Crawling AJAX-Based Web Applications through Dynamic Analysis of User Interface State Changes

ALI MESBAH, University of British Columbia
ARIE VAN DEURSEN and STEFAN LENSELINK, Delft University of Technology

Using JAVASCRIPT and dynamic DOM manipulation on the client side of Web applications is becoming a widespread approach for achieving rich interactivity and responsiveness in modern Web applications. At the same time, such techniques—collectively known as AJAX—shatter the concept of webpages with unique URLs, on which traditional Web crawlers are based. This article describes a novel technique for crawling AJAX-based applications through automatic dynamic analysis of user-interface-state changes in Web browsers. Our algorithm scans the DOM tree, spots candidate elements that are capable of changing the state, fires events on those candidate elements, and incrementally infers a state machine that models the various navigational paths and states within an AJAX application. This inferred model can be used in program comprehension and in analysis and testing of dynamic Web states, for instance, or for generating a static version of the application. In this article, we discuss our sequential and concurrent AJAX crawling algorithms. We present our open source tool called CRAWLJAX, which implements the concepts and algorithms discussed in this article. Additionally, we report a number of empirical studies in which we apply our approach to a number of open-source and industrial Web 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 Storage and Retrieval**]: Information and Retrieval—*Search process*; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms: Design, Algorithms, Experimentation

Additional Key Words and Phrases: Crawling, Ajax, Web 2.0, hidden web, dynamic analysis, DOM crawling

ACM Reference Format:
Mesbah, A., van Deursen, A., and Lenselink, S. 2012. Crawling AJAX-based Web applications through dynamic analysis of user interface state changes. ACM Trans. Web 6, 1, Article 3 (March 2012), 30 pages.
DOI = 10.1145/2109205.2109208 http://doi.acm.org/10.1145/2109205.2109208

1. INTRODUCTION

The Web as we know it is undergoing a significant change. A technology that has lately gained a prominent position, under the umbrella of *Web 2.0*, is AJAX (Asynchronous JAVASCRIPT and XML) [Garrett 2005], in which the combination of JAVASCRIPT and Document Object Model (DOM) manipulation, along with asynchronous server

This is a substantially revised and expanded version of our paper "Crawling AJAX by Inferring User Interface State Changes," which appeared in the *Proceedings of the 8th International Conference on Web Engineering (ICWE)* [Mesbah et al. 2008].

Authors' addresses: A. Mesbah, Department of Electrical and Computer Engineering, University of British Columbia, 2332 Main Mall, V6T1Z4 Vancouver, BC, Canada; email: amesbah@ece.ubc.ca; A. van Deursen and S. Lenselink, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands; emails: arie.vandeursen@tudelft.nl and; S.R.Lenselink@student.tudelft.nl.

DOI 10.1145/2109205.2109208 http://doi.acm.org/10.1145/2109205.2109208

communication is used to achieve a high level of user interactivity. Highly visible examples include Gmail and Google Docs.

With this new change in developing Web applications comes a whole set of new challenges, mainly due to the fact that AJAX shatters the concept of a webpage upon which many Web technologies are based. Among these challenges are the following.

- *Searchability*. Searchability ensures that AJAX sites are crawled and 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*. Testability involves systematically dynamic user interface (UI) elements and analyzing AJAX states to find abnormalities and errors.

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 and create a model of the navigational paths and states.

General Web search engines, such as Google and Bing, cover only a portion of the Web called the *publicly indexable Web* that consists of the set of webpages reachable purely by following hypertext links, ignoring forms [Barbosa and Freire 2007] and client-side scripting. The Web content behind forms and client-side scripting is referred to as the *hidden Web*, which is estimated to comprise several millions of pages [Barbosa and Freire 2007]. 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 [Barbosa and Freire 2007; de Carvalho and Silva 2004; Lage et al. 2004; Ntoulas et al. 2005; Raghavan and Garcia-Molina 2001], crawling the hidden web, induced as a result of client-side scripting, has gained very little attention so far.

Crawling AJAX-based applications 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 article, we propose an approach for analyzing these user interface states automatically. Our approach is based on a crawler that can exercise client-side code and identify clickable elements (which may change with every click) that change the state within the browser's dynamically built DOM. From these state changes, we infer a *state-flow graph*, which captures the states of the user interface and the possible transitions between them. The underlying ideas have been implemented in an open source tool called CRAWLJAX.[1] To the best of our knowledge, CRAWLJAX is the first and currently the only available tool that can detect dynamic contents of AJAX-based Web applications automatically without requiring specific URLs for each Web state.

