OXPath: Little Language, Little Memory, Great Value*

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

Data about everything is readily available on the web—but often only accessible through elaborate user interactions. For automated decision support, extracting that data is essential, but infeasible with existing heavy-weight data extraction systems. In this demonstration, we present OXPath, a novel approach to web extraction, with a system that supports informed job selection and integrates information from several different web sites. By carefully extending XPath, OXPath exploits its familiarity and provides a light-weight interface, which is easy to use and embed. We highlight how OXPath guarantees optimal page buffering, storing only a constant number of pages for non-recursive queries.

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

Where do you want to work tomorrow? As three of your job applications have been accepted, you have to pick one where *(1)* your wife can find a good job as well, *(2)* you don't have to pay through the nose for a rental, *(3)* you can follow your passions, and *(4)* your neighbors are used to children.

All this information is readily available on some webpage, but manually extracting, aggregating, and ranking that information is tedious and often unmanageable due to the size of the relevant data. We are forced to settle with approximate solutions agonizing not to miss some crucial data.

*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. The views expressed in this article are solely those of the authors.

WWW 2011, March 28–April 1, 2011, Hyderabad, India.
ACM 978-1-4503-0637-9/11/03.

```
doc("rightmove...")/desc::field()[1]/{"Oxford"}/foll::*#rent
  /{click/}//*#maxPrice/{"1,250 PCM"/}/foll::field()[last()]
  /{click/}/(//a[.#='next']/{click/})*//ol#summaries/li[
    .//a[.#='details']/{click/}//text()[.='Furnished']]:<R>
```

Figure 1: OXPath for finding affordable rentals

Automating this process, however, is not affordable with existing web querying, extraction, or automation tools. Web querying tools do not support navigation between web pages, form filling, or the simulation of user actions, all essential for our task. Web extraction and automation approaches [2, 4, 7, 3] address these issues, but are not affordable either: *(1)* They use specifically designed, complex languages [2, 4, 7], which are unfamiliar to web developers. *(2)* Yet most cannot deal properly with many modern, scripted web sites as pointed out in [1]. *(3)* They rely on a sophisticated, heavy weight infrastructure, often including server and frontend components. *(4)* They require considerable hardware resources: Neither provides strict memory guarantees and only [2] provides bounds for extraction time.

In this demonstration, we introduce **OXPath**, a careful extension of XPath for simulating user interactions with web sites to extract data from such sites. OXPath is

(1) *affordable to learn* as it extends XPath with only four new features, sufficient for solving most extraction tasks.

(2) *expressive enough* for most web extraction tasks. In particular, it deals with scripted web sites as it is based on an existing browser engine.

(3) *affordable to execute*: Evaluating OXPath is polynomial in the number of visited HTML elements. Its memory consumption is independent of the number of visited pages. To the best of our knowledge, OXPath is the first web extraction tool that can guarantee constant page buffering for non-recursive extraction queries.

(4) *affordable to embed* as it builds on existing technologies and requires little additional processing over XPath.

We demonstrate these characteristics of OXPath in a decision support system for selecting the ideal job: The user can specify UK locations and weighted criteria such as the number of jobs, the availability of affordable rentals, etc. Each criteria has some parameters, e.g., the wage range for jobs. For each location the relevant data is extracted with OXPath (from different web sites) and the ranked list is presented to the user. Figure 1 shows how to extract apartments from `rightmove.co.uk`. We select the first (visible) form field on the start page and enter "Oxford". From there we navigate to and click the rent submit button. On the returned page, we fill the maximum price field and click the submit button (the last visible form element). We iterate over all result

pages by clicking the next links and retrieve each furnished rental (recognized on a details page).

In the second demo part, we profile OXPath's evaluation algorithm: We visualize the tree of pages as navigated by an expression and show the time to render and select the relevant data, illustrating that often rendering time dominates the evaluation. We also visualize OXPath's buffer management and highlight when pages are dropped from the buffer. This reinforces our theoretical result: Even for very large extraction tasks, OXPath rarely buffers more than a handful pages at any time. OXPath is available on `diadem-project.info/oxpath` under a BSD license.

2. OXPATH: LANGUAGE

OXPath is an extensions of XPath: XPath expressions are also OXPath expressions and retrain their same semantics, computing sets of nodes, integers, strings or Booleans.

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 adapt the notion of Kleene star over path expressions from [5]. Nodes and values marked by extraction markers are streamed out as records of the result tables. For efficient processing, we cannot fix an apriori order on nodes from different pages. Therefore, we do not allow access to the order of nodes in sets that contain nodes from multiple pages.

