Delft University of Technology
Software Engineering Research Group
Technical Report Series

Exposing the Hidden-Web Induced by Ajax

Ali Mesbah and Arie van Deursen

Report TUD-SERG-2008-001

TUD-SERG-2008-001

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Exposing the Hidden-Web Induced by Ajax

Ali Mesbah
Software Engineering Research Group
Delft University of Technology
The Netherlands
A.Mesbah@tudelft.nl

Arie van Deursen
Software Engineering Research Group
Delft University of Technology & CWI
The Netherlands
Arie.vanDeursen@tudelft.nl

ABSTRACT

AJAX is a very promising approach for improving rich interactivity and responsiveness of web applications. At the same time, AJAX techniques increase the totality of the hidden web by shattering the metaphor of a web 'page' upon which general search engines are based. This paper describes a technique for exposing the hidden web content behind AJAX by automatically creating a traditional multi-page instance. In particular we propose a method for crawling AJAX applications and building a state-flow graph modeling the various navigation paths and states within an AJAX application. This model is used to generate linked static HTML pages and a corresponding Sitemap. We present our tool called CRAWLJAX which implements the concepts discussed in this paper. Additionally, we present a case study in which we apply our approach to two AJAX applications and elaborate on the obtained results.

Categories and Subject Descriptors

H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia—*Navigation*; H.3.3 [**Information Search and Retrieval**]: Search process; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Algorithms, Experimentation.

Keywords

ajax, hidden web, crawling ajax, search engine accessibility, web engineering.

1. INTRODUCTION

The web as we know it is undergoing a significant change. A set of concrete technologies, under the umbrella of Rich Internet Applications (RIA) and *Web 2.0*, have made the web of today a lot more interactive and responsive for end users than it used to be a few years ago.

A technology that has gained a prominent position is the AJAX (Asynchronous JavaScript and XML) [9] approach, in which a clever combination of JavaScript and Document Object Model (DOM) manipulation, along with asynchronous server communication is used to achieve a high level of user interactivity. Highly visible examples include Google Maps, Google Documents, and the recent version of Yahoo! Mail.

With this new change in developing web applications comes a whole set of new challenges, mainly due to the fact that AJAX shatters the metaphor of a web 'page' upon which many web technologies are based. One of these challenges is the way AJAX increases the totality of the *hidden-web* [18].

General web search engines, such as Google and Yahoo!, cover only a portion of the web called the *publicly indexable web* which consists of the set of web pages reachable purely by following hypertext links, ignoring forms [2] and client-side scripting. The pages not reached this way are referred to as the *hidden web*, which is estimated to comprise several millions of pages [2]. With the wide adoption of AJAX techniques that we are witnessing today this figure will only increase.

Although there has been extensive research on finding and exposing the hidden-web behind forms [2, 5, 11, 17, 18], the hidden-web induced as a result of client-side scripting in general and AJAX in particular has gained very little attention so far.

Consequently, while AJAX techniques are very promising in terms of improving rich interactivity and responsiveness [15], AJAX sites themselves may very well end up in the hidden web. Thus, they will fail to meet the simple rule that determines the success or failure of any public web site: *"if you can't find it, it doesn't exist"*.

In this paper, we will be concerned with the question how a web engineer can expose his or her AJAX web application to general search engines. It is unlikely that in the near future search engines will change the way they crawl the web, due to the many challenges AJAX sites impose on search engines. Hence, the responsibility rests on the shoulders of web developers to make sure the AJAX applications they build are as accessible and discoverable by search engines as possible.

We propose to expose the essential parts of an AJAX application to the general search engines by creating a traditional multi-page instance. To that end, we propose a new type of crawler that can exercise client side code, and which can identify clickable elements (which may change with every click) within the browser's DOM dynamically. The crawler uses these to build up a *state-flow graph* modeling the various navigation paths within an AJAX application. This graph is subsequently used to generate a traditional multi-page mirror version of the original AJAX application, along with a *sitemap* informing search engines about the generated pages that are available for crawling.

The underlying ideas have been implemented in a tool called CRAWLJAX. We have applied CRAWLJAX to two AJAX applications, the results of which are discussed in this paper.

The primary application of our approach lies in helping web engineers exposing their AJAX sites to search engines. Moreover, we believe that the crawling techniques that are part of our solution have other applications, such as within search engines or for automatically exercising all user interface elements of an AJAX site for testing purposes.

The paper is structured as follows. We start out, in Section 2 by exploring the reasons AJAX induces hidden-web content and discuss the difficulties of crawling and indexing such applications. In Section 3, we present some of the existing techniques that can be used to make AJAX sites more accessible to search engines. In Section 4, we present the overall view of our proposed solution, followed by a detailed discussion of our new crawling techniques, the generation process, and the CRAWLJAX tool in Sections 5–7. In Section 8 the results of applying our methods to two AJAX applications are shown, after which Section 9 discusses the findings and open issues. We conclude with a brief survey of related work, a summary of our key contributions, and suggestions for future work.

2. AJAX HIDDEN-WEB INDUCTION

First of all, we take a closer look at why AJAX actually induces hidden-web content. AJAX has a number of properties which makes it extremely difficult for search engines to crawl such web applications.

Client-side Execution The common ground for all AJAX applications is a JavaScript engine which operates between the browser and the web server, and which acts as an extension to the browser. This engine typically deals with server communication and user interface rendering. This client engine enables us to create rich and responsive user interface behavior. Any search engine willing to approach such an application must have support for the execution of the scripting language. Equipping a general search crawler with the necessary environment complicates its design and implementation considerably. The major search giants such as Google[1] currently have little or no support for executing JavaScript due to scalability and security issues.

State Changes & Navigation Traditional web applications are based on the multi-page interface paradigm consisting of multiple (dynamically generated) unique pages each having a unique URL.

In AJAX applications, not every state change necessarily has an associated REST-based [7] URI [15]. Ultimately, an AJAX application could consist of a single-page [16] with a single URL. This characteristic makes it very difficult for a search engine to index and point to a specific state on an AJAX application. For crawlers, navigating through traditional multi-page web applications has been as easy as extracting and following the hypertext links on each page. In AJAX, hypertext links can be replaced by events which are handled by the client engine. Simply extracting and retrieving the internal hypertext links is not sufficient any longer to navigate the application.

