Crawling AJAX by Inferring User Interface State Changes

Ali Mesbah
Delft University of Technology
The Netherlands
A.Mesbah@tudelft.nl

Engin Bozdag
Delft University of Technology
The Netherlands
V.e.Bozdag@tudelft.nl

Arie van Deursen
Delft Univ. 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 shatter the metaphor of a web 'page' upon which general search crawlers are based. This paper describes a novel technique for crawling* AJAX *applications through dynamic analysis and reconstruction of user interface state changes. Our method dynamically infers a 'state-flow graph' modeling the various navigation paths and states within an* AJAX *application. This reconstructed model can be used to generate linked static pages. These pages could be used to expose* AJAX *sites to general search engines. Moreover, we believe that the crawling techniques that are part of our solution have other applications, such as within general search engines, accessibility improvements, or in automatically exercising all user interface elements and conducting state-based testing of* AJAX *applications. We present our open source tool called* CRAWLJAX *which implements the concepts discussed in this paper. Additionally, we report a case study in which we apply our approach to a number of representative* AJAX *applications and elaborate on the obtained results.*

1 Introduction

The web as we know it is undergoing a significant change. A technology that has gained a prominent position lately, under the umbrella of *Web 2.0*, is AJAX (Asynchronous JavaScript and XML) [13], 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. Among these challenges are the following:

Searchability ensuring that AJAX sites are indexed by the general search engines, instead of (as is currently often the case) being ignored by them because of the use of client-side scripting and dynamic state changes in the DOM;

Testability systematically exercising dynamic user interface (UI) elements and states of AJAX to find abnormalities and errors;

Accessibility examining whether all states of an AJAX site meet certain accessibility requirements.

One way to address these challenges is through the use of a crawler that can automatically walk through different states of a highly dynamic AJAX site, create a model of the navigational paths and states, and generate a traditional linked page-based static version. The generated static pages can be used, for instance, to expose AJAX sites to general search engines or to examine the accessibility [2] of different dynamic states. Such a crawler can also be used for conducting state-based testing of AJAX applications [16] and automatically exercising all user interface elements of an AJAX site in order to find e.g., link-coverage, broken-links, and other errors.

To date, no crawler exists that can handle the complex client code that is present in AJAX applications. The reason for this is that crawling AJAX is fundamentally more difficult than crawling classical multi-page web applications. In traditional web applications, states are explicit, and correspond to pages that have a unique URL assigned to them. In AJAX applications, however, the state of the user interface is determined dynamically, through changes in the DOM that are only visible after executing the corresponding JavaScript code.

In this paper, we propose an approach to analyze and reconstruct these user interface states automatically. Our approach is based on a crawler that can exercise client side code, and can identify clickable elements (which may change with every click) that change the state within the browser's dynamically built DOM. From these state

978-0-7695-3261-5/08 $25.00 © 2008 IEEE
DOI 10.1109/ICWE.2008.24

changes, we infer a *state-flow graph*, which captures the states of the user interface, and the possible transitions between them. This graph can subsequently be used to generate a multi-page static version of the original AJAX application.

The underlying ideas have been implemented in a tool called CRAWLJAX.[1]

We have performed an experiment of running our crawling framework over a number of representative AJAX sites to analyze the overall performance of our approach, evaluate the effectiveness in retrieving relevant clickables, assess the quality and correctness of the detected states and generated static pages, and examine the capability of our tool on real sites used in practice and the scalability in crawling sites with thousands of dynamic states and clickables. The cases span from internal to academic and external commercial AJAX web sites.

The paper is structured as follows. We start out, in Section 2 by exploring the difficulties of crawling and indexing AJAX. In Sections 3 and 4, we present a detailed discussion of our new crawling techniques, the generation process, and the CRAWLJAX tool. In Section 5 the results of applying our methods to a number of AJAX applications are shown, after which Section 6 discusses the findings and open issues. Section 7 presents various applications of our crawling techniques. We conclude with a brief survey of related work, a summary of our key contributions, and suggestions for future work.

2 Challenges of Crawling AJAX

AJAX has a number of properties making it extremely difficult for, e.g., search engines to crawl such web applications.

2.1 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. 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[2] currently have little or no support for executing JavaScript due to scalability and security issues.

[1]The tool is available for download from http://spci.st.ewi.tudelft.nl/crawljax/.

[2] http://googlewebmastercentral.blogspot.com/2007/11/spiders-view-of-web-20.html

```
1  <a href="javascript:OpenNewsPage();">
2  <a href="#" onClick="OpenNewsPage();">
3  <div onClick="OpenNewsPage();">
4  <a href="news.html" class="news">
5  <input type="submit" class="news"/>
6  <div class="news">
7  <!-- jQuery function attaching events to elements
8      having attribute class="news" -->
9  $(".news").click(function() {
10   $("#content").load("news.html");
11 });
```

Figure 1. Different ways of attaching events to elements.

