Taking the OXPath down the Deep Web[*]

Andrew Sellers, Tim Furche, Georg Gottlob, Giovanni Grasso, Christian Schallhart
Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD
firstname.lastname@comlab.ox.ac.uk

ABSTRACT

Although deep web analysis has been studied extensively, there is no succinct formalism to describe user interactions with AJAX-enabled web applications.

Toward this end, we introduce OXPath as a superset of XPath 1.0. Beyond XPath, OXPath is able (1) to fill web forms and trigger DOM events, (2) to access dynamically computed CSS attributes, (3) to navigate between visible form fields, and (4) to mark relevant information for extraction. This way, OXPath expressions can closely simulate the human interaction relevant for navigation rather than rely exclusively on the HTML structure. Thus, they are quite resilient against technical changes.

We demonstrate the expressiveness and practical efficacy of OXPath to tackle a group flight planning problem. We use the OXPath implementation and visual interface to access the popular, highly-scripted travel site Kayak. We show, how to formulate OXPath expressions to extract all booking information with just a few lines of code.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

General Terms

Languages, Algorithms

Keywords

Web extraction, web automation, XPath, AJAX

1. INTRODUCTION

You need to meet your co-authors to discuss some breakthrough ideas. With grant money in short supply, a service *"to find out which of your co-authors' universities would be the cheapest location for a meeting, considering flight and accommodation"*, would be extremely useful.

[*]The research leading to these results has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement no. 246858.

```
doc("http://www.kayak.com/flights")//field()[5]/{$origin}
      /following::field()[@type='text'][1]/{$destination}
      /following::field()[last()]/{click /}
      //tbody[@class~='flightresult'][1]:<flight>/tr[2]
        [td[2]/a:<price=string(.)>] [td[4]:<airline=string(.)>]
```

Figure 1: OXPath for finding the cheapest flights

Everyone is daily faced with such problems: each individual piece of information is readily available on some webpage, but their (manual) extraction and aggregation is often unmanageable due to the number of possible combinations—forcing us to accept far from optimal solutions.

Extracting and aggregating web information is certainly not a new challenge. Previous approaches fall, in the overwhelming majority, into two classes: In the first class, service providers are obliged to deliver their data in a structured fashion, as in the case of the Semantic Web, Linked Open Data, or "Web 2.0-APIs", allowing their clients to process the received data with languages such as XPath, XQuery, or SPARQL. Since many service providers have no incentive for answering requests with such structured information, we often face approaches from the second class. In this case, we have to wrap unstructured information sources to extract and aggregate relevant data. However, wrapping a web site is often tedious, especially for AJAX-enabled web applications that reveal the relevant data only through user interactions. Most older works on web data extraction [6, 8, 4, 7, 9] do not adequately address web page scripting. Even where scripting is addressed, user input is often ignored, as in [2]. Otherwise, the simulation of user actions is neither declarative, nor succinct, but rather relies on imperative action scripts, as in [1].

Though XPath is the language of choice to query a set of nodes in an XML or HTML tree, it is aimed at static XML documents. However, many current Web applications, such as GMail or FaceBook, extensively rely on Javascript and HTML events to implement complex user interactions that cannot be adequately addressed in XPath.

Therefore, we introduce OXPath, an extension of XPath, to allow the declarative specification of user interactions with (scripted) web applications. We show that just four concise extensions over XPath enable OXPath to deal with scripted web applications while retaining XPath's declarativity and succinctness. Underlying these extensions is OXPath's ability to access the *dynamic DOM trees of a current browser engine*, reflecting all changes caused by scripting: (1) The *simulation of user actions*, such as filling form fields or hovering over a details button, enables interaction with AJAX applications which modify the DOM dynamically. (2) *Selection based on dynamically computed CSS attributes* allows navigation e.g. to the first green section title. (3) For expressing the interaction with forms, *navigation exclusively relying on visible fields* is essential. (4) Marking expressions allows *identification of relevant pieces* for extraction.

Our demonstration shows how OXPath is able to extract the data needed for finding the cheapest location for a group meeting from existing travel sites. Figure 1 gives a flavour of the language, displaying an OXPath expression for solving the above task.For details, see Section 3.

We also demonstrate a browser-based visual interface for OXPath which records user interactions (including DOM events, form fillings and navigation) with a Web site and allows visual selection of data values for extraction.