Dynamic Representational Model Indexing traditional web applications consists of following links, retrieving and saving the HTML source-code of each page. The state changes in AJAX applications are dynamically represented through the run-time changes on the DOM. This implies that the source code in HTML does not represent the state anymore. Any search engine aimed at crawling and indexing such applications, will need to have access to this run-time dynamic representational model of the application.

Delta-communication AJAX applications rely on a delta-communication [15] style of interaction in which merely the state changes are exchanged asynchronously between the client and the server, as opposed to the full-page retrieval approach in traditional web applications. Just retrieving and indexing the delta state changes could have the side-effect of losing the context of the changes.

[1] `http://www.google.com/support/webmasters/bin/answer.py?answer=66355&query=cloaking`

Clickables Because of the very dynamic nature of AJAX and the way events (e.g., `onclick`) can be attached to DOM elements at run-time, it is not just the hypertext link element that forms the doorway to the next state. For instance, a `div` element could also have an `onclick` event attached to it so that it becomes *Clickable*. Finding these run-time clickables is another non-trivial task for a crawler.

3. DESIGN FOR DISCOVERABILITY

There are some techniques that assist in making a modern AJAX website more accessible and discoverable [4] by search engines. We briefly discuss a number of such techniques in this section before introducing our proposed solution.

3.1 Client-side Design

Graceful Degradation In web engineering terms, the concept behind *Graceful Degradation* [8] is to design and build for the latest and greatest user-agent and then add support for less capable devices, i.e., focus on the majority on the mainstream and add some support for outsiders. Graceful Degradation allows a web site to ‘step down’ in such a way as to provide a reduced level of service rather than failing completely. A well-known example is the menu bar generated by JavaScript which would normally be totally ignored by search engines. By using HTML list items with hypertext links inside a `noscript` tag, the site can degrade gracefully.

Progressive Enhancement The term *Progressive Enhancement* was first introduced by Steven Champeon[2] and has been used as the opposite side to Graceful Degradation. This technique aims for the lowest common denominator, i.e., a basic markup HTML document, and begins with a simple version of the web site, then adds enhancements and extra rich functionality for the more advanced user-agents using CSS and JavaScript. Because the basic content is more accessible to search engine crawlers, AJAX sites built with Progressive Enhancement methods can improve their discoverability by search engines.

Unobtrusive JavaScript Enhanced behavior and rich functionality through Progressive Enhancement is provided by unobtrusive, externally linked JavaScript known as *Unobtrusive JavaScript*.

The concept revolves around the separation of JavaScript functionality from the structure, content, and presentation layers. An unobtrusive script, similar to an external CSS, is silently ignored by user-agents that do not support it, but is applied by more capable devices.

Figure 2 shows different ways a news page can be opened. Links in lines 1 and 2 will simply be ignored by search engines where as in 3 and 4 they can simply follow the `href` link and index the news page. So by thinking about search engines in advance, AJAX developers can improve the accessibility of the pages.

The ultimate unobtrusive solution (line 4-6) is to register the necessary event handlers programmatically, rather than inline. This is commonly achieved by assigning a particular CSS selector, in this case `thenews`, to the elements which need to be acted upon by the script. Lines 8-10 show the jQuery[3] code responsible for attaching the required functionality to the `onClick` event handlers.

[2] `http://hesketh.com/publications/progressive_enhancement_paving_way_for_future.html`

[3] `http://jquery.com`

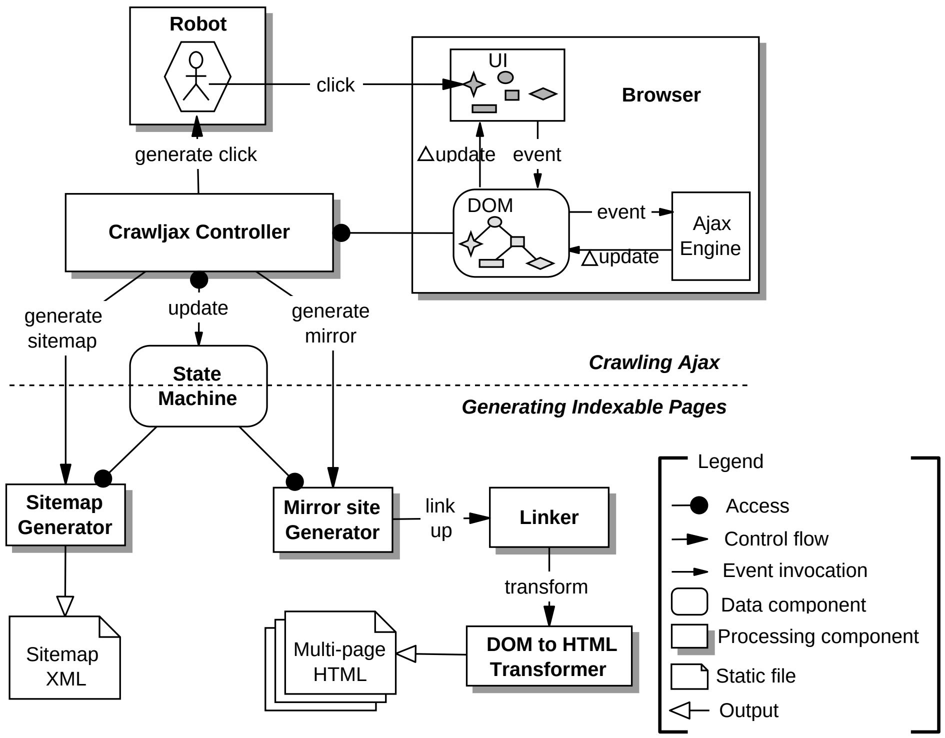

Figure 1: Processing view of the CRAWLJAX architecture.

```
<a href="javascript:OpenNewsPage();">
<a href="#" onClick="OpenNewsPage();">
<a href="news.html" onClick="OpenPage(this.href);">
<a href="news.html" class="thenews">
<input type="submit" class="thenews"/>
<div class="thenews">

$(".thenews").click(function() {
  $("#content").load("news.html");
});
```

Figure 2: Different ways of defining clickables in Ajax.

3.2 Server-side Generation

Another way to expose the hidden-web content behind AJAX applications is by making the content available to search engines at the server-side by providing it in an accessible style. The content could, for instance, be exposed through RSS feeds.