Actions. For explicitly simulating user actions, such as clicks or mouse-overs, OXPath introduces *contextual action steps*, as in `{click}`, and *absolute action steps* with a trailing slash, as in `{click /}` . Since actions may modify or replace the entire DOM, OXPath's semantics assumes that they produce a new DOM. Absolute actions return DOM roots, while contextual actions return those nodes in the resulting DOMs which are matched by the action-free prefix of the performed action: The *action-free prefix* AFP(*action*) of *action* is constructed by removing all intermediate contextual actions and extraction markers from the segment starting at the previous absolute action. Thus, the action-free prefix selects nodes on the new page, if there are any. For instance, the following expression enters "Oxford" into Google's search form using a contextual action—thereby maintaining the position on the page—and clicks its search button using an absolute action.

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

Style Axis and Visible Field Access. We introduce two extensions for lightweight visual navigation: a new axis for accessing CSS DOM node properties and a new node test for selecting only visible form fields. The `style` axis is similar to the `attribute` axis, but navigates dynamic CSS properties instead of static HTML properties. For example, the following expression selects the sources for the top story on Google News based on visual information only:

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

The `style` axis provides access to the actual CSS properties (as returned by the DOM `style` object), rather than only to inline styles.

An essential application of the `style` axis is the navigation of *visible fields*. This excludes fields which have type or visibility `hidden`, or have display property `none` set for themselves or in an ancestor. To ease field navigation, we introduce the node-test `field()` as an abbreviation. In the above Google search for "Oxford", we rely on the order of the visible fields selected with `descendant::field()[1]` and `following::field()[1]`. 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.

Extraction Marker. Navigation and form filling are often means to data extraction: While data extraction requires records with many related attributes, XPath only computes a single node set. Hence, we introduce a new kind of qualifier, the *extraction marker*, to identify nodes as representatives for records and to form attributes from extracted data. For example, `:<story>` identifies the context nodes as story records. To select the text of a node as title, we use `:<title=string(.)>`. Therefore,

```
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 nesting in the result above mirrors the structure of the OXPath expression: An extraction marker in a predicate represents an attribute to the (last) extraction marker outside the predicate.

Kleene Star. Finally, we add the Kleene star, as in [5], to OXPath. For example, we use the following expression to query Google for "Oxford", traverse all accessible result pages, and to extract all contained links.

