# The OXPath to Success in the Deep Web *

Andrew Sellers
supervised by Georg Gottlob
Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD
Andrew.Sellers@comlab.ox.ac.uk

## ABSTRACT

The world wide web provides access to a wealth of data. Collecting and maintaining such large amounts of data necessitates automated processing for extraction, since appropriate automation can perform extraction tasks that would be otherwise infeasible. Modern web interfaces, however, are generally designed primarily for human users, delivering sophisticated interactions through the use of client-side scripting and asynchronous server communication. To this end, we introduce OXPath, a careful extension of XPath that facilitates data extraction from the deep web. OXPath exploits XPath's familiarity and theoretical foundations. OXPath, then, achieves favourable evaluation complexity and optimal page buffering, storing only a constant number of pages for non-recursive queries. Further, OXPath provides a lightweight interface, which is easy to use and embed. This paper outlines the motivation, theoretical framework, current implementation, and preliminary results obtained so far. We conclude with proposed future work on OXPath, including an investigation of how to deploy OXPath efficiently in a highly elastic computing framework (cloud).

## 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. PROBLEM

The interactive nature of modern web interfaces exacerbates an unfortunate problem: the dynamic nature of these user interfaces, driven by client and server-side scripting (e.g. AJAX), creates challenges for automated processes to access this information with the techniques developed for extracting static web content.

The deep web, the part of the web accessible only through such web interfaces, contains significant amounts of useful information; while each individual piece of information is readily available on some webpage, their manual extraction and aggregation is often impractical due to the expense required for human users to tediously navigate the relevant web interfaces for each query and extract relevant results.

As identified in [1], an appropriate automated tool for web data extraction must be more than a mere screen scraper. Useful, generalized extraction requires navigation of web applications in a way that simulates a human user as well as the ability to accurately isolate the data for extraction while capturing its association to other extracted results. Further, extraction may occur over several web pages and data may be extracted repeatedly over time, necessitating that such a tool support identification of results even if multiple formats are used or minor visual changes occur.

To enable automation, data accessible to humans through existing web interfaces needs to be transformed into structured information. This is the task of web extraction tools. To address these challenges, web extraction tools must **(1)** *interact* with rich interfaces of (scripted) web applications to find the interesting data and to perform actions on behalf of the user, **(2)** provide extraction capabilities sufficiently *expressive* and *precise* to specify the relevant data to be extracted in the majority of web extraction tasks, **(3)** while remaining *robust* enough to compensate for changes in the presentation of web content, yet **(4)** be *easy to learn* for web developers and **(5)** *scale* well even if the number of relevant web sites is large, and **(6)** *embed* easily in existing web programming environments both on servers and in clients. The latter two desiderata are essential for web extraction tools to become a staple of web programmers to interact with web applications, just as XPath or CSS are for accessing single web pages today.

## 2. STATE OF THE ART

The automatic extraction and aggregation of web information is not a new challenge. Previous approaches, in the overwhelming majority, either (1) require service providers to deliver their data in a structured fashion (e.g. the Semantic Web); or, (2) "wrap" unstructured information sources to extract and aggregate relevant data. The first case levies requirements that service providers have little incentive to adopt, which leaves us with wrapping as the only realistic

*Partially based on joint work with Tim Furche, Giovanni Grasso, and Christian Schallhart. 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 author.


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

choice. Wrapping a web site, however, is often tedious, since many AJAX-enabled web applications reveal the relevant data only through user interactions. As recognized in [1], the transformation of the web from primarily static pages to highly scripted web applications has posed a significant challenge to existing web extraction and automation tools.

Traditional web extraction tools (e.g. [9] and [6]) are unable to deal with highly scripted web sites. More recent tools have moved towards an approach of recording user actions in an actual browser and replaying those actions for extracting data: Lixto [3] comprises a fully visual and interactive wrapper generator framework, based on the Mozilla browser embedded in an Eclipse environment. Though data extraction is based on the declarative, highly expressive Elog language, actions are scripted separately in imperative action scripts. There are also no guarantees on the number buffered pages. In the spirit of Lixto, several visual wrappers allow action replay, e.g. WARGO [14]. Lixto also uses separate languages for navigation and extraction; its navigation language is severely limited by the lack of iteration and conditions.

Deep web extraction tools such as [4] have become adept at dealing with scripted, highly visual web sites, but follow an approach where extraction programs are inferred automatically from examples provided by the user. Though this allows for easy wrapper generation, the precision necessary for many fully automated tasks is sacrificed. They also do not consider page management or evaluation complexity. BODE [16] is a browser-based extraction tool with an extraction language called BODED. But the language is imperative and, though flexible for the integration of additional functionality, hard to optimize. Again, page buffering is not considered.