The inferred model can be used, for instance, to expose AJAX sites to general search engines or to examine the accessibility [Atterer and Schmidt 2005] of different dynamic states. The ability to automatically exercise all the executable elements of an AJAX site gives us a powerful test mechanism. CRAWLJAX has successfully been used for conducting automated model-based and invariant-based testing [Mesbah and van Deursen 2009], security testing [Bezemer et al. 2009], regression testing [Roest et al. 2010], and cross-browser compatibility testing [Mesbah and Prasad 2011] of AJAX Web applications.

[1] http://crawljax.com

We have performed a number of empirical studies to analyze the overall performance of our approach. We evaluate the effectiveness in retrieving relevant clickables and assess the quality and correctness of the detected states and edges. We also examine the performance of our crawling tool, as well as the scalability in crawling AJAX applications with a large number of dynamic states and clickables. The experimental benchmarks span from open source to industrial Web applications.

This article is a revised and extended version of our original paper in 2008 [Mesbah et al. 2008]. The extensions are based on three years of tool usage and refinements. In addition, we report on our new multi-threaded, multi-browser crawling approach, as well as a new (industrial) empirical study, evaluating its influence on the runtime performance. The results of our study show that by using five browsers instead of one, we can achieve a decrease of up to 65% in crawling runtime.

The article is further structured as follows. We start out in Section 2 by exploring the difficulties of crawling AJAX. In Section 3 we present a detailed discussion of our AJAX crawling algorithm and technique. In Section 4 we extend our sequential crawling approach to a concurrent multiple-browser crawling algorithm. Section 5 discusses the implementation of our tool CRAWLJAX. In Section 6, the results of applying our techniques to a number of AJAX applications are shown, after which Section 7 discusses the findings and open issues. We conclude with a survey of related work, a summary of our key contributions, and suggestions for future work.

2. CHALLENGES OF CRAWLING AJAX

AJAX-based Web applications have a number of properties that make them very challenging to crawl automatically.

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 [Mesbah and van Deursen 2008]. This engine typically deals with server communication and user-interface modifications. Any search engine willing to approach such an application must have support for the execution of the scripting language. Equipping a crawler with the necessary environment complicates its design and implementation considerably. The major search engines, such as Google and Bing, currently have little or no support for executing scripts, and thus ignore content produced by JAVASCRIPT,[2] due to scalability and security issues.

2.2. State Changes and Navigation

Traditional Web applications are based on the multi-page interface paradigm consisting of multiple pages, each having a unique URL. In AJAX applications, not every state change necessarily has an associated REST-based URI [Fielding and Taylor 2002]. Ultimately, an AJAX application could consist of a single page with a single URL [van Deursen 2007]. This characteristic makes it difficult for a search engine to index and point to a specific state in an AJAX application. Crawling traditional webpages constitutes extracting and following the hypertext links (the `src` attribute of anchor tags) on each page. In AJAX, hypertext links can be replaced by events which are handled by JAVASCRIPT; that is, it is no longer possible to navigate the application by simply extracting and retrieving the internal hypertext links.

[2] `http://code.google.com/web/ajaxcrawling/docs/getting-started.html`

```
1  <A href="javascript:OpenNewsPage();">...
2  <A href="#" onClick="OpenNewsPage();">...
3  <DIV onClick="OpenNewsPage();">...

5  <DIV class="news"/>
6  <SPAN id="content"/>
7  <!-- jQuery function attaching events to elements having attribute class="news".
8  The news contents are injected into the SPAN element -->
9  <script>
10   $(".news").click(function() {
11     $("#content").load("news.html");
12 });
13 </script>
```

Fig. 1. Different ways of attaching events to elements.

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 runtime changes on the DOM tree in the browser. This means that the initial HTML source code retrieved from the server does not represent the state changes. An AJAX crawler will need to have access to this runtime dynamic document object model of the application.