```
doc("google.com")/descendant::field()[1]/{"Oxford"}
  /following::field()[1]/{click /}
  ( (/desc::a.l:<Link=(@href)>)
    /ancestor::*/desc::a[next]{click /} )*
```

To limit the range of the Kleene star, one can specify upper and lower bounds on the multiplicity, e.g., `(...)*{3,8}`.

3. OXPATH: SYSTEM

Our implementation consists of the three layers shown in Figure 2: the *embedding layer* provides a development API and a host environment, the *engine layer* performs the evaluation of OXPath expressions, and the *web access layer* accesses the web in a browser-neutral fashion.

Embedding Layer. To evaluate an OXPath expression, we need to provide the environment with bindings for all occurring XPath variables; the environment in turn provides the final expression to the OXPath engine. Variables in OXPath expressions are commonly used to fill web forms with multiple values. To this end, the host environment allows value bindings based on databases, files, other OXPath expressions, or Java functions. In our default implementation, we stream the extracted matches to a file without buffering, while other implementations may choose to store the matches e.g. in a database instead. Finally, we offer a GUI to support the visual design of the OXPath queries.

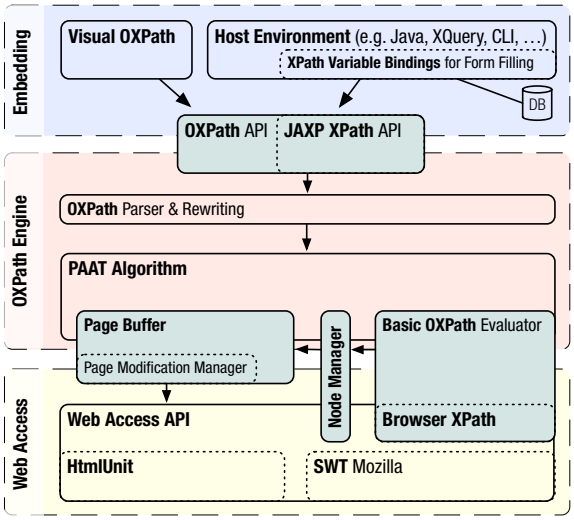

Figure 2: OXPath System Architecture

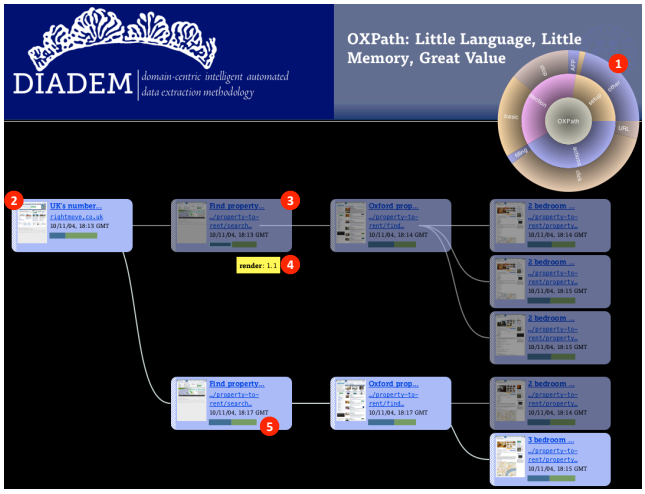

Figure 3: OXPath Visual Profiler

Engine Layer. After parsing, the *query optimizer* expands abbreviated expressions, such as `field()`, and feeds the resulting queries to our Page-At-A-Time (PAAT) algorithm [6]. This algorithm controls the overall evaluation strategy and uses the browser's XPath engine to evaluate individual XPath steps and a buffer manager to handle page modifications. In this demonstration (see Section 4), we illustrate the inner workings of the buffer manager using the OXPath visual profiler shown in Figure 3. The visual profiler is an AJAX webpage that uses the InfoVis Toolkit (`thejit.org`) to replay or step through the evaluation of an OXPath expression. Its main feature is a visualization of the page navigation tree, showing how pages are traversed by OXPath.

Web Access Layer. For evaluating OXPath expressions on web pages, we require programmatic access to a dynamic DOM rendering engine, as employed by all modern web browsers. We identified HtmlUnit (`htmlunit.sourceforge.net`), the Mozilla-based JREX (`jrex.mozdev.org`), and the also Mozilla-based SWT widget (`eclipse.org/swt`) as backends. To decouple our own implementation from their implementation details, we provide the web access layer as a facade which allows for exchanging the underlying browser engine independently of our other code. In our current implementation, we use HtmlUnit as backend, since it renders pages efficiently (compared with the SWT Mozilla embedding) and is implemented entirely in Java.

4. DEMO DESCRIPTION

For this demonstration, we showcase OXPath with a proof-of-concept decision support system (`diadem-project.info/oxpath/placerank`): It ranks a set of UK locations using data extracted from a number of different web sites with user provided criteria and weights. Separate OXPath expressions navigate the interfaces of each site, filling the necessary values for each location. The criteria are realized either through form filling, if the web site already provides a corresponding option, or as OXPath predicates otherwise. OXPath extracts all the relevant data, the actual weighting and ranking is done in the host language; in our case, a straightforward Java program.

We extract and aggregate data from several deep web sites: Reed (`reed.co.uk`) for jobs, Rightmove (`rightmove.co.uk`) for rentals, Yahoo's Upcoming (`upcoming.yahoo.com`) for events, Thomson Local (`thomsonlocal.com`) for local businesses, and UpMyStreet (`upmystreet.co.uk`) for neighborhood assessment. For each site, the user may also provide a list of criteria for ranking the postcode, though the system proposes a default set. These criteria can be built from the data of the above web sites using a number of additional parameters for each (criteria in italics are not available through the search interface of the respective site and thus are realized through OXPath predicates):

- From Reed, we extract the number of jobs that match the following criteria (in the given postcode): job category, salary range, and *number of applicants*.
- From Upcoming, we extract the number of events in the specified postcode area, allowing the user to configure the period of time, and event category (e.g., "Jazz").
- From Thomson Local, we extract local businesses if their category and *distance from our postcode* match.
- From Rightmove, we extract all rentals that have the chosen number of bedrooms, maximum rental, *included furnishings*, and *availability (e.g., long term)*.
- From UpToMyStreet, we extract any of four neighborhood quality measures (averages): income, education level, and number of children.

On the website, we also show the OXPath implementations for all five web sites. For space reasons, we highlight here only the most interesting OXPath expressions. Figure 4 shows the navigation and extraction performed on Reed.

We first load Reed's start page and fill the search field for the job category ❶, the location ❷, and the maximum distance ❸ from the given postcode. To submit the search form, we `click` the "Search Jobs" button ❹:

```
doc("reed.co.uk")//field()[@title#='keyword']/{"Java"/}❶
  //field()[@title#='town']/{"OX1"}❷/foll::field()[1]
  /{"5 miles"}❸/foll::field()[@title='SearchJobs']/{click/}❹
```

For space reasons, we abbreviate the XPath axes. The `#` operator is a node test we adopt for convenience: `A#B` filters the nodes in `A` to those with `id` attribute `B`. The node-test `field()` is used to identify only visible form fields.

Figure 4: Finding an OXPath through `reed.co.uk`

Since the search may return a paginated result, we use an OXPath Kleene star expression (❻+) to iterate over all the result pages: On each result page, we identify the "Next" link at the top of the results listing, until there is no further:

```
/(//a[.#="Next"][1]/{click/})*❻+
```

On each result page, we identify and extract those job listings that have less than the given number of maximum applicants (in this case 10):

```
//div.resultEntry[.//label[.#="Applications"]
                          /foll::text()[1]<10]:<job>
```

The extraction marker `:<job>` extracts each `div.resultEntry` with the label `job` if it matches the predicates.

For those jobs, we also access their details page to extract the full job description (rather than the abbreviated one shown on the result page):

```
[h4/a/{click/}❺//div.jobDescription:<desc=.>]
```

The extraction marker `:<desc>` extracts those descriptions as children of the corresponding `job` element, as it occurs nested in a predicate following `:<job>`.

Further criteria such as salary range or posting date can be included in the same way (full example on the website).

To visualize OXPath's buffer management, we use the OXPath profiler (Figure 3): It shows the tree of pages that OXPath navigates while evaluating the expressions. The evaluation can be replayed either automatically or stepwise. At the top of the page ❶, a summary of the evaluation time of the various components of OXPath is given. In the page tree, we show for each page ❷ some meta-data and the time spent rendering and navigating that page ❺. A page is shaded gray ❸ if it is no longer buffered by OXPath. As Figure 3 shows, OXPath keeps at most the pages on the path from the start page to the current one in memory. This behavior is confirmed on a larger scale in Figure 5, where we evaluate the above OXPath expression with location "London" and distance "30 miles".

We conclude the demonstration by going into details for one more web site, to illustrate the versatility of OXPath: extracting rentals from `rightmove.co.uk`. On Rightmove the user has to perform a longer sequence of actions (than on Reed) to obtain the results: Fill "Oxford" into the input field and click on the "Rent" submit button. On the returned page, fill in the maximum price field, and click submit. To find all result pages, repeatedly click the next link.

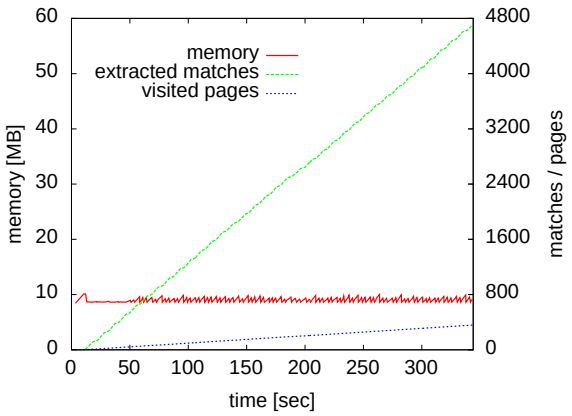

Figure 5: Memory, results, and pages

```
1 doc("rightmove...")/desc::field()[1]/{"Oxford"}❶/foll::*#rent
2   /{click/}❷//*#maxPrice/{"1,250 PCM"/}❸/foll::field()[last()]
3   /{click/}❹/(//a[.#='next']/{click/})*❺//ol#summaries/li[
4     .//a[.#='details']/{click/}❻//text()[.='Furnished']]:<R>
```

Figure 6: OXPath for finding affordable rentals

The OXPath expression in Figure 6 realizes this sequence of actions in Lines 1–4: To fill the first input field, we use OXPath's `field()` nodetest and a contextual action ❶ (denoted with just `{}` around the action). The navigation continues from the same field after a contextual action. The other actions are absolute actions (with an added `/`) where the navigation resumes at the root of the page retrieved by the action. We locate and click ❷ the "Rent" button to submit the form. Rightmove does not directly return results, but asks for a refinement of the query. On the refinement page, we identify the drop-down list for the maximum price and also submit the refinement form by clicking ❹ its submit button (the last visible web form field). Again we iterate over all result pages. In order to extract only "Furnished" apartments, we need to navigate to the details page ❻. We extract the filtered results from the result page using the `:<R>` extraction marker.

5. REFERENCES

[1] A. Alba, V. Bhagwan, and T. Grandison. Accessing the deep web: when good ideas go bad. In *OOPSLA*, 2008.

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

[3] J. P. Bigham, A. C. Cavender, R. S. Kaminsky, C. M. Prince, and T. S. Robison. Transcendence: enabling a personal view of the deep web. In *IUI*, 2008.

[4] M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller. Automation and customization of rendered web pages. In *UIST*, 2005.

[5] M. Marx. Conditional XPath. *ACM Trans. Database Syst.*, 30(4), 2005.

[6] OXPath. `http://www.diadem-project.info/oxpath`.

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