There has also been previous work focused on finding a language for web automation: a single sequence of user action is replayed, rather than an exhaustive traversal of all user action sequences leading to interesting data as in web extraction. Thus efficiency and page buffer management have not been considered extensively. Chickenfoot [5] is a programming system for web automation that allows users to program Javascript scripts that run in Firefox, having access to the rendered view of a page. Though useful for rapid wrapper generation, their results are often unpredictable to users and lack the precision and formal semantics offered by a declarative language. There are a number of other systems that rely on recording navigation from user actions, e.g., WebVCR [2] and WebMacros[15] or more recently [12], but all suffer from limitations on modern web pages and consider only the replay of a single action sequence rather than scalable multi-path data extraction.

# 3. PROPOSED APPROACH

Given the current state of web data extraction, we identify in Section 1 the six key requirements a data extraction tool must satisfy to deal with modern web applications. The solution we propose is OXPath, a careful extension of XPath, that allows the declarative specification of user interactions with (scripted) web applications. We show that just four concise extensions of XPath address all six requirements and enable OXPath to effectively extract data from 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) Extraction markers allow *identification of relevant pieces* for extraction.

XPath forms an optimal foundation for OXPath as it is (1) a declarative language, so we are able to readily adapt its clean semantics, data model, and favourable evaluation characteristics; (2) well known and studied by web practitioners; and, (3) readily embeddable in other platforms.

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.

OXPath, however, complements XPath with novel features specific to HTML, allowing the full specification of web data extraction tasks, including user interaction, iteration over result pages, and extraction of relevant, structured data.

Though there is an argument for more expressive languages for web extraction (such as monadic Datalog [7]), we believe that OXPath hits the sweet spot of being easy to learn, lightweight, and highly performant, yet capable enough for most extraction scenarios. In [8] the value of basic XPath as a component in data extraction is demonstrated. We build on that and extend XPath for page navigation, form filling, and extraction, turning it into a full-fledged extraction language that is still compact and efficient with optimal memory complexity. Further, OXPath is embeddable and familiar, making it ideal for deployment into other web technologies.

# 4. METHODOLOGY

In this section, we present details of the OXPath language and system. For space reasons, we omit discussion here of the OXPath data model, semantics, our Page-At-A-Time (PAAT) algorithm, and associated complexity results; these are discussed in [13]. We define the language and provide an implementation so that appropriate experiments can be conducted; in particular, we are interested in the time and memory costs to OXPath query evaluation, the cost of our additional features compared to standard XPath, and a characterization of OXPath expression evaluation when compared to page fetching and rendering time.

