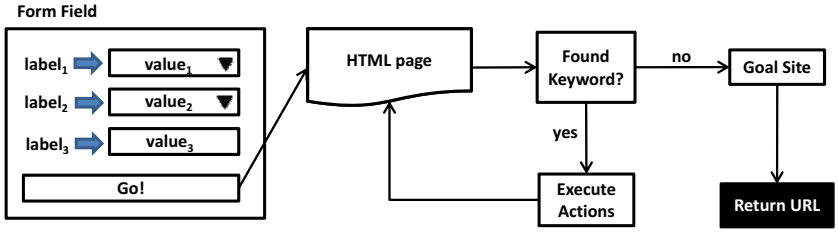


**Fig. 4.** Navigation process

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 [10], could be used in this context as well and are planned as future work. In Figure 5 we have marked potential candidates for keywords with a rectangle. 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.

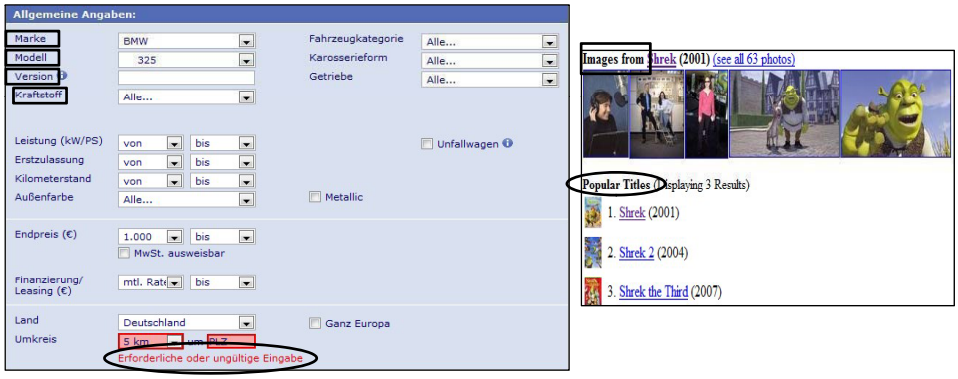


**Fig. 5.** Intermediate pages for `http://www.autoscout24.de` (left) and `http://www.imdb.com` (right)

### 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 perfomed to access the next - preferrably 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 [16]. 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*.

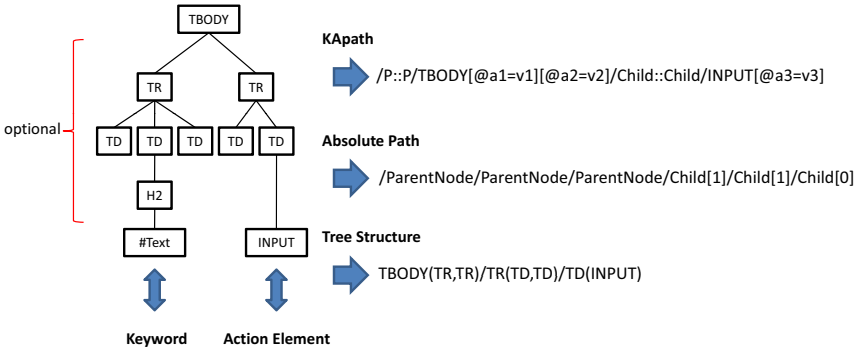


**Fig. 6.** Example KApath expression that allows optional HTML elements in the intermediate page

Figure 6 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. However, it can still happen that the desired action elements have no (e.g. links) or dynamic attributes (e.g. visibility). For these cases 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. Together, keyword, KApath, absolute path and the tree structure form the navigation model for this intermediate page (cf. Figure 6).
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.
The following type of actions are supported by our system:

- Clicking on links,
- Entering text in input fields,
- Selecting options from a dropdown or checkbox menu, and
- Submitting forms.

## 4 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 [2], 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 [14].

### 4.1 Experimental Results

**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

| # 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 |

**Table 1.** Time (in seconds) for navigation experiments

for this analysis are of limited benefit, because our major focus is on correctly identifying hidden dependencies between the dropdown menus.

**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 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.

### 4.2 Open Issues

Our evaluation revealed the following open issues of our system.