The OXPath prototype and interface are available on `diadem-project.info/oxpath` under the new BSD license.

2. OXPATH: LANGUAGE & SYSTEM

First and foremost OXPath is just XPath: Every XPath expression is also an OXPath expression. We extend XPath with (1) a new kind of location step (for actions and form filling), (2) a new axis (for selecting nodes based on visual attributes), (3) a new node-test (for selecting visible fields), and (4) a new kind of predicate (for marking data to be extracted). For page navigation, we also adapt the notion of Kleene star over path expressions from [5].

OXPath carefully extends XPath with these features in order to provide the necessary expressiveness for web data extraction while remaining able to make strong and favourable guarantees on time and memory w.r.t. expression evaluation. While XQuery was a candidate for extension to form a new language, it allows query composition and queries with arbitrary many variables rather than the two variable limitation of queries of XPath. These features may be useful for general XML processing, but we have found little use for them in web extraction. Certainly, inclusion of these features does not justify the performance penalty and the resulting uncertainty to time and memory requirements.

2.1 Simulating User Interaction

In this section, we limit our presentation to navigating scripted, form-based web applications.

Explicit Event Simulation. For explicitly simulating DOM 3 events such as clicks , OXPath introduces *action steps*. There are two types of action steps: *contextual action steps* such as **`{click}`** and *absolute action steps* such as **`{click /}`** with a trailing slash. Where XPath expressions select nodes from a single DOM, OXPath expressions may select nodes from multiple DOMs. Actions may modify the current DOM or replace it entirely. Thereby, absolute actions return the root nodes of the DOMs resulting from executing the action. Contextual actions return those nodes in the resulting DOMs, that are matched by the action-free prefix of the expressions. The *action-free prefix* is built by removing all intermediate contextual actions from the segment starting at the previous absolute action.

For instance, the following expression visits Google News and follows the link to the web site of the top story:

```
doc("news.google.com")/desc::h2.title[1]/a/{click /}
```

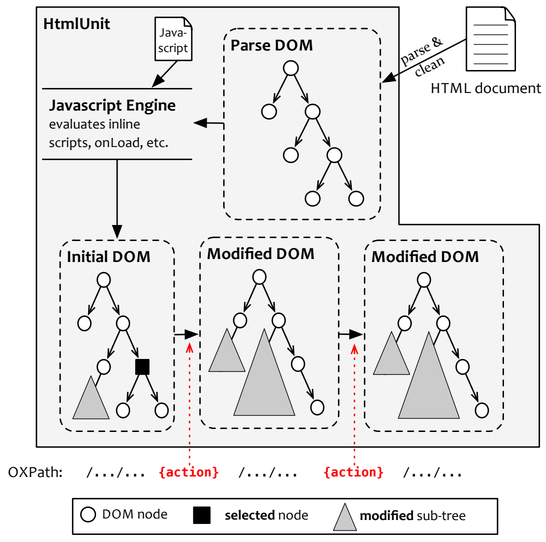


Figure 2: OXPath Event-Actions

We adopt the . operator from CSS as the node test for an HTML `class` attribute.

Figure 2 illustrates OXPath's architecture for simulating user interactions: We use the web testing framework HtmlUnit [3] to simulate a browser. First, an HTML document is cleaned, parsed, and any inline or on-load scripts are executed. On the resulting DOM, we evaluate the expression up to the first action, execute it (and any implicit intermediate actions) for each selected node, and obtain a set of new (or modified) DOMs. The expression up to the next action is evaluated against the new DOMs and so on.

Simulating Form Filling. In addition to actions for explicitly simulating DOM events, OXPath provides actions for filling form fields which usually trigger an entire series of events. The following expression searches for "Oxford":

```
doc("google.com")//input[@name='q']/{"Oxford"}/
                  following::input[@type='submit']/{click}
```

It navigates to `google.com`, simulates a user moving the focus to the search text field (identified by its id `q`), entering "Oxford" (simulating each keystroke to trigger keyboard events), and clicking on the submit button.

Form filling actions in OXPath are either strings for text fields, numbers for selecting options in a select box, or DOM event keywords. Simulated events invoke all appropriate event hanlders. OXPath also allows the use of XPath variables (such as `$x`) in form actions; variable bindings are provided by the environment.