In the spirit of Progressive Enhancement, an approach called *Hijax*[4] involves building a traditional multi-page website first. Then, using unobtrusive event handlers, links and form submissions are intercepted and routed through the `XMLHttpRequest` object.

Generating and serving both the AJAX and the multi-page version depending on the visiting user-agent is yet another approach. One option is the use of XML/XSLT to generate indexable pages for search crawlers [1].

In these approaches, however, the server-side architecture will need to be quite modular, capable of returning delta changes as required by AJAX, as well as entire pages.

The server-side generation approach increases the complexity, development costs, and maintainability effort. In the next section we propose our solution, which aims at assisting the developer in the automatic generation of the indexable version of their AJAX application, thus significantly reducing the cost and effort of making AJAX sites more accessible to search engines.

[4] `http://www.domscripting.com/blog/display/41`

4. PROPOSED SOLUTION: POST-SITE GENERATION

In order to improve search engine discoverability for AJAX applications, we propose a post-site secondary site strategy, in which a linked multi-page mirror site is automatically generated after the AJAX application has been built. This mirror site is fully accessible to the search engines. In this approach, called CRAWLJAX, the input is an AJAX site already in place, with or without using the concepts as mentioned in Section 3, and the output is a traditional multi-page version of the application displaying the same content and structure. Our only requirement for crawling is that all *Clickable* (see 5.3.1 for a definition) elements should have unique IDs. The need for this constraint is explained in Section 5 and evaluated in our discussion Section 9.

Figure 1 depicts the processing view of our CRAWLJAX approach. As can be seen, the architecture can be divided in the following two parts:

Crawling AJAX: the main purpose of this step is to find and execute clickables and note the changes in the run-time DOM automatically, in a recursive way. A *State Machine* is used to record the navigational paths and state changes. This step supports three modes, namely *Full Auto Scan*, *DSL*, and *Annotations*, which can be used, respectively, to crawl automatically, to define the crawling navigational paths in a Domain Specific Language, and to define the elements to be taken into the crawling process by element annotations.

Generating Indexable Pages: the state machine with all the states filled in by the previous step is used to generate an index-

```
<html> <head> <title>News Ajax Site</title>
<link href="style.css" rel="stylesheet" type="text/css"/>
<script type="text/javascript" src="jquery.js"></script>
<script>
$(document).ready(function() {
  $(".remote").click(function(){
   $('#content').load('content.php?state=' + this.id);
  });
});

function changeState(id) {
  $('#content').load('content.php?state=' + id);
  return false;
}
</script> </head>

<body>
<div class="leftPan"> <h3>Menu</h3>
<ul>
 <li><a class="remote" href="#" id="headline">
     <b>Headlines</b></a> </li>
 <li><div class="remote" id="interview">
     Interviews</div> </li>
 <li><span onclick="changeState('technology');"
           id="technology1">Technology</span> </li>
</ul>
</div>
<div class="rightPan">
  <div id="content">
   <!-- This is where the content is loaded -->
  </div>
</div> </body> </html>
```

Figure 3: Source-code of a Single-page AJAX News Site.

able version of the AJAX application. This step is responsible for linking up and transforming the DOM instances into static HTML pages and generating a *Sitemap* for the generated HTML pages.

The details of these two main steps are explained in Section 5 and Section 6 respectively.

5. A METHOD FOR CRAWLING AJAX APPLICATIONS

In this section we discuss our approach for crawling AJAX in more detail. We use a simple single-page AJAX News site as shown in Figure 3 as example to explain the concepts. An example rendered view that can result from this HTML and JavaScript code is shown in Figure 4, which displays the view after having clicked the "Headlines" menu item. The difficulties of crawling AJAX applications were mentioned in Section 2 and this site is a typical example of how difficult it is for a general search engine to crawl and index such applications. Note how all the doorways to other states are dynamically set using JavaScript.

We apply *reverse engineering* techniques to deduce a state machine of the navigational model along the state changes of the AJAX application through a dynamic analysis of the run-time DOM changes.

As can be seen in Figure 3, even a `div` (such as the one with id '`interview`' in the second list item) can become clickable in AJAX by attaching an event to it. Detecting whether such an element is clickable by inspecting the code is very difficult due to the various ways events can be attached to DOM elements in AJAX. That is why we conduct a dynamic analysis for this purpose by actually running the application and trying to change its state.

5.1 The State-flow Graph

In traditional multi-page web applications, each state is represented by a URI. In AJAX however, it is the internal structure

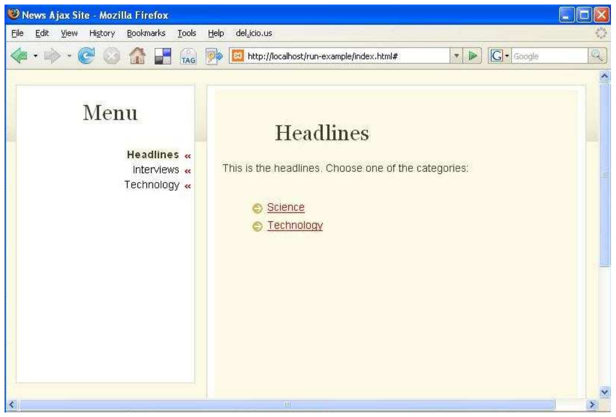
News Ajax Site - Mozilla Firefox
http://localhost/run-example/index.html#
Menu
Headlines
Interviews
Technology
Headlines
Science
Technology
This is the headlines. Choose one of the categories:

Figure 4: The News site after clicking on the 'headline' clickable.

change of the DOM tree on the single-page interface that represents a state change. Such internal state changes can be modeled by recording the paths to these DOM changes to be able to navigate the different states.

For that purpose we define a *state-flow graph* as follows:

DEFINITION 1. *A* ***state-flow graph*** *for an* AJAX *site* **A** *is a 3 tuple* $< \boldsymbol{r}, \boldsymbol{V}, \boldsymbol{E} >$ *where:*

1. $\boldsymbol{r}$ *is the root node (called Index) representing the initial state after* **A** *has been fully loaded into the browser.*

2. $\boldsymbol{V}$ *is a set of vertices representing the states. Each* $\boldsymbol{v} \in \boldsymbol{V}$ *represents a run-time state in* **A**.