2.2 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 [11] URI [20]. Ultimately, an AJAX application could consist of a single-page [19] 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 (or the `src` attribute) on each page. In AJAX, hypertext links can be replaced by events which are handled by the client engine; it is not possible any longer to navigate the application by simply extracting and retrieving the internal hypertext links.

2.3 Dynamic Document Object Model (DOM)

Crawling and 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 means 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 document object model of the application.

2.4 Delta-communication

AJAX applications rely on a delta-communication [20] 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. Retrieving and indexing the delta state changes, for instance, through a proxy between the client and the server, could have the side-effect of losing the context and actual meaning of the changes. Most of such delta updates become meaningful after they have been processed

by the JavaScript engine on the client and injected into the DOM.

2.5 Elements Changing the Internal State

To illustrate the difficulties involved in crawling AJAX, consider Figure 1. It is a highly simplified example, showing different ways in which a news page can be opened.

The example code shows how in AJAX sites, it is not just the hypertext link element that forms the doorway to the next state. Note the way events (e.g., `onClick`, `onMouseOver`) can be attached to DOM elements at runtime. As can be seen, a `div` element (line 3) can have an `onclick` event attached to it so that it becomes a *clickable* element capable of changing the internal DOM state of the application when clicked. The necessary event handlers can also be programmatically registered in AJAX. The jQuery[3] code responsible (lines 9–11) for attaching the required functionality to the `onClick` event handlers using the `class` attribute of the elements can also be seen.

Finding these clickables at run-time is another non-trivial task for a crawler. Traditional crawlers as used by search engines will simply ignore all the elements (not having a proper `href` attribute) except the one in line 4, since they rely on JavaScript only.

3 A Method for Crawling AJAX

The challenges discussed in the previous section will make it clear that crawling AJAX based on static analysis of, e.g., the HTML and JavaScript code is not feasible. Instead, we rely on a dynamic approach, in which we actually exercise clicks on all relevant elements in the DOM. From these clicks, we reconstruct a *state-flow graph*, which tells us in which states the user interface can be. Subsequently, we use these states to generate static, indexable, pages.

An overview of our approach is visualized in Figure 3. As can be seen, the architecture can be divided in two parts: (1) inferring the state machine, and (2) using the state machine to generate indexable pages.

In this section, we first summarize our state and state-flow graph definition, followed by a discussion of the most important steps in our approach.

3.1 User Interface States

In traditional multi-page web applications, each state is represented by a URL and the corresponding web page. In AJAX however, it is the internal structure change of the DOM tree on the (single-page) user interface that represents a state change. Therefore, to adopt a generic approach for all AJAX sites, we define a state change as a change on the DOM tree caused either by server-side state changes propagated to the client, or client-side events handled by the AJAX engine.

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

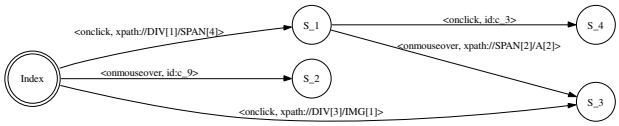

Figure 2. The state-flow graph visualization.

3.2 The State-flow Graph

The user interface state changes in AJAX can be modeled by recording the paths (events) 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* [18], but different in that in the former vertices are *states*, where as in the latter vertices are *events*.

As an example of a state-flow graph, Figure 2 depicts the visualization of the state-flow graph of a simple AJAX site. It illustrates how from the start page three different states can be reached. The edges between states are labeled with an identification (either via its ID-attribute or via an XPath expression) of the element to be clicked in order to reach the given state. Thus, clicking on the `//DIV[1]/SPAN[4]` element in the Index state leads to the `S_1` state, from which two states are reachable namely `S_3` and `S_4`.

3.3 Inferring the State Machine

The state-flow graph is created incrementally. Initially, it only contains the root state and new states are created and added as the application is crawled and state changes are analyzed.