Visual Selection. To allow easy and robust expressions for form filling, navigation between just the visible form fields is essential. Unfortunately, XPath as a general XML selection language does not give access to the visual attributes of DOM nodes. Therefore, we introduce in OXPath two extensions for lightweight visual navigation: a new axis for accessing CSS properties of DOM nodes and a new node test for selecting only visible form fields.

The **`style`** axis is comparable to the **`attribute`** axis, but navigates the (computed) CSS properties of a node rather than its properties. For example, we can select the sources for the top story on Google News using only visual information:

```
doc("news.google.com")//*[style::color="#767676"]
```

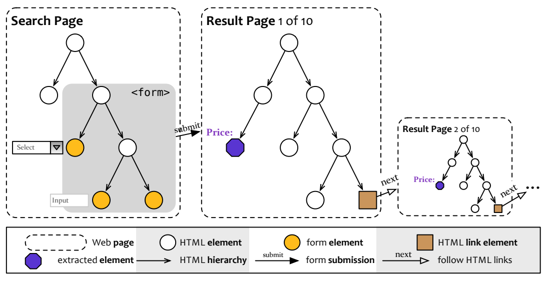


Figure 3: OXPath Page Iteration and Extraction

With the style axis, we can write expressions that are resilient to changes in the HTML structure or changes to `id` or `class` attributes. The style axis gives access to the actual CSS properties of DOM nodes (as returned by the DOM `style` object), regardless of where they are specified. This contrasts to an XPath expression using `attribute::style` which queries only inline style declarations.

The most common use of the `style` axis for deep web extraction is to navigate only over the *visible fields* of a page. This excludes those fields that have type or visibility `hidden`, and those that have display property `none` or are contained in such an element. To speed-up and simplify the navigation of visible form fields, we introduce the node-test `field()`. In the Google example, we can thus rely only on the order of the visible fields to fill the search form:

```
doc("google.com")/descendant::field()[1]/{"Oxford"}
                /following::field()[1]/{click}
```

Such an expression is not only easier to write, it is also far more robust against changes on the web site. For it to fail, either the order or set of visible form fields has to change.

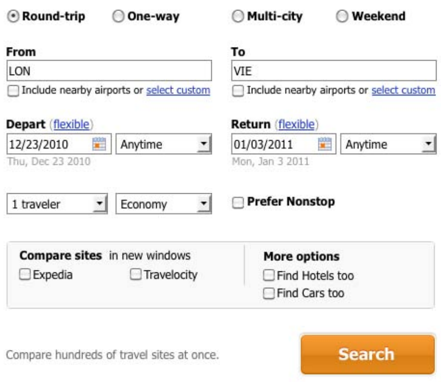
Page Iteration. Relevant web data is often organized in records over multiple pages each linking to the next. This is true for most result pages of web forms. Extracting information from all such results then requires following all next links until there is no further. For this purpose, we borrow the Kleene star from [5] to express the repeated navigation over a specified path. The expression `//a(/b/d)∗`, e.g., selects all nodes reached via alternating `b` and `d` nodes from an `a` node. For navigating result pages for web forms this is particularly useful: From the search result, we collect all the result pages by repeatedly clicking on the next link. Since we use an absolute action step, the remaining expression `//h3` is evaluated relative to the roots of all these pages.

```
... foll::field()[1]/{click /}
    (//table[@id='nav']/descendant::a[last()]/{click /})∗//h3
```

We can limit the number of pages navigated to by a positional predicate, as in any XPath step.

2.2 Data Extraction

Navigation and form filling are ultimately means to data extraction: In XPath, only a single node-set can be returned, while data extraction requires records to be made up by many related attributes. Attributes may be atomic or records themselves, thus allowing record nesting. Thus, the result of an OXPath expression is the set of selected nodes and an XML document with the extracted records. We introduce a new kind of predicate, the **extraction marker**, for indicating both representative nodes for records and their attributes. Extraction markers for records are denoted as in



[✓ Round-trip] One-way Multi-city Weekend

From: LON
Include nearby airports or select custom
To: VIE
Include nearby airports or select custom

Depart (flexible): 12/23/2010 Anytime
Thu, Dec 23 2010
Return (flexible): 01/03/2011 Anytime
Mon, Jan 3 2011

1 traveler | Economy | Prefer Nonstop

Compare sites in new windows: Expedia, Travelocity
More options: Find Hotels too, Find Cars too

Compare hundreds of travel sites at once.