3. $\boldsymbol{E}$ *is a set of edges between vertices. Each* $(\boldsymbol{v}_1, \boldsymbol{v}_2) \in \boldsymbol{E}$ *represents a clickable* $\boldsymbol{c}$ *connecting two states if and only if state* $\boldsymbol{v}_2$ *is reached by executing* $\boldsymbol{c}$ *in state* $\boldsymbol{v}_1$.

Our state-flow graph is similar to the *event-flow graph* [14], but different in that in the former vertices are *states*, where as in the latter vertices are *events*. Note that ultimately, it is the state changes that we need in order to generate static HTML pages.

As an example of a state-flow graph, Figure 5 depicts the state-flow graph of our News site. It illustrates how from the start page 3 different states can be reached. Furthermore, clicking on the Index menu item leads to the headline state, from which two states are reachable – the Science and Technology headlines also visible in the main pane in Figure 4.

The state-flow graph is created incrementally as the nodes are clicked. Initially, it only contains the root state while new states are dynamically created.

5.2 Crawling Components

The Crawling AJAX process, as shown in Figure 1, is based on the following components:

Embedded Browser: CRAWLJAX utilizes an embedded browser capable of executing JavaScript and the supporting technologies required by AJAX (e.g., DOM, `XMLHttpRequest`).

Robot: Whilst artificial events can be programmatically triggered on the DOM document tree (e.g., using `element.fireEvent`), only the listeners will be dispatched: Actions associated with the event will not be performed due to security issues. Hence, we use a Robot to simulate real user clicks and inputs on the embedded browser to fire possible events and actions attached to candidate clickables.

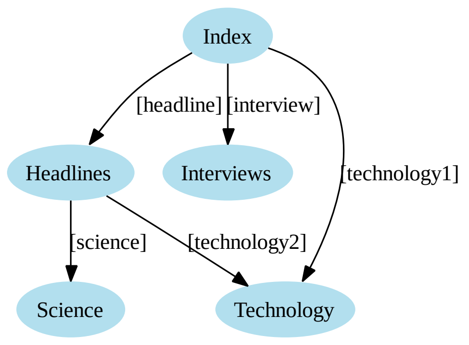

Figure 5: The State-flow Graph.

Based on an analysis of the DOM tree we will identify candidate clickable elements (see Section 5.3.1). The `<client.x, client.y>` screen coordinates of the such elements are used to move the Robot's pointer to the element's location.

CRAWLJAX Controller: The controller has access to the embedded browser's DOM and controls the Robot's actions. It is also responsible for updating the State Machine when relevant changes occur in the DOM. After the crawling process is over, the controller also calls the Sitemap and Mirror site generator processes.

State Machine: The state machine is a data component maintaining the state-flow graph, as well as a pointer to the current state.

As mentioned before, CRAWLJAX adopts three different modes to crawl an AJAX site: Full Auto Scan, Annotations, and DSL. We present each mode in the subsequent sub-sections.

5.3 Full Auto Scan

In the Full Auto Scan mode, CRAWLJAX crawls the site automatically by finding all possible clickables and executing them. Algorithm 1 shows the Full Auto Scan algorithm.

5.3.1 *Finding Clickables*

This mode expects a list of HTML tag element names (e.g., `div`, `a`, `span`, `input`) and the URL of the AJAX site to begin with. Elements having such tag names are considered *candidate clickables*. After the required environment is initialized, the recursive (depth-first) `crawl` procedure is called. For each tag name the present elements on the DOM are added to the candidate clickable list (line 13).

In order to find out whether a certain element in the candidate clickable list, is indeed clickable, the crawler instructs the robot to execute a click (line 15) on the element in the browser.

5.3.2 *Comparing DOM Trees: Edit Distance*

In order to determine if a click results in a new state, the DOM before and after a click is compared. For this purpose the *edit distance* between two DOM trees is calculated (line 17) using the Levenshtein [12] method. A similarity threshold τ is used under which two DOM trees are considered clones. This threshold (0.0 – 1.0) can be defined by the developer. A threshold of 0 means two DOM states are seen as clones if they are *exactly* the same in terms of structure and content. Any change is, therefore, seen as a state change.

If a change is detected, we add a new state to the state-flow diagram of the state machine (lines 18-19). The current state pointer of the state machine is also changed to this newly added state at that moment (line 20).

```
...
<div id="content">
<!-- content of headline has been loaded -->
 <p><h2>Headlines</h2>
  This is the headlines. Choose one of the categories:
 </p>
 <ul>
  <li><div onclick="changeState('science');"
      id="science">Scince</div> </li>
  <li><a href="#" onclick="changeState('technology');"
      id="technology2">Technology</a> </li>
  </ul>
</div>
...
```

Figure 6: The DOM state after clicking on the 'headline' element.

Looking at our example site, our algorithm detects the three elements with IDs `headline`, `interview`, and `technology1` as clickables, since clicking on any of them causes the DOM to change.

Algorithm 1 CRAWLJAX Full Scan

```
 1: procedure START (url, Set tags)
 2: browser ← initBrowser(url)
 3: robot ← initRobot()
 4: sm ← initStateMachine()
 5: crawl(sm, tags)
 6: linkupAndSaveAsHTML(sm)
 7: generateSitemap(sm)
 8: end procedure
 9:
10: procedure CRAWL (StateMachine sm, Set tags)
11: cs ← sm.getCurrentState()
12: Δupdate ← diff(cs.getDom(), browser.getDom())
13: Set C ← getCandidateClickables(Δupdate, tags)
14: for c ∈ C do
15:     robot.click(c)
16:     dom ← browser.getDom()
17:     if distance(cs.getDom(), dom) > τ then
18:         ns ← State(c, dom)
19:         sm.addState(ns)
20:         sm.changeState(ns)
21:         crawl(sm, tags)
22:         sm.changeState(cs)
23:         if browser.history.canBack then
24:             browser.history.goBack()
25:         else
26:             browser.reload()
27:             clickThroughTo(cs)
28:         end if
29:     end if
30: end for
31: end procedure
```

5.3.3 *Delta Updates*

After a clickable has been identified, the `crawl` procedure is recursively called to find new candidate clickables and eventually clickables in the delta updates (line 13) of the document tree after each state change. The delta update changes are detected through a *Diff* [3, 16] algorithm (line 12) by comparing the DOM tree before and after executing a clickable.

```
crawl MyAjaxSite {
  url: http://localhost/run-example/index.html;

  navigate Nav1 {
    click: headline;
    click: science;
    ...
  }

  navigate Nav2 {
    click: headline;
    click: technology2;
    ...
  }

  navigate Nav3 {
    click: interview;
    input: article "john doe";
    click: search;
  }
  ...
}
```

Figure 7: An instance of CASL.