The following components, also shown in Figure 3 participate in the construction of the state flow graph:

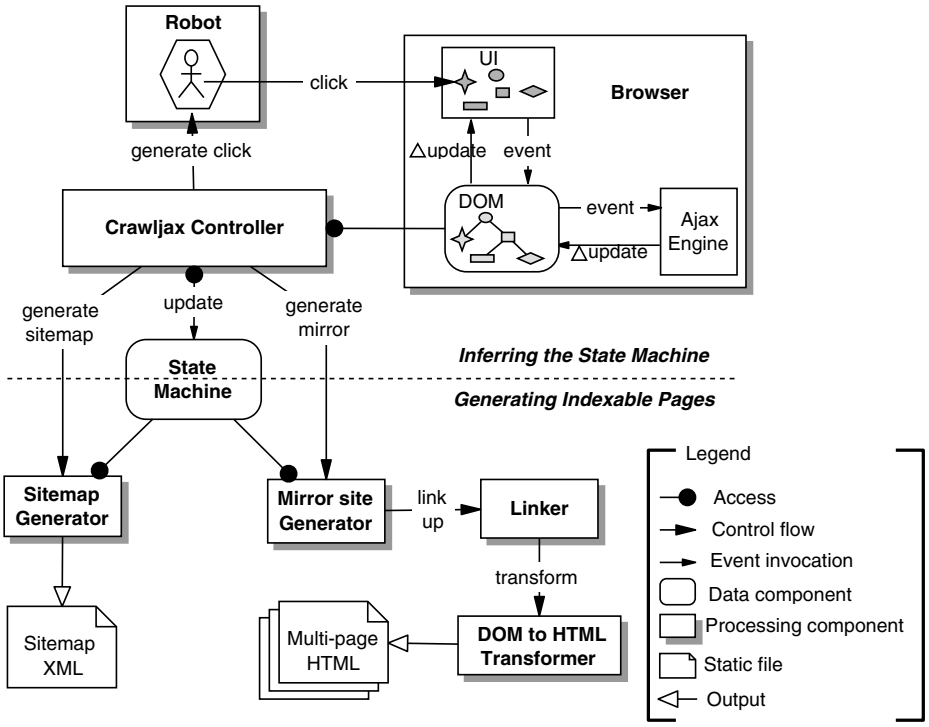

Figure 3. Processing view of the crawling architecture.

- Embedded Browser: Our approach is based on an embedded browser interface (with different implementations: IE, Mozilla) capable of executing JavaScript and the supporting technologies required by AJAX (e.g., CSS, DOM, `XMLHttpRequest`).

- Robot: A robot is used to simulate user input (e.g., `click`, `mouseOver`, text input) on the embedded browser.

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

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

The algorithm used by these components to actually infer the state machine is shown in Algorithm 1. The start procedure (lines 1-8) takes care of initializing the various components and processes involved. The actual, recursive, crawling procedure starts at line 10: the main steps are explained below.

3.4 Detecting Clickables

There is no direct way of obtaining all clickable elements in a DOM-tree, due to the reasons explained in Section 2. Therefore, our algorithm makes use of a set of *candidate elements*, which are all exposed to an event type (e.g., click, mouseOver). We use the *click* event type to present our algorithm, note, however, that other event types can be used just as well to analyze the effects on the DOM in the same manner.

We distinguish three ways of obtaining the *candidate* elements:

- In a *Full Auto Scan* mode, the candidate clickables are labeled as such based on their HTML tag element name. For example, all elements with a tag `div`, `a`, `span`, `input` are considered as candidate clickable. This is the mode that is displayed in Algorithm 1.

- In the *annotation* mode, we allow the HTML elements to have an attribute `crawljax="true"`. This gives users the opportunity to explicitly mark certain elements as to be crawled, or elements to be excluded from the process by setting the attribute to false. Note that this mode requires access to the source code of the application for applying the annotations.

- In the *configured* mode, we allow a user to specify by means of a domain-specific language which elements

Algorithm 1 Full Auto Scan

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

should be clicked (explained in more detail in Section 3.8). This allows the most precise control over the actual elements to be clicked.

Note that, if desirable, these modes can be combined. After the candidate elements have been found, the algorithm proceeds to determine whether these elements are indeed clickable. For each candidate element, the crawler instructs the robot to execute a click (line 15) on the element (or other event types, e.g., mouseOver), in the browser.

3.5 Creating States

After firing an event on a candidate clickable, the algorithm compares the resulting DOM tree with the DOM tree as it was just before the event fired, in order to determine whether the event results in a state change.

For this purpose the *edit distance* between two DOM trees is calculated (line 17) using the Levenshtein [15] 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 according to our similarity metric, we create (line 19) a new state and add it to the state-flow graph of the state machine (line 20). In order to recognize an already met state, we compute a hashcode for each DOM state and which we use to compare every new state to the list of already visited states on the state-flow graph. Thus, in line 19 if we have a state containing the particular DOM tree already, that state is returned, otherwise a new state is created.

Furthermore, a new edge is created on the graph (line 21) between the state before the event and the current state. The element on which the event was fired is also added as part of the new edge. Moreover, the current state pointer of the state machine is also updated to this newly added state at that moment (line 22).

3.6 Processing Document Tree Deltas

After a clickable has been identified, and its corresponding state created, the `crawl` procedure is recursively called (line 23) to find new possible states in the changes made to the DOM tree.