**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.

**Deep Web Navigation**

– Dynamic request URLs: Usually, different request URLs only differ in the searchpart[5], 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.

[5] The part of the URL after the ?.

– 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.

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.

## 5 Related Work

A number of navigation concepts have been proposed for accessing Deep Web sources. [3] and [9] 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 [7] 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 [1]. The navigation is represented as a sequence of pages, with envisioned future support for standard process-flow languages such as BPEL [15]. In [12] a navigation sequence was specified in NESQL [11]. The NESQL expression contains several informations 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.
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 [3] relies on a subset of serial-Horn Transaction F-Logic [8]. As discussed in Section 3.3, the saved action sequences are just macro procedures, which are interpreted by our JavaScript macro engine.

## 6 Conclusion and Future Work

In this paper we presented a framework for bridging the gap between the start and goal page of Deep Web pages. We have proposed a simple, but efficient, Deep

Web navigation strategy, which we have found to be very effective thus far. The main idea is to change a heavy-weight navigation calculus for an intermediate page identification procedure and a set of actions that navigate to the next page. 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.
For future work we plan to investigate how to automatically suggest meaningful and discriminatory keywords to the user and the use of more elaborate techniques to identify intermediate pages, such as the visual appearance of the Web page.

## References


1. Baumgartner, R., Ceresna, M., Ledermüller, G.: Deep Web Navigation in Web Data Extraction. In: CIMCA/IAWTIC, pp. 698–703. (2005)
2. Bergman, M. K.: The Deep Web: Surfacing Hidden Value. White Paper, http://www.brightplanet.com/images/stories/pdf/deepwebwhitepaper.pdf (2001)
3. Davulcu, H., Freire, J., Kifer, M., Ramakrishnan, I. V.: A Layered Architecture for Querying Dynamic Web Content. In: SIGMOD, pp. 491–502. (1999)
4. He, H., Meng, W., Yu, C. T., Wu, Z.: WISE-Integrator: A System for Extracting and Integrating Complex Web Search Interfaces of the Deep Web. In: VLDB, pp. 1314–1317. (2005)
5. He, B., Patel, M., Zhang, Z., Chang, K. C.-C.: Accessing the Deep Web. In: Commun. ACM, 50(5), pp. 94–101. (2007)
6. Hornung, T., Simon, K., Lausen, G.: Mashing Up the Deep Web - Research in Progress. In: WEBIST 2008. (2008)
7. Julasana, N., Khandelwal, A., Lolage, A., Singh, P., Vasudevan, P., Davulcu, H., Ramakrishnan, I. V.: WinAgent: A System for Creating and Executing Personal Information Assistants Using a Web Browser. In: IUI, pp. 356–357. (2004)
8. Kifer, M.: Deductive and Object-oriented Data Languages: A Quest for Integration. In: DOOD, pp. 187–212. (1995)
9. Lage, J. P., da Silva, A. S., Golgher, P. B., Laender, A. H. F.: Collecting Hidden Web Pages for Data Extraction. In: WIDM, pp. 69–75. (2002)
10. Nigam, K., McCallum, A., Thrun, S., Mitchell, T. M.: Learning to Classify Text from Labeled and Unlabeled Documents. In: AAAI/IAAI, pp. 792–799. (1998)
11. Pan, A., Raposo, J., Álvarez, M., Hidalgo, J., Viña, Á.: Semi-Automatic Wrapper Generation for Commercial Web Sources. In: EISIC, pp. 265–283. (2002)
12. Raposo, J., Álvarez, M., Losada, J., Pan, A. : Maintaining Web Navigation Flows for Wrappers. In: DEECS, pp. 100–114. (2006)
13. Simon, K., Lausen, G.: ViPER: Augmenting Automatic Information Extraction with Visual Perception. In: CIKM, pp. 381–388. (2005)
14. Wang, Y.: Deep Web Navigation by Example. In: Master's Thesis, Institute of Computer Science, Albert-Ludwigs University Freiburg. (2008)
15. Web Services Business Process Execution Language Version 2.0, `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`. (2007)
16. XML Path Language (XPath) Version 1.0, `http://www.w3.org/TR/xpath`. (1999)