Figure 6 shows the DOM state after '`headline`' has been clicked. Clicking on `headline` loads the corresponding content into the `div` element with ID `content`. This new content is seen as a *delta update* so Crawljax looks for candidate clickables in there and finds the `science` and `technology2` as clickables in the same way.

5.3.4 *Navigating the States*

As already mentioned, navigating (back and forth) through an Ajax site is not as easy as navigating a classical web site. A dynamically created DOM state does not register itself with the browser history engine automatically, so triggering the 'Back' function of the browser might not bring us to the previous state. This complicates traversing the application when crawling Ajax.

Browser History Support It is possible to register each state change with the browser history through frameworks such as the jQuery history/remote plugin[5] or the Really Simple History library[6].

If an Ajax application has support for the browser history, then for changing the state in the browser, Crawljax simply uses the built-in history back functionality to move back. For instance, if Crawljax's browser is on the `Science` state, it needs to go back to the `Headlines` state to be able to click on the `technology2` clickable to end up in the `Technology` state. If our News site has support for history, then going to state `Headlines` is as simple as calling the browser back method (lines 23-24).

Click Through From Initial State In case the browser history is not supported (which is the case with many Ajax applications currently), the only way to get to a previous state is by reloading the initial page and following the path of clickables from the initial state to the desired state (lines 26-27).

This is also one of the main reasons behind our requirement that clickables should have IDs. When we `reload` the application in the browser, all the internal objects are replaced by new ones and the ID attribute is a means to be sure we can follow the path to a certain state by clicking on those saved IDs in the state machine.

Note that because of side effects of the clicks, there is no guarantee that we reach the exact same state when we traverse an ID-path a second time. It is, however, as close as we can get.

[5] `http://stilbuero.de/jquery/history/`
[6] `http://code.google.com/p/reallysimplehistory/`

5.3.5 *Identifying Clone States*

Our example shows that the `Technology` state can be navigated to either directly from the `Index` state, or through the `Headlines` state. In order to recognize an already met state, we compute a *hashcode* for each DOM state and use the hashcodes to compare every new state to the list of already visited states on the state-flow graph. This way we can easily identify clone states and avoid creating unnecessary duplicated ones in our state machine.

It is worth mentioning that in order to avoid a loop, a list of visited candidate clickables is maintained to exclude already checked elements in the recursive algorithm if needed. Also a depth length can be defined to constrain the depth level of the recursive function (not shown in the algorithm).

5.4 Annotations

There are situations in which a Full Auto Scan that takes every clickable and every state change on the DOM into account is not desirable. Perhaps only parts of an Ajax site are relevant to be exposed to search engines.

For that reason, we believe the developer should also be given the opportunity to define which parts of their application they want to be crawled and indexed. One way to do that is through *annotating* the source-code by setting the attribute `crawljax="true"` on the clickables.

Crawljax automatically finds all the annotated elements and adds only those to the list of candidate clickables. The rest of the process is the same as the Full Auto Scan process.

5.5 CASL

In addition to the Annotations, we provide the developer with a Domain Specific Language (DSL) [6] called Ajax Crawling Specification Language (CASL). Using CASL, the developer can define the elements (based on IDs) to be clicked, along with the exact order in which the crawler should crawl and index the Ajax application. CASL has two commands basically: `click` and `input`.

Figure 7 shows the CASL instance for our example application. `Nav1` tells Crawljax to crawl and index the states generated by clicking on `headline` and `science` in that order. `Nav3` commands the crawler to crawl to the `Interviews` state, then insert the text '`john doe`' into the input element `article` and afterward click on the `search` element and index the resulting states.

6. GENERATING INDEXABLE PAGES

After the crawling Ajax process is finished, the created state-flow graph is passed to the *Mirror Site Generation* and *Sitemap Generation* processing components.

6.1 Mirror Site Generation

6.1.1 *Linking the States*

To enable a general search engine to find all the generated states, we first establish links for the DOM states in the state-flow graph. We do so by examining the element type of the clickables. If the clickable is a hypertext link (an `a`-element), the `href` attribute is updated. In case of other types of clickables (e.g., `div, span`) we replace the clickable by a hypertext link element. The `href` attribute in both situations represents the link to the name and location of the to be generated static page.

6.1.2 *Transforming DOM to HTML*

After the linking process, each DOM object in the state-flow graph is transformed into the corresponding HTML string repre-

```
<html> <head> <title>News Ajax Site</title>
<link href="style.css" rel="stylesheet" type="text/css"/>
</head>
<body>
<div class="leftPan"> <h3>Menu</h3>
<ul>
 <li><a href="/generated/headline.html" class="remote"
     id="headline"><b>Headlines</b></a></li>
 <li><a href="/generated/interview.html" class="remote"
     id="interview">Interviews</a>
 <li><a href="/generated/technology1.html"
     id="technology1">Technology</a>
 </li> </ul> </div>

<div class="rightPan">
  <div id="content">
    <p><h2>Headlines</h2>
     This is the headline. Choose one of the categories:
    </p>
    <ul>
    <li><a href="/generated/science.html"
           id="science">Scince</a> </li>
    <li><a href="/generated/technology2.html"
           id="technology2">Technology</a> </li>
    </ul>
  </div> </div> </body> </html>
```

Figure 8: The static 'headline' page generated by CRAWLJAX.

sentation and saved on the file system in a dedicated directory (e.g., /generated/). Each generated static file represents the style, structure, and content of the AJAX application as seen in the browser, in exactly its specific state at the time of crawling.

Figure 8 shows the generated HTML file for the 'headlines' state. Note how the various AJAX clickables e.g., science, from Figure 6 are turned into traditional hypertext links, accessible by search engines.