Upon every new (recursive) entry into the `crawl` procedure, the first thing done (line 12) is computing the differences between the previous document tree and the current one, by means of an enhanced *Diff* algorithm [6, 19]. Such "delta updates" may be due, for example, to a server request call that injects new elements into the DOM. The resulting delta updates are used to find new candidate clickables (line 13), which are then further processed in a depth-first manner.

It is worth mentioning that in order to avoid a loop, a list of visited elements is maintained to exclude already checked elements in the recursive algorithm. We use the tag name, the list of attribute names and values, and the XPath expression of each element to conduct the comparison. Additionally, a depth number can be defined to constrain the depth level of the recursive function (not shown in the algorithm).

3.7 Navigating the States

Upon completion of the recursive call, the browser should be put back into the state it was in before the call. Unfortunately, navigating (back and forth) through an AJAX site is not as easy as navigating a classical web site. A dynamically changed DOM state does not register itself with the browser history engine automatically, so triggering the 'Back' function of the browser does not bring us to the previous state. This complicates traversing the application when crawling AJAX. We distinguish two situations:

Browser History Support It is possible to programatically register each state change with the browser history through frameworks such as the jQuery history/remote plugin[4] or the Really Simple History library[5]. If an AJAX application has support for the browser history (line 25), then for changing the state in the browser, we can simply use the built-in history back functionality to move backwards (line 26).

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 saving information about the elements and the order in which their execution results in reaching to a particular state. Once we have such information, we can reload the application (line 28) and follow and execute the elements from the initial state to the desired state. As an optimization step, we use Dijkstra's shortest path algorithm [10] to find the shortest element execution path on the graph to a certain state (line 29).

We initially considered using the ID attribute of a clickable element to find it back after a reload of the page. When we `reload` the application in the browser, all the internal objects are replaced by new ones and the ID attribute would be a way to follow the path to a certain state by clicking on those elements whose IDs have been saved in the state machine. Soon we realized that firstly, not all AJAX sites assign ID attributes to the elements and, secondly, if IDs are provided, they are not always persistent, i.e., they are dynamically set and can change with each reload.

To overcome these challenges, we adopt XPath to provide a better, more reliable, and persistent element identification mechanism. For each state changing element, we reverse engineer the XPath expression of that element which gives us its exact location on the DOM (line 18). We save this expression in the state machine (line 19) and use it to find the element after a reload, persistently (line 31).

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

[4] `http://stilbuero.de/jquery/history/`

[5] `http://code.google.com/p/reallysimplehistory/`

```
crawl MyAjaxSite {
  url: http://spci.st.ewi.tudelft.nl/aowe/;
  navigate Nav1 {
    event: type=mouseover xpath=/HTML/BODY/SPAN[3];
    event: type=click id=headline;
    ...
  }
  navigate Nav2 {
    event: type=click
           xpath="//DIV[contains(.,"Interviews")]";
    event: type=input id=article "john doe";
    event: type=click id=search;
  } ...
}
```

Figure 4. An instance of CASL.

3.8 CASL: Crawling AJAX Specification Language

To give users control over which candidate clickables to select, we have developed a Domain Specific Language (DSL) [9] called Crawling AJAX Specification Language (CASL). Using CASL, the developer can define the elements (based on IDs and XPath expressions) to be clicked, along with the exact order in which the crawler should crawl the AJAX application. CASL accepts different types of `events`. The event types include `click`, `mouseover`, and `input` currently.

Figure 4 shows an instance of CASL. `Nav1` tells our crawler to crawl by first firing an event of type `mouseover` on the element with XPath `/HTML/BODY/SPAN[3]` and then clicking on the element with ID `headline` in that order. `Nav2` commands the crawler to crawl to the `Interviews` state, then insert the text '`john doe`' into the input element with ID `article` and afterward click on the `search` element. Using this DSL, the developer can take control of the way an AJAX site should be crawled.

3.9 Generating Indexable Pages

After the crawling AJAX process is finished, the created state-flow graph can be passed to the generation process, corresponding to the bottom part of Figure 3.

The first step is to establish links for the DOM states by following the *outgoing edges* of each state in the state-flow graph. For each clickable, the element type must be examined. If the element 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 element by a hypertext link element. The `href` attribute in both situations represents the link to the name and location of the generated static page.

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

Here, we can adhere to the Sitemap Protocol[6], generating a valid instance of the protocol automatically after each crawling session consisting of the URLs of all generated static pages.

4 Tool Implementation

We have implemented the concepts presented in this paper in a tool called CRAWLJAX. CRAWLJAX is released under the open source BSD license and is available for download. More information about the tool can be found on our website `http://spci.st.ewi.tudelft.nl/crawljax/`.