2.4. Delta Communication

AJAX applications rely on a delta-communication style of interaction [Mesbah and van Deursen 2008], 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 data served by the server, through a proxy between the client and the server, for instance, could have the side effect of losing the context and actual meaning of the changes, because most of such updates become meaningful after they have been processed by the JAVASCRIPT engine on the client and injected into the runtime DOM tree.

2.5. Clickable 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, 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.

Event handlers can also be dynamically registered using JAVASCRIPT. The jQuery[3] code (lines 5–13) attaches a function to the `onClick` event listener of the element with `class` attribute `news`. When this element is clicked, the news content is retrieved from the server and injected into the `SPAN` element with ID `content`.

There are different ways to attach event listeners to DOM elements. For instance, if we have the following handler.

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

```
    var handler = function()  alert(’Element clicked!’)  ;
```

we can attach it to an `onClick` listener of a DOM element `e` in the following ways.

```
    (1) e.onclick = handler;

    (2) if(e.addEventListener) {
            e.addEventListener(‘click’, handler, false)
        } else if(e.attachEvent) { // IE
                    e.attachEvent(‘onclick’, handler)
        }
```

In the first case (1), we can examine the DOM element at runtime and find out that it has the handler attached to its onClick listener. In the second case (2), however, there is no way of finding this information by examining the DOM element, since the event model maintaining the handler registration information is separated from the DOM.

Finding these clickable elements at runtime is another nontrivial task for an AJAX crawler.

3. A METHOD FOR CRAWLING AJAX

The challenges discussed in the previous section should make it clear that crawling AJAX is more demanding than crawling the classical Web. In this section, we propose a *dynamic analysis* approach, in which we open the AJAX application in a browser, scan the DOM tree for candidate elements that are capable of changing the state, fire events on those elements, and analyze the effects on the DOM tree. Based on our analysis, we infer a *state-flow graph* representing the user interface states and possible transitions between them.

In this section, we first present the terminology used in this article, followed by a discussion of the most important components of our crawling technique, as depicted in Figure 3.

3.1. Terminology

3.1.1. User Interface State and State Changes. In traditional multi-page Web applications, each state is represented by a URL and the corresponding webpage. In AJAX however, it is the internal structure of the DOM tree of the (single-page) user interface that represents a state. Therefore, to adopt a generic approach for all AJAX sites, we define a state change as a change on the DOM tree caused by (1) either client-side events handled by the AJAX engine, (2) or server-side state changes propagated to the client.

3.1.2. State Transition and Clickable. On the browser, the end user can interact with the Web application through the user interface by clicking on an element, bringing the mouse-pointer over an element, and so on. These actions can cause events that, as just described, can potentially change the state of the application. We call all DOM elements that have event listeners attached to them and can cause a state transition, *clickable* elements. For the sake of simplicity, we use the *click* event type to present our approach; note, however, that other event types can be used just as well to analyze the effects on the DOM in the same manner.

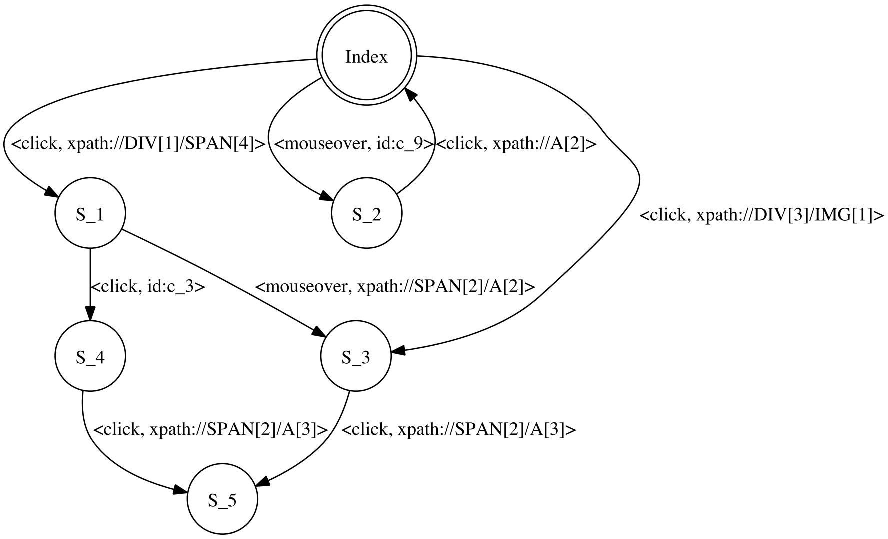

Index
<click, xpath://DIV[1]/SPAN[4]>
<mouseover, id:c_9>
<click, xpath://A[2]>
<click, xpath://DIV[3]/IMG[1]>
S_1
S_2
<click, id:c_3>
<mouseover, xpath://SPAN[2]/A[2]>
S_4
S_3
<click, xpath://SPAN[2]/A[3]>
<click, xpath://SPAN[2]/A[3]>
S_5

Fig. 2. The state-flow graph visualization.