## 4.1 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 [11]. 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>
```

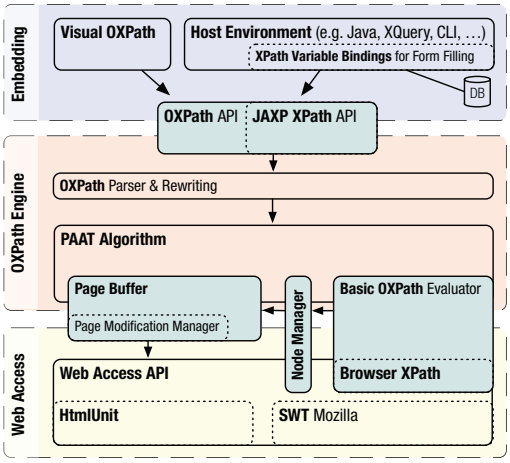


**Figure 1: OXPath System Architecture**

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 [11], 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 /}/
   ( descendant::a.l:<Link=(@href)>
     /ancestor::*/descendant::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}`.

## 4.2 OXPath: System

Our implementation consists of the three layers shown in Figure 1: 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.

*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 [13]. 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.

*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.

## 5. RESULTS

We have already achieved significant results. In particular, in [13], we prove the following theorem that establishes an upperbound to OXPath expression evaluation based on the `eval` function, the function implemented in the PAAT algorithm:

THEOREM 1 (PAAT FOR OXPATH). *Evaluating an OXPath expression* Expr *using* `eval` *takes at most* $O((p \cdot n)^6 \cdot q^3)$ *time and* $O(n^5 \cdot (q+d)^2)$ *where* $q$ *is the size* Expr *and* $n$ *is the maximum number of nodes in any page accessed by the evaluation of* Expr*,* $p$ *is the number of such pages, and* $d$ *is the maximum level of such a page.*

Even with the theoretical worst case, OXPath remains polynomial, and therefore, tractable. However, as our empirical analysis will show below, OXPath expression evaluation time is dominated by page retrieval and rendering time.

The other major result relates to page management in OXPath. The buffer management features of the PAAT algorithm ensures that pages are only buffered when they will be needed for further evaluation. We are then able to proof the following:

THEOREM 2 (MINIMAL BUFFERING). *The minimal number of page buffers necessary for evaluating an OXPath expression* Expr *is the maximum number of branching points on any path to a page in the page navigation tree that is reached when evaluation* Expr.

This notion of minimal buffering is empirically observed by measuring the memory footprint over time during the evaluation of an OXPath expression.

We continue with a brief discussion of evaluation of our implementation. We profile OXPath's evaluation time and memory on complex queries. Further, we compare its performance against both the speed for retrieval and rendering of web pages in the underlying browser as well as the evaluation time of its XPath engine. The experiments have been performed on an Intel Core 2 Duo with four 3.00GHz cores and 3 GB main memory, running Windows XP 32-bit using Java 1.6.0.21.

*Scaling OXPath.* In order to demonstrate how OXPath's PAAT algorithm scales, in particular where memory use is concerned, we run ten different types of queries that required complex page buffering, traversal of as many as 1000 pages, and extraction of thousands of pieces of information. Each query is evaluated multiple times and the results are averaged. The results of one such query is shown in Figure 2. Pages retrieved and results extracted are linear w.r.t. time, but memory size remains constant even as both the number of pages accessed and the results extracted increases. The jitter in memory use is due to the repeated ascends and descends of the citation hierarchy. This same characterization of memory, pages, and results over time was observed in all of the tested expressions.

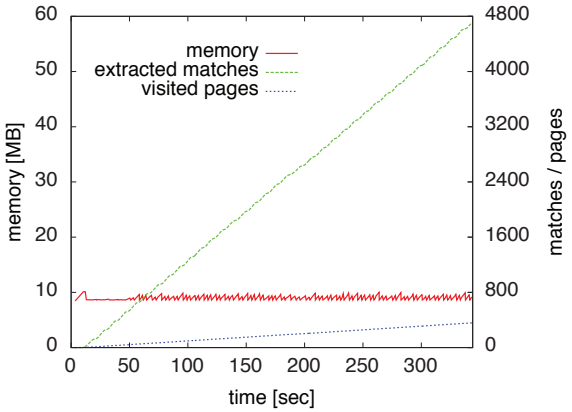


**Figure 2: Memory, results, and pages**

*Scaling from Basic XPath to Full OXPath.* To judge the practical performance of OXPath's implementation, we compare it with the built-in XPath engine of HtmlUnit (based on Java's default XPath engine). Second, we evaluate how OXPath's actions affect query evaluation, in particular how contextual actions perform compared to absolute actions. Finally, we show that the most expensive feature introduced by OXPath, the Kleene star, still scales well with query size. All queries are evaluated on Google.

HtmlUnit's XPath engine (Xalan) is shown to be exponential in increasing query size, whereas OXPath's PAAT algorithm is linear as expected. The queries tested were a set of 15 expressions built as follow: starting with the base query `//div/parent::div`, the $(i+1)_{th}$ expression is obtained by appending the pattern `/div/parent::div` to the $i_{th}$ one.

We also evaluated actions on pages that do not result in new page retrievals, comparing absolute actions and contextual action evaluation. Contextual actions suffer a small penalty to evaluation time compared to their absolute equivalents, but generally the evaluation time does not increase significantly with the number of performed actions. This result is to be expected, as contextual actions "replay" their action-free prefixes in order to find their place in the document. This penalty is minimal, however. Our experiments found the overhead was less than 15% even after six such actions are evaluated as part of the same expression for a reasonable web page consisting of a sophisticated web form. This data was the average of several runs on the web form found at `google.com/advanced_product_search`.

Our tests show the performance of the Kleene-star performs linearly as expected with increasing number of expansions of the Kleene star operation.

*Profiling OXPath.* In the final experiment, we profile the influence of the different evaluation steps on the overall evaluation time of an OXPath expression. We evaluated the same query on the following web pages: `apple.com`, `bing.com`, `diadem-project.info`, `www2011india.com`, and the Wikipedia page on Hyderabad, retrieving links on each of these pages, clicking on all of them and extracting the `html` tag of each result page. The average of profiling in several trials determined that OXPath evaluation was less than 2% of total processing time, the rest of the time can be attributed to initial browser setup (13%) and page fetching and rendering (85%).