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

The embedded browser interface has two implementations: IE-based on Watij[7] and Mozilla-based on XULRunner[8]. Webclient[9] is used to access the run-time DOM and the browser history mechanism in the Mozilla browser. For the Mozilla version, the Robot component makes use of the `java.awt.Robot` class to generate native system input events on the embedded browser. The IE version uses an internal Robot to simulate events.

The generator uses JTidy[10] to pretty-print DOM states and Xerces[11] to serialize the objects to HTML. In the Sitemap Generator, XMLBeans[12] generates Java objects from the Sitemap Schema[13] which after being used by CRAWLJAX to create new URL entries, are serialized to the corresponding valid XML instance document.

The *state-flow graph* is based on the JGrapht[14] library. The grammar of CASL is implemented in ANTLR[15]. ANTLR is used to generate the necessary parsers for CASL. In addition, StringTemplate[16] is used for generating the source-code from CASL. Log4j is used to optionally log various steps in the crawling process, such as the identification of DOM changes and clickables. CRAWLJAX is entirely based on Maven[17] to generate, compile, test (JUnit), release, and run the application.

[6] `http://www.sitemaps.org/protocol.php`
[7] `http://watij.com`
[8] `http://developer.mozilla.org/en/docs/XULRunner/`
[9] `http://www.mozilla.org/projects/blackwood/webclient/`
[10] `http://jtidy.sourceforge.net`
[11] `http://xerces.apache.org/xerces-j/`
[12] `http://xmlbeans.apache.org`
[13] `http://www.sitemaps.org/schemas/sitemap/0.9/sitemap.xsd`
[14] `http://jgrapht.sourceforge.net`
[15] `http://www.antlr.org`
[16] `http://www.stringtemplate.org`
[17] `http://maven.apache.org`

5 Case Studies

In order to evaluate the effectiveness, correctness, performance, and scalability of the proposed crawling method for AJAX, we have conducted a number of case studies, which are described in this section, following Yin's guidelines for conducting case studies [24].

5.1 Subject Systems

We have selected 6 AJAX sites for our experiment as shown in Table 1. The case ID, the actual site, and a number of real clickables to illustrate the type of the elements can be seen for each case object.

Our selection criteria include the following: sites that use AJAX to change the state of the application by using JavaScript, assigning events to HTML elements, asynchronously retrieving delta updates from the server and performing partial updates on the DOM.

The first site C1 in our case study is an AJAX test site developed internally by our group 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 1 and Table 1.

Our second case object, C2, is Sun's Ajaxified PETSTORE 2.0[18] which is built on the Java ServerFaces, and the Dojo AJAX toolkit. 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.

The other four cases are all external AJAX sites and we have no access to their source-code. C4 is an AJAX site that can function as a tool for comparing the visual impression of different typefaces. C3 (online shop), C5 (sport center), and C6 (Gucci) are all single-page commercial sites with many clickables and states.

5.2 Experimental Design

Our goals in conducting the experiment include:

G1 Effectiveness: evaluating the effectiveness of obtaining high-quality results in retrieving relevant clickables including the ones dynamically injected into the DOM,

G2 Correctness: assessing the quality and correctness of the states and static pages automatically generated,

G3 Performance: analyzing the overall performance of our approach in terms of input size versus time,

[18] `http://java.sun.com/developer/releases/petstore/`

Table 1. Case objects and examples of their clickable elements.

Case	AJAX site	Clickable Elements
C1	spci.st.ewi.tudelft.nl/demo/aowe/	`<span id="testspan2" class="testing">testing span 2</span>` `<a onclick="nav('l2'); return false;" href="#">Second link</a>` `<a title="Topics" href="#Topics" class="remoteleft left">Topics of Interest</a>`
C2	PETSTORE	`<a class="accordionLink" href="#" id="feline01" onmouseout="this.className= 'accordionLink';" onmouseover="this.className='accordionLinkHover';">Hairy Cat</a>`
C3	www.4launch.nl	`<div onclick="setPrefCookies('Gaming', 'DESTROY', 'DESTROY'); loadHoofdCatsTree('Gaming', 1, '')"><a id="uberCatLink1" class="ubercat" href="javascript:void(0)">Gaming</a></div>` `<td onclick="open_url('..producteninfo.php?productid=037631',..)">Harddisk Skin</td>`
C4	www.blindtextgenerator.com	`<input type="radio" value="7" name="radioTextname" class="js-textname iradio" id="idRadioTextname-EN-li-europan"/>` `<a id="idSelectAllText" title="Select all" href="#">`
C5	site.snc.tudelft.nl	`<div class="itemtitlelevel1 itemtitle" id="menuitem_189_e">organisatie</div>` `<a href="#" onclick="ajaxNews('524')">...</a>`
C6	www.gucci.com[a]	`<a onclick="Shop.selectSort(this); return false" class="booties" href="#">booties</a>` `<div id="thumbnail_7" class="thumbnail highlight"><img src="...001_thumb.jpg" /><div class="darkening"/></div>`

[a] `http://www.gucci.com/nl/uk-english/nl/spring-summer-08/womens-shoes/`

G4 Scalability: examining the capability of CRAWLJAX on real sites used in practice and the scalability in crawling sites with thousands of dynamic states and clickables.