6.1.3 Deploying the Mirror Site

Next, the generated pages have to be uploaded to the server. For the mirror site to look exactly like the AJAX version, care must be taken so that internal links, to for instance CSS files and images, are not broken.

6.1.4 Linking to the AJAX Site & Vice Versa

The original AJAX site can link to the mirror site to form the first doorway for search engines. There are also possible ways of linking the mirror site pages to the original state in the AJAX application. The simplest approach is to link to the original state of the AJAX site. This means that the users themselves then have to find their way to the specific state of the static page.

Another, more elegant, solution involves allowing the user to jump to that very specific state on the AJAX site. This requires the AJAX application to implement and support browser bookmarking for each state. Solutions exist, many of which use the URL fragment identifier [15] to keep track of, and allow users to return to the application in a given state. The user-agent property of the visiting agent could be used [1] to redirect a web user to the corresponding AJAX state in this case. Although the content returned to the user (AJAX) and the search engine (generated HTML pages) is exactly the same, care must be taken to avoid *cloaking*[7] possibilities.

6.2 Sitemap Creation

The Sitemap, initially proposed by Google, is a static XML file that allows a web developer to inform search engines about URLs

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset
 xmlns:ns="http://www.sitemaps.org/schemas/sitemap/0.9">
<url>
 <loc>
 http://localhost/run-example/generated/index.html
 </loc>
 <lastmod>2007-10-29</lastmod>
 <changefreq>weekly</changefreq>
</url>
<url>
 <loc>
 http://localhost/run-example/generated/headline.html
 </loc>
 <lastmod>2007-10-29</lastmod>
 <changefreq>weekly</changefreq>
</url>
<url>
 <loc>
 http://localhost/run-example/generated/technology2.html
 </loc>
 <lastmod>2007-10-29</lastmod>
 <changefreq>weekly</changefreq>
</url>
<url>
 <loc>
 http://localhost/run-example/generated/science.html
 </loc>
 <lastmod>2007-10-29</lastmod>
 <changefreq>weekly</changefreq>
</url>
<url>
 <loc>
 http://localhost/run-example/generated/interview.html
 </loc>
 <lastmod>2007-10-29</lastmod>
 <changefreq>weekly</changefreq>
</url>
</urlset>
```

Figure 9: Generated Sitemap XML file.

on a website that are available for crawling. A Sitemap file consists of one or more URLs and a number of optional descriptors of the URL, such as the estimated change rate, date of last modification and a local crawling priority. Google, Yahoo!, and Microsoft have announced[8] auto-discovery and support for the protocol.

CRAWLJAX adheres to Sitemap Protocol 0.9[9], and generates a valid instance of the protocol automatically after each crawling session consisting of the URLs of all generated static pages.

Figure 9 presents the generated Sitemap file for the News site. For each state in the state-flow graph, an URL entry is created with the location, last modification date, and change frequency. This way, general search engines can be notified of the generated static files in a standard way.

7. TOOL IMPLEMENTATION

We have implemented the concepts presented in this paper in a tool called CRAWLJAX. At the moment the tool is available on request. More information about the tool and conducted case studies can be found on our website[10].

CRAWLJAX is implemented in Java 5. We have engineered a variety of software and web tools to build and run CRAWLJAX. Here we briefly mention the main modules and libraries.

The embedded browser is implemented using Mozilla XULRunner[11]. Webclient[12] is used to access the run-time DOM and the

[7] http://www.google.com/support/webmasters/bin/answer.py?answer=66355&query=cloaking

[8] http://www.google.com/press/pressrel/sitemapsorg.html

[9] http://www.sitemaps.org/protocol.php

[10] http://swerl.tudelft.nl/bin/view/Main/Crawljax/

[11] http://developer.mozilla.org/en/docs/XULRunner/

[12] http://www.mozilla.org/projects/blackwood/webclient/

	AOWE	PETSTORE
DOM string size (byte)	5226	24636
Expected Clickables	16	34
Candidate Clickables	25	36
Clickables	17	28
Generated Static Pages	16	28
Generated Sitmap URLs	16	28
Crawl Performance (ms)	55031	119264
Generation Performance (ms)	31859	65531
DOM Pretty-print (ms)	3965	12046

Table 1: Results of running CRAWLJAX on AOWE and PETSTORE.

browser history mechanism in the embedded browser. The Robot component makes use of the `java.awt.Robot` class to generate native system input events on the embedded browser.

The Mirror Site Generator uses JTidy[13] to pretty-print DOM states and Xcerces[14] to serialize the objects to HTML. In the Sitemap Generator, XMLBeans[15] generates Java objects from the Sitemap Schema[16] which after being used by CRAWLJAX to create new URL entries, are serialized to the corresponding valid XML instance document.

The grammar of CASL is implemented in ANTLR[17]. ANTLR is used to generate the necessary parsers for CASL. In addition, StringTemplate[18] is used for generating the source-code from CASL.

CRAWLJAX is entirely based on Maven[19] to generate, compile, test (JUnit), and run the application. Log4j is used to optionally log various steps in the crawling process, such as the identification of DOM changes and clickables.

8. CASE STUDY

We have performed a case study set up according to Kitchenham's guidelines [10] to evaluate the application of our framework over two representative AJAX sites. Our goals include (1) analyzing the overall performance of our approach, (2) evaluating the effectiveness of CRAWLJAX in obtaining high-quality results in retrieving relevant clickables, and (3) assessing the quality of the static pages automatically generated by CRAWLJAX.

Because of the very dynamic nature of AJAX applications, and since other comparable tools are not available to conduct similar methods as CRAWLJAX carries out currently, it is difficult to define a baseline against which we can compare the results. Hence, we manually inspect the objects under examination and determine which expected behavior should form our reference baseline.

Case study results including generated sites and CRAWLJAX log files are made available through the CRAWLJAX web site.

8.1 Case Objects

We have selected two AJAX sites for the experiment, the first one (AOWE) developed internally by our group and the second (PETSTORE) is an external open-source web application.

AOWE Ajax Site

The AOWE AJAX site has been implemented using the jQuery AJAX library. Although the site is small, it is representative by having different types of dynamically set clickables as shown in Figure 2.

For the case study we manually added extra clickables in different states of the application, especially in the delta updates, to explore whether deep clickables dynamically injected into the DOM can be found by CRAWLJAX. The site was deployed on our local server and a reference model was created manually by clicking through the different states in a browser. In total 16 clickables were noted of which 10 were on the top level, i.e., index state. The clickable elements were of the types `a`, `div`, `span`, and `input`. All clickables in this application have unique IDs.

Ajaxified Sun PETSTORE