# 6. CONCLUSIONS AND FUTURE WORK

OXPath is the first web extraction system with strict memory guarantees to the best of our knowledge. These memory guarantees reflect strongly in our experimental evaluation. Just as important, OXPath is built on standard web technology, such as XPath and DOM, so that it is familiar and easy to learn for web developers. We believe that it has the potential to become an important part of the toolset of developers interacting with the web. To further simply OXPath expressions and enhance their robustness, we plan to investigate additional features, such as more expressive visual language constructs and multi-property axes.

While we are encouraged by our results so far, we are excited by the potential offered by further investigation into OXPath. A strength of OXPath is that it is focused and easily embeddable. We want to exploit that potential by realizing OXPath in a variety of contexts: in the cloud for large-scale extraction and in the browser for ad-hoc and development use.

*In the Cloud.* OXPath is designed for highly parallel execution: the host language can assign different bindings for the same variable to create multiple OXPath queries. These queries can be processed with separate web sessions hosted on separate computing instances. We think this approach to parallel decomposition is ideally suited for the share-nothing nature of computing instances in elastic computing environments (e.g. Amazon's EC2 service at `aws.amazon.com`). The selection of a host language for the cloud requires careful consideration: in particular, most existing web programming languages, such as XQuery, do not provide access to a dynamic DOM. Beyond variable bindings, any useful host language almost certainly requires aggregation and subquerying capabilities; for this reason, a possible host language could extend SQL or XQuery.

Significant work remains to adapt OXPath to execute favourably in a cloud. The naive approach replicates our OXPath engine into arbitrary many computing instances. Significant challenges to this approach warrant further investigation: (1) One significant challenge to consider is that our current implementation requires a rendered DOM for each OXPath expression. Replicating so many web browsers may be undesirable. (2) Further, if we replicate our current implementation in many cloud-based instances, a significant fraction of action sequences may be repeated. Similarly, in multi-way navigation, repeated paths also occur; these "replays" may not always be necessary, demanding optimizations. (3) We also must address how to consolidate the output extracted by OXPath expressions running in multiple instances. The conventional engineering approach for such a task in Amazon Web Services (AWS) is to store interim results in ephemeral instance storage and combine them via MapReduce or a similar technique. We will have to evaluate whether such an approach is appropriate for OXPath or if new control mechanisms for cloud-based output must be introduced into the language. (4) OXPath in a cloud also necessitates formulation of etiquette rules. We cannot allow OXPath, as a web "citizen", to consume resources of public web servers to such an extent that it degrades or inhibits their operation. One possible solution to explore is detecting when caching pages is possible so that multiple accesses do not involve the server. This would introduce a notion of locality to our cloud, as we would have to manage which instances store cached pages and optimally evaluate appropriate expressions from them.

Each of these issues raises the question of how to optimize the evaluation of several parallel OXPath expressions in order to minimize expensive browser instantiations, unnecessary replication of common action sequences, and how to best consolidate output extracted by OXPath expressions running in multiple instances. The design of a sufficient task scheduler dictates how these features should be specified and integrated within a coherent approach, balancing these features between host language and core OXPath. Depending on our findings, we may need to adjust our semantics to support such optimizations.

*In the Browser.* Our implementation already proves the viability of OXPath operation within a browser. Further browser-based work includes development of a tool suite, including a visual generation and debugging environment for expressions. As a concise language with a clear formal semantics, OXPath is amenable to significant optimization and a good target language for automated generation of web extraction programs. In particular, we consider the automatic generalization of OXPath expressions to support robust web data extraction and ease usability with shorter and clearer expressions.

# 7. REFERENCES


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

[2] V. Anupam, J. Freire, B. Kumar, and D. Lieuwen. Automating web navigation with the webvcr. In *WWW*, pages 503–517, 2000.

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

[4] 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.

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

[6] C.-H. Chang, M. Kayed, M. R. Girgis, and K. F. Shaalan. A survey of web information extraction systems. *TKDE*, 2006.

[7] G. Gottlob and C. Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. of the ACM*, 2004.

[8] M. Kowalkiewicz, M. E. Orlowska, T. Kaczmarek, and W. Abramowicz. Robust web content extraction. In *WWW*, 2006.

[9] A. H. F. Laender, B. A. Ribeiro-Neto, A. S. da Silva, and J. S. Teixeira. A brief survey of web data extraction tools. *Sigmod Rec*, 2002.

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

[11] M. Marx. Conditional XPath. *TODS*, 2005.

[12] P. Montoto, A. Pan, J. Raposo, F. Bellas, and J. López. Automating navigation sequences in AJAX websites. In *ICWE*, 2009.

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

[14] J. Raposo, A. Pan, M. Álvarez, J. Hidalgo, and n. A. Vi. The Wargo system: Semi-automatic wrapper generation in presence of complex data access modes. In *DEXA*, 2002.

[15] A. Safonov. Web macros by example: users managing the WWW of applications. In *CHI*, 1999.

[16] J.-Y. Su, D.-J. Sun, I.-C. Wu, and L.-P. Chen. On design of browser-oriented data extraction system and plug-ins. *J. of Marine Sc. and Tech.*, 2010.