Environment & Tool Configuration
We use a laptop with Intel Pentium M 765 processor 1.73GHz, with 1GB RAM and Windows XP to run CRAWLJAX.

Configuring CRAWLJAX itself is done through a simple `crawljax.properties` file, which can be used to set the URL of the site to be analyzed, the tag elements CRAWLJAX should look for, the depth level, and the similarity threshold. There are also a number of other configuration parameters that can be set, such as the directory in which the generated pages should be saved in.

Output
We determine the average DOM string size, number of candidate elements, number of detected clickables, number of detected states, number of generated static pages, and performance measurements for crawling and generating pages separately for each experiment object. The actual generated linked static pages also form part of the output.

Method of Evaluation
Since other comparable tools and methods are currently not available to conduct similar experiments as with CRAWLJAX, it is difficult to define a baseline against which we can compare the results. Hence, we manually inspect the systems under examination and determine which expected behavior should form our reference baseline.

G1: For the experiment we have manually added extra clickables in different states of C1, especially in the delta updates, to explore whether clickables dynamically injected into the DOM can be found by CRAWLJAX. 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. To constrain the reference model for C2, we chose two product categories, namely CATS and DOGS, from the five available categories. We annotated 36 elements (product items) by modifying a JavaScript method which turns the items retrieved from the server into clickables on the interface. For the four external sites (C3–C6) which have many states, it is very difficult to manually inspect and determine, for instance, the number of expected clickables and states. Therefor, for each site, we randomly selected 10 clickables in advance by noting their tag name, attributes, and XPath expression. After each crawling process, we checked the presence of the 10 elements among the list of detected clickables.

G2: After the generation process the generated HTML files and their content are manually examined to see whether the pages are the same as the corresponding DOM states in AJAX in terms of *structure*, *style*, and *content*. Also the internal linking of the static pages is manually checked. To test the clone detection ability we have intentionally introduced a clone state into C1.

G3: We measure the time in milliseconds taken to crawl each site. We expect the crawling performance to be directly proportional to the input size which is comprised of the average DOM string size, number of candidate elements, and number of detected clickables and states.

We also measure the generation performance which is the period taken to generate the static HTML pages from the inferred state-flow graph.

G4: To test the capability of our method in crawling real sites and coping with unknown environments, we run CRAWLJAX on four external cases C3–C6. We run CRAWLJAX with depth level 2 on C3 and C5 each having a

huge state space to examine the scalability of our approach in analyzing tens of thousands of candidate clickables and finding clickables.

5.3 Results and Evaluation

Table 2 presents the results obtained by running CRAWLJAX on the subject systems. The measurements were all read from the log file produced by CRAWLJAX at the end of each process.

G1 As can be seen in Table 2, for C1 CRAWLJAX finds all the 16 expected clickables and states with a precision and recall of 100%.

For C2, 33 elements were detected from the annotated 36. One explanation behind this difference could be the way some items are shown to the user in PETSTORE. PETSTORE uses a `Catalog Browser` to show a set of the total number of the product items. The 3 missing product items could be the ones that were never shown on the interface because of the navigational flow e.i., the order of clickables.

CRAWLJAX was able to find 95% of the expected 10 clickables (noted initially) for each of the four external sites C3–C6.

G2 The clone state introduced in C1 is correctly detected and that is why we see 16 states being reported instead of 17. Inspection of the static pages in all cases shows that the generated pages correspond correctly to the DOM state.

G3 When comparing the results for the two internal sites, we see that it takes CRAWLJAX 14 and 26 seconds to crawl C1 and C2 respectively. As can be seen, the DOM in C2 is 5 times and the number of candidate elements 3 times higher. In addition to the increase in DOM size and the number of candidate elements, CRAWLJAX cannot rely on the browser Back method when crawling C2. 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 C2 due to the increase in the input size. It is clear that the running time of CRAWLJAX increases linearly with the size of the input. We believe that the execution time of a few minutes to crawl and generate a mirror multi-page instance of an AJAX application automatically without any human intervention is very promising. Note that the performance is also dependent on the CPU and memory of the machine CRAWLJAX is running on, as well as the speed of the server and network properties of the case site. C6, for instance, is slow in reloading and retrieving updates from its server and that increases the performance measurement numbers in our experiment.