Our second case object is Sun's Ajaxified PETSTORE 2.0[20] which is built on the Java ServerFaces, and the Dojo AJAX toolkit[21]. This open-source web application is designed to illustrate how the Java EE Platform can be used to develop an AJAX-enabled Web 2.0 application and adopts many advanced rich AJAX components.

To constrain the reference model we chose two product categories, namely CATS and DOGS, from the five available categories. Manual inspection of the application revealed that although most elements had IDs, the IDs used were not always unique. The IDs on elements were also set using the IDs of the items in the database. Therefore, we first made all the IDs unique through the available SQL insert statements of the application for the two chosen categories and afterwards annotated all the relevant product items by modifying a JavaScript method which turns the items retrieved from the server into clickables on the interface.

It is worth mentioning that although we were not familiar with the application the modification was carried out in $+-$ 20 minutes, most of which was spent on finding out *where* the modifications should take place.

8.2 Tool Configuration

Configuring CRAWLJAX itself is done through the Maven Project Object Model (POM). Through the POM, the URL of the site to be analyzed, and the tag elements CRAWLJAX should look for can be set. For the similarity threshold we defined τ as 0, i.e., every single change in DOM is seen as a change. The depth level was set to 4.

8.3 Results

Table 1 presents the results obtained by running CRAWLJAX on AOWE and PETSTORE.

The number of candidate clickables and actual identified clickables were read from the log file at the end of each crawling process. After the generation process, the number of generated HTML files and their content were manually examined to see whether the pages were the same as the corresponding states in AJAX in terms of structure, style, and content. Also the internal linking of the static pages was checked. In addition, the URL entries in the generated Sitemap XML file were examined.

The execution time for the crawling and generation processes were computed separately. The *Crawling Performance* represents the time in milliseconds taken by CRAWLJAX to find the clickables and build the state-flow graph, excluding the time needed to pretty-print the DOM into string. The *Generation Performance* shows the period taken to generate the static HTML pages and the Sitemap from the state machine. The *DOM Pretty-print* indicates the time required to transform a DOM object into the corresponding HTML string representation.

[13] http://jtidy.sourceforge.ne
[14] http://xerces.apache.org/xerces-j/
[15] http://xmlbeans.apache.org
[16] http://www.sitemaps.org/schemas/sitemap/0.9/sitemap.xsd
[17] http://www.antlr.org
[18] http://www.stringtemplate.org
[19] http://maven.apache.org
[20] http://java.sun.com/developer/releases/petstore/
[21] http://dojotoolkit.org/

8.4 Observations

As can be seen in Table 1, CRAWLJAX finds 17 clickables on AOWE instead of the expected 16. After closer inspection, we noticed that the extra false clickable is caused by the following code pattern: `<span id="y"><div id="x">text</div></span>`. In this case the `span` element is the actual clickable, however, since the `div` element is inside the `span`, it can also be seen as a clickable. Since clicking on any of the two results in the same state, we see that the actual expected 16 HTML pages and Sitemap entries were correctly generated.

Mouseover-dependent Clickables For PETSTORE the scenario is more complicated. From the 34 annotated clickables, CRAWLJAX was able to find only 28. The reason behind this difference is the way the items are shown to the user. PETSTORE uses a `Catalog Browser` to show a set of the total number of the product items and defines an `onMouse` event on an `img` element to browse through the other items one by one. For our robot this means a two step action. First CRAWLJAX has to know about the `onMouse` behavior and move the mouse pointer to the `img` element, after which a new clickable appears, and then that new element has to be clicked. The 6 missing product items were the ones that would be shown in that manner.

Constantly Updating DOM Another issue we had with PETSTORE in the beginning of the experiment was that all the 36 candidate clickables found were also seen as clickables. This phenomenon was caused by the `banner.js` which constantly changed the DOM with textual notifications. Hence, we had to either disable this banner to conduct our experiment or use a higher similarity threshold so that the textual changes were not seen as a relevant state change for detecting clickables.

History Back Implementation CRAWLJAX assumes that if the Browser Back functionality is implemented, then it is implemented correctly. Yet another interesting observation with PETSTORE was the fact that even though Back is implemented for some states, it is not correctly implemented in the sense that calling the Back method brings the browser in a different state than expected which naturally confuses CRAWLJAX. AOWE implements the Back method correctly.

Performance It takes CRAWLJAX 55031 ms to crawl AOWE and 119264 ms to crawl PETSTORE. As can be seen, the DOM in PETSTORE is 4 times bigger than that in AOWE which also explains the higher execution time for the DOM Pretty-print. There are also 11 more clickables in PETSTORE. In addition to the increase in DOM size and the number of clickables, CRAWLJAX cannot rely on the browser Back method when crawling PETSTORE. This means for every state change on the browser CRAWLJAX has to reload the application and click through to the previous state to go further. This reloading and clicking through has a negative effect on the performance. The generation time also doubles for PETSTORE due to the increase in the input size.

9. DISCUSSION

This section discusses a number of important characteristics of our techniques and discusses both the strengths and open issues.

9.1 Evaluation

As revealed in the case study, CRAWLJAX can find and crawl deep clickables correctly. Also the generated HTML pages are correct and represent exactly the corresponding state in the AJAX version. The static pages are correctly linked and the Sitemap is generated as expected. The weakness seems to be finding clickables that appear through complex AJAX widgets which require the user to have an understanding of the application. The `Catalog Browser` for instance in PETSTORE is an example. The user must understand from the context and shape of the state that hovering on an image will allow them to browse the catalog and see more items. Currently, we are exploring how such patterns could be detected and the corresponding clickables executed automatically.

9.2 Performance

It is clear that the running time of CRAWLJAX increases linearly with the size of the input. The tool is intended to be used internally by web developers. Therefore, we believe that although the performance could be improved, the execution time of a few minutes to generate a mirror multi-page instance of an AJAX application automatically without any human intervention is acceptable.

9.3 Combining the Crawling Modes