Figure 4: Kayak flight search form

`:<story>` following the expression selecting the desired nodes. Those for attributes are denoted as in `:<title=string(.)>`, specifying the value to be extracted. For instance,

```
doc("news.google.com")//div[@class~="story"]:<story>
      [.//h2:<title=string(.)>]
      [.//span[style::color="#767676"]:<source=string(.)>]
```

extracts, from Google News, a `story` element for each current story, containing its title and its sources, as in:

```
<story><title >Tax cuts ...</title>
       <source>Washington Post</source>
       <source>Wall Street Journal</source> ...   </story>
```

The `:<source>` marker occurs in a predicate after `:<story>`. Thus, the extracted `source` elements are children of the `story` records in the output. In general, the structure of a record mirrors the structure of the OXPath expression: An extraction marker in a predicate represents an attribute to the (last) extraction marker outside the predicate. Attributes within a record are returned in document order.

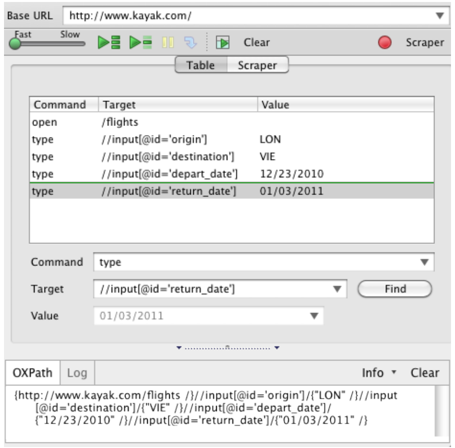
Figure 3 summarizes the typical extraction process on a form-driven web application: (1) On the search page, we navigate and fill form fields with the specified values and submit the form. (2) On the first result page, we extract all items that match OXPath expressions with extraction markers (here the `:<Price>`). (3) Most web sites paginate the results, and thus OXPath expressions often contain page iterations. On each result page, we follow the link to the next result page until there is no further such link.

3. DEMO DESCRIPTION

In the demo, we extract data from the flight booking site Kayak (`kayak.com/flights`) and present on our project page (`diadem-project.info/oxpath`) examples from other sites.

Web application. Kayak showcases OXPath's capability to deal with modern web applications as it uses a heavily scripted interface. Figure 4 shows its search form. Though that form contains a "Search" button, form submission is triggered by an event handler. Results are presented as structured records ordered by price. However, to access details such as the flight number, the user has to click on a link that retrieves those details asynchronously and displays them dynamically. In the demonstration, we give a brief run-through of searching on Kayak, including a visualization of the used event handlers.

Filling the Search Form. Let's turn again to the query from the introduction: *"Find out which of your co-authors' universities would be the cheapest location for a meeting,*



```
Base URL  http://www.kayak.com/
```

Command	Target	Value
open	/flights	
type	//input[@id='origin']	LON
type	//input[@id='destination']	VIE
type	//input[@id='depart_date']	12/23/2010
type	//input[@id='return_date']	01/03/2011

Command	type
Target	//input[@id='return_date']
Value	01/03/2011

```
{http://www.kayak.com/flights /}//input[@id='origin']/{"LON" /}//input
    [@id='destination']/{"VIE" /}//input[@id='depart_date']/
    {"12/23/2010" /}//input[@id='return_date']/{"01/03/2011" /}
```

Figure 5: OXPath Firefox plug-in

considering flights and accommodation". In the live demo, we show how to extract both prices for flights and accommodations, but here we focus only on flights. We assume that `$origin` and `$meeting` are bound one by one to all combinations of author's universities and possible meeting places.

We have implemented a visual interface, shown in Figure 5, for creating and testing OXPath expressions based on the Selenium IDE, a Firefox plug-in for web testing. It is able to record user actions, e.g., for form filling. In Figure 5, the interface is shown with a sequence of actions and OXPath expressions identifying the target nodes for those actions. The sequence is the result of recording the user actions for filling in origin and destination airport as well as travel times. If we continue filling and submitting the form, the necessary actions will be appended to the sequence. This interfaces additionally allows the visual selection of records and attributes to be extracted.