G4 CRAWLJAX was able to run smoothly on the external sites. Except a few minor adjustments (see Section 6) we did not witness any difficulties. C3 with depth level 2 was crawled successfully in 83 minutes resulting in 19247 examined candidate elements, 1101 detected clickables, and 1071 detected states. The generation process for the 1071 states took 13 minutes. For C5, CRAWLJAX was able to finish the crawl process in 107 minutes on 32365 candidate elements, resulting in 1554 detected clickables and 1234 states. The generation process took 13 minutes. As expected, in both cases, increasing the depth level from 1 to 2 expands the state space greatly.

6 Discussion

6.1 Back Implementation

CRAWLJAX assumes that if the Browser Back functionality is implemented, then it is implemented correctly. An interesting observation was the fact that even though Back is implemented for some states, it is not correctly implemented i.e., calling the Back method brings the browser in a different state than expected which naturally confuses CRAWLJAX. This implies that the Back method to go to a previous state is not reliable and using the reload and click-through method is much more safe.

6.2 Constantly Changing DOM

Another interesting observation in C2 in the beginning of the experiment was that every element was seen as a clickable. 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.

6.3 Cookies

Cookies can also cause some problems in crawling AJAX applications. C3 uses Cookies to store the state of the application on the client. With Cookies enabled, when CRAWLJAX reloads the application to navigate to a previous state, the application does not start in the expected initial state. In this case, we had to disable Cookies to perform a correct crawling process.

6.4 State Space

The set of found states and generated HTML pages is by no means complete, i.e., CRAWLJAX generates a static instance of the AJAX application but not necessarily *the* instance. This is partly inherent in dynamic web applications.

Table 2. Results of running CRAWLJAX on 6 AJAX applications.

Case	DOM string size (byte)	Candidate Elements	Detected Clickables	Detected States	Generated Static Pages	Crawl Performance (ms)	Generation Performance (ms)	Depth	Tags
C1	4590	540	16	16	16	14129	845	3	A, DIV, SPAN, IMG
C2	24636	1813	33	34	34	26379	1643	2	A, IMG
C3	262505	150	148	148	148	498867	17723	1	A
		19247	1101	1071	1071	5012726	784295	2	A, TD
C4	40282	3808	55	56	56	77083	2161	2	A, DIV, INPUT, IMG
C5	165411	267	267	145	145	806334	14395	1	A
		32365	1554	1234	1234	6436186	804139	2	A, DIV
C6	134404	6972	83	79	79	701416	28798	1	A, DIV

Any crawler can only crawl and index a snapshot instance of a dynamic web application in a point of 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.

The number of possible states in the state space of almost any realistic web application is huge and can cause the well-know *state explosion problem* [23]. Just as a traditional web crawler, CRAWLJAX provides the user with a set of configurable options to constrain the state space such as the maximum search depth level, the similarity threshold, maximum number of states per domain, maximum crawling time, and the option of ignoring external links and links that match some pre-defined set of regular expressions, e.g., mail:*, *.ps, *.pdf.

The current implementation of CRAWLJAX keeps the DOM states in the memory which can lead to an state explosion and out of memory exceptions with approximately 3000 states on a machine with a 1GB RAM. As an optimization step we intend to abstract and serialize the DOM state into a database and only keep a reference in the memory. This saves much space in the memory and enables us to handle much more states. With a cache mechanism, the essential states for analysis can be kept in the memory while the other ones can be retrieved from the database when needed in a later stage.

7 Applications

As mentioned in the introduction, we believe that the crawling and generating capabilities of our approach have many applications for AJAX sites.

We believe that the crawling techniques that are part of our solution can serve as a starting point and be adopted by general search engines to be able to crawl AJAX sites. 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 [4] 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 [4]. With the wide adoption of AJAX techniques that we are witnessing today this figure will only increase. Although there has been extensive research on crawling and exposing the data behind forms [4, 8, 14, 21, 22], crawling 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 [20, 5], AJAX sites themselves may very well be ignored by the search engines.

There are some industrial proposed techniques that assist in making a modern AJAX website more accessible and discoverable by general search engines. In web engineering terms, the concept behind *Graceful Degradation* [12] 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. The term *Progressive Enhancement*[19] 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.

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*[20] 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. Another option is the use of XML/XSLT to generate indexable pages for search crawlers [3]. 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 Graceful Degradation and Progressive Enhancement approaches mentioned constrain the use of AJAX and have limitations in the content exposing degree. It is very hard to imagine a single-page desktop-style AJAX application that degrades into a plain HTML website using the same markup and client-side code. The more complex the AJAX functionality, the higher the cost of weaving advanced and accessible functionality into the components[21]. The server-side generation approaches increase the complexity, development costs, and maintainability effort as well. We believe our proposed solution can assist the web 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. Such an automatically built mirror site can also improve the accessibility[22] of the application towards user-agents that do not support JavaScript.