When it comes to states that need textual input from the user (e.g., input forms) CASL can be very helpful to crawl and generate the corresponding state. The Full Auto Scan, however, does not have the knowledge to provide such input automatically. Therefore, we believe a combination of the three modes to take the best of each could provide us with a powerful tool not only for crawling but also for automatic testing of AJAX applications.

9.4 ID Requirement

As far as the ID requirement is concerned, if browser Back is correctly implemented by an AJAX site, the requirement could fall altogether. Since no reloading of the site is needed to navigate the state-flow graph when Back is implemented, persistent IDs could be set by CRAWLJAX which has access to the run-time DOM.

Currently, we are also investigating the possibilities of utilizing XPath to find and record the location of clickables in the DOM instead of using unique IDs to identify elements persistently.

9.5 Incomplete set of HTML pages

The set of generated HTML pages is by no means complete, i.e., CRAWLJAX generates an static instance of the AJAX application but not necessarily *the* instance. This is partly inherent to dynamic web applications. Any crawler can only index an instance of a dynamic web application in a point in time. The order in which clickables are chosen could generate different states. Even executing the same clickable twice from an state could theoretically produce two different DOM states depending on, for instance, server-side factors. Hence, CRAWLJAX crawls and generates an instance of the web application at a certain point in time.

10. RELATED WORK

There has been extensive research on finding and exposing the hidden-web behind forms [2, 5, 11, 17, 18]. On the contrary, the hidden-web induced as a result of client-side scripting in general and AJAX in particular has gained very little attention so far. As far as we know, there are no academic research papers on crawling and exposing the hidden-web AJAX at the moment.

There are, however, some industrial proposed approaches for improving the discoverability of AJAX as discussed in Section 3.

The concept behind CRAWLJAX, is the opposite direction of our earlier work RETJAX [16], in which we try to reverse-engineer a traditional multi-page website to AJAX.

Shelly and Young [19] discuss the possible ways of improving the accessibility for DHTML websites. CRAWLJAX also improves accessibility towards user-agents that do not support JavaScript by creating the multi-page instance.

The work of Memon *et al.* [13, 14] on GUI Ripping for testing purposes is related to our work in terms of how they reverse engineer an event-flow graph of desktop GUI applications by applying dynamic analysis techniques.

11. CONCLUDING REMARKS

In this paper, we have studied how AJAX induces hidden-web content and explored ways of improving the discoverability of such applications. In particular, we have proposed a method to crawl AJAX applications by automatically detecting and executing clickables and building a state-flow graph representation of the run-time paths and states. Besides the Full Auto Scan mode, we provide the developer with two alternatives: Annotations and a DSL called CASL, to control the way the site is crawled. We have discussed how such a graph can be used to generate a traditional multi-page instance of the original application, fully accessible to the search engines. This mirror site also improves the accessibility of the application towards user-agents that do not support JavaScript.

In summary, this paper makes the following contributions:

1. An approach to increase the discoverability of hidden-web content induced by AJAX.
2. A novel method to automatically crawl AJAX applications and build a state-flow graph model of the states and paths.
3. A technique to transform the run-time DOM state changes of AJAX applications into static HTML pages and generate a corresponding Sitemap.
4. The tool CRAWLJAX implementing the methods and concepts discussed in this paper.
5. A case study report covering the application of our approach to two AJAX applications.

Future work consists of conducting more case studies to improve the ability of finding clickables in different AJAX applications. Strengthening the tool by extending its functionalities and improving the performance is another direction we foresee. We will also explore possibilities of dropping the ID requirement by adopting alternatives such as XPath.

12. REFERENCES

[1] Backbase. Designing rich internet applications for search engine accessibility, 2005. backbase.com Whitepaper.

[2] L. Barbosa and J. Freire. An adaptive crawler for locating hidden-web entry points. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 441–450. ACM Press, 2007.

[3] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504. ACM Press, 1996.

[4] A. Dasgupta, A. Ghosh, R. Kumar, C. Olston, S. Pandey, and A. Tomkins. The discoverability of the web. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 421–430. ACM Press, 2007.

[5] A. F. de Carvalho and F. S. Silva. Smartcrawl: a new strategy for the exploration of the hidden web. In *WIDM '04: Proceedings of the 6th annual ACM international workshop on Web information and data management*, pages 9–15. ACM Press, 2004.

[6] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[7] R. Fielding and R. N. Taylor. Principled design of the modern Web architecture. *ACM Trans. Inter. Tech. (TOIT)*, 2(2):115–150, 2002.

[8] M. Florins and J. Vanderdonckt. Graceful degradation of user interfaces as a design method for multiplatform systems. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*, pages 140–147. ACM Press, 2004.

[9] J. Garrett. Ajax: A new approach to web applications. Adaptive path, 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

[10] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Softw.*, 12(4):52–62, 1995.

[11] J. P. Lage, A. S. da Silva, P. B. Golgher, and A. H. F. Laender. Automatic generation of agents for collecting hidden web pages for data extraction. *Data Knowl. Eng.*, 49(2):177–196, 2004.

[12] V. L. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10:707–710, 1996.

[13] A. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE '03: 10th Working Conference on Reverse Engineering*, pages 260–269. IEEE Computer Society, 2003.

[14] A. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *ESEC/FSE '01: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 256–267, New York, NY, USA, 2001. ACM Press.

[15] A. Mesbah and A. van Deursen. An architectural style for Ajax. In *WICSA '07: Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture*, pages 44–53. IEEE Computer Society, 2007.

[16] A. Mesbah and A. van Deursen. Migrating multi-page web applications to single-page Ajax interfaces. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 181–190. IEEE Computer Society, 2007.

[17] A. Ntoulas, P. Zerfos, and J. Cho. Downloading textual hidden web content through keyword queries. In *JCDL '05: Proceedings of the 5th ACM/IEEE-CS joint conference on Digital libraries*, pages 100–109. ACM Press, 2005.

[18] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 129–138. Morgan Kaufmann Publishers Inc., 2001.

[19] C. C. Shelly and G. Young. Accessibility for simple to moderate-complexity DHTML web sites. In *W4A '07: Proceedings of the 2007 international cross-disciplinary conference on Web accessibility*, pages 65–73. ACM Press, 2007.

TUD-SERG-2008-001
ISSN 1872-5392