Below we show different OXPath expressions for identifying the same form fields or data, and how changes in the web site affect these expressions. To select the origin airport on Kayak, we can use any of the following expressions:

```
/descendant::*[105]      //input[@id='origin']
//field()[5]             //field()[@type='text'][1]
```

The first two are pure XPath expressions, while the latter two use OXPath's `field()` node-test. The expressions using `field()` navigate only among the form fields *visible to the user*. Thus the expression is resilient to most common changes such as adding hidden fields, changing the field ids, or modifying the structure of the container elements.

The following expression fills in origin and destination airport using the corresponding variables and submits the form:

```
  doc("www.kayak.com/flights")//field()[5]/{$origin}
2       /following::field()[@type='text'][1]/{$destination}
        /following::field()[last()]/{click /}
```

For both form inputs (`{$origin}` and `{$destination}`), we use contextual action steps to preserve the context. For example, the `following::field()` in line 2 is evaluated relative to the form field filled by `{$origin}`. It selects all visible fields after the one(s) the action is performed on. The destination airport is the first such field that also has type `text`. The "Search" button is the last visible form element. To submit the form, we use an absolute click action step on that button. We simulate all necessary events because Kayak uses event handlers for form validation and submission.

Extracting the Cheapest Flight. On the result page obtained by executing the above OXPath expression, we want to extract the cheapest flight as follows:

```
  <flight><airline>Austrian Airlines</airline>
2        <price>$177</price></flight>
```

Kayak returns flights sorted by price. Thus, we identify the first record as the first `flightresult` table body:

```
//tbody[@class~='flightresult'][1]:<flight>
```

The `:<flight>` extraction marker indicates that for each such row a `flight` element is extracted. Its attributes to be extracted are price and flight, both located in `td` children of its second row; thus, this expression finds the cheapest flight:

```
  doc("www.kayak.com/flights")//field()[5]/{$origin}
2       /following::field()[@type='text'][1]/{$destination}
        /following::field()[last()]/{click /}
4       //tbody[@class~='flightresult'][1]:<flight>/tr[2]
          [td[2]/a:<price=string(.)>] [td[4]:<airline=string(.)>]
```

OXPath can deal both with separate detail pages and with details that are dynamically inserted into the DOM by scripting, as on Kayak. For example, to get the flight number, we navigate from the flight record selected above to the details link, click it and extract the number:

```
  ...//tbody[@class~='flightresult'][1]:<flight>/tr[2]
2     [td[2]/a:<price=string(.)>] [td[4]:<airline=string(.)>]
      [.//a[contains(., 'details')]{click /}
4      /following::table[1]//nowrap[.#='Flights']:<nr=string(.)>]
```

Extracting all Prices. If we are not just interested in the cheapest flight, but rather all flights, we need to inspect several or even all result pages. For that, we use the Kleene-star construct from [5]:

```
(/descendant::a[contains(.,'Next')][1]{click /})*
```

This expression returns the root nodes of all pages reachable by following next links from the first result page or any subsequent result page. We conclude with the full OXPath expression to extract all flights with their price and airline:

```
  doc("www.kayak.com/flights")//field()[5]/{$origin}
2       /following::field()[@type='text'][1]/{$destination}
        /following::field()[last()]/{click /}
4       (/descendant::a[contains(.,'Next')][1]{click /})*
        //tbody[@class~='flightresult'][1]:<flight>/tr[2]
6         [td[2]/a:<price=string(.)>] [td[4]:<airline=string(.)>]
```

4. REFERENCES

[1] R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with Lixto. In *VLDB*, 2001.

[2] C. Duda, G. Frey, D. Kossman, and C. Zhou. AJAXSearch: Crawling, Indexing, and Searching Web 2.0 Applications. In *VLDB*, 2008.

[3] HtmlUnit. `http://htmlunit.sourceforge.net/`

[4] M. Liu and T. W. Ling. A rule-based query language for HTML. In *DASFAA*, 2001.

[5] M. Marx. Conditional XPath. *TODS*, 30(4), 2005.

[6] A. Mendelson, G. Mihaila, and T. Milo. Querying the World Wide Web In *DIS*, 1996.

[7] QL2 Software. WebQL. `http://www.ql2.com/`.

[8] A. Sahuguet and F. Azavant. WysiWyg web wrapper factory (W4F). In *WWW*, 1999.

[9] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.