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 tool not only for crawling but also for automatic testing of AJAX applications.

The ability to automatically exercise all the executable elements of an AJAX site gives us a powerful test mechanism. The crawler can be utilized to find abnormalities is AJAX sites. As an example, while conducting the case study, we noticed a number of *404 Error*s and exceptions on C3 and C4 sites. Such errors can easily be detected and traced back to the elements and states causing the error state in the inferred state-flow graph. The asynchronous interaction in AJAX can cause race conditions [20] between requests and responses, and the dynamic DOM updates can also introduce new elements which can be sources of faults. Detection of such conditions by analyzing the generated state machine and static pages can be assisted as well. In addition, testing AJAX sites for compatibility on different browsers (e.g., IE, Mozilla) can be automated using CRAWLJAX.

The crawling methods and the produced state machine can be applied in conducting state machine testing [1] for automatic test case derivation, verification, and validation based on pre-defined conditions for AJAX applications.

8 Related Work

The concept behind CRAWLJAX, is the opposite direction of our earlier work RETJAX [19], in which we try to reverse-engineer a traditional multi-page website to AJAX.

The work of Memon *et al.* [17, 18] 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.

There are some industrial proposed approaches for improving the accessibility and discoverability of AJAX as discussed in Section 7.

There has been extensive research on crawling the hidden-web behind forms [4, 7, 8, 14, 21, 22]. This is sharp contrast with the the hidden-web induced as a result of client-side scripting in general and AJAX in particular, which has gained very little attention so far. As far as we know, there are no academic research papers on crawling AJAX at the moment.

9 Concluding Remarks

Crawling AJAX is the process of turning a highly dynamic, interactive web-based system into a static mirror site, a process that is important to improve searchability, testability, and accessibility of AJAX applications. This paper proposes a crawling method for AJAX. The main contributions of the paper are:

- An analysis of the key problems involved in crawling AJAX applications;
- A systematic process and algorithm to infer a state machine from an AJAX application, which can be used to

[19] http://hesketh.com/publications/progressive_enhancement_paving_way_for_future.html

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

[21] http://blogs.pathf.com/agileajax/2007/10/accessibility-a.html

[22] http://bexhuff.com/node/165

generate a static mirror site. Challenges addressed include the identification of clickable elements, the detection of DOM changes, and the construction of the state machine;

- The open source tool CRAWLJAX, which implements this process;
- Six case studies used to evaluate the effectiveness, correctness, performance, and scalability of the proposed approach.

Although we have been focusing on AJAX in this paper, we believe that the approach could be applied to any DOM-based web application.

Future work consists of conducting more case studies to improve the ability of finding clickables in different AJAX settings. The fact that the tool is available for download for everyone, will help to identify exciting case studies. Furthermore, strengthening the tool by extending its functionality, improving the performance, and the state explosion optimization are other directions we foresee. Exposing the hidden-web induced by AJAX using CRAWLJAX and conducting automatic state-based testing of AJAX application based on the reverse engineering techniques are other applications we will be working on.

References

[1] A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4(3):326–345, July 2005.

[2] R. Atterer and A. Schmidt. Adding usability to web engineering models and tools. In *Proceedings of the 5th International Conferencee on Web Engineering (ICWE'05)*, pages 36–41. Springer, 2005.

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

[4] 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.

[5] E. Bozdag, A. Mesbah, and A. van Deursen. A comparison of push and pull techniques for Ajax. In *Proceedings of the 9th IEEE International Symposium on Web Site Evolution (WSE'07)*, pages 15–22. IEEE Computer Society, 2007.

[6] 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.

[7] 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.

[8] 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.

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

[10] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

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

[12] 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.

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

[14] 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.

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

[16] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *Proceedings of the 1st IEEE International Conference on Software Testing Verification and Validation (ICST'08)*. IEEE Computer Society, 2008.

[17] 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.

[18] 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. ACM Press, 2001.

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

[20] A. Mesbah and A. van Deursen. A component- and push-based architectural style for Ajax applications. *Journal of Systems and Software (JSS)*, 2008. To appear.

[21] 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.

[22] 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.

[23] A. Valmari. The state explosion problem. In *LNCS: Lectures on Petri Nets I, Basic Models, Advances in Petri Nets*, pages 429–528. Springer-Verlag, 1998.

[24] R. K. Yin. *Case Study Research: Design and Methods*. SAGE Publications Inc, 3d edition, 2003.