3.1.3. The State-Flow Graph. To be able to navigate an AJAX-based Web application, the application can be modeled by recording the click trails to the various user-interface-state changes. To record the states and transitions between them, we define a state-flow graph as follows.

Definition 1. A state-flow graph G for an AJAX site $\mathbf{A}$ is a labeled, directed graph, denoted by a 4 tuple $< \boldsymbol{r}, \boldsymbol{V}, \boldsymbol{E}, \mathcal{L} >$ where

(1) $\boldsymbol{r}$ is the root node (called Index) representing the initial state after $\mathbf{A}$ has been fully loaded into the browser.
(2) $\boldsymbol{V}$ is a set of vertices representing the states. Each $v \in \boldsymbol{V}$ represents a runtime DOM state in $\mathbf{A}$.
(3) $\boldsymbol{E}$ is a set of (directed) edges between vertices. Each $(v_1, v_2) \in \boldsymbol{E}$ represents a clickable c connecting two states if and only if state v_2 is reached by executing c in state v_1.
(4) $\mathcal{L}$ is a labeling function that assigns a label from a set of event types and DOM element properties to each edge.
(5) G can have multi-edges and be cyclic.

As an example of a state-flow graph, Figure 2 depicts the visualization of the state-flow graph of a simple AJAX site. The edges between states are labeled with an identification (either via its ID attribute or an XPath expression) of the clickable. Thus, clicking on the `//DIV[1]/SPAN[4]` element in the Index state leads to the `S_1` state, from which two states are directly reachable, namely `S_3` and `S_4`.

The state-flow graph is created incrementally. Initially, it only contains the root state; 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.

— *Embedded Browser*. The embedded browser provides a common interface for accessing the underlying engines and runtime objects, such as the DOM and JAVASCRIPT.

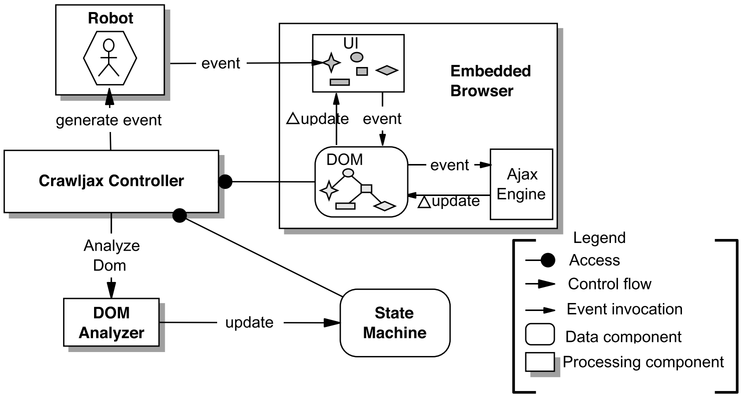


Fig. 3. Processing view of the crawling architecture.

—*Robot*. A robot is used to simulate user actions (e.g., `click`, `mouseOver`, text input) on the embedded browser.
—*Controller*. The controller has access to the embedded browser's DOM. It also controls the Robot's actions and is responsible for updating the state machine when relevant changes occur on the DOM.
—*DOM Analyzer*. The analyzer is used to check whether the DOM tree has changed after an event has been fired by the robot. In addition, it is used to compare DOM trees when searching for duplicate states in the state machine.
—*Finite State Machine*. The finite state machine is a data component maintaining the state-flow graph, as well as a pointer to the state being currently crawled.

3.2. Inferring the State Machine

The algorithm used by these components to actually infer the state machine is shown in Algorithm 1. The main procedure (lines 1–5) takes care of initializing the various components and processes involved. The actual, recursive crawl procedure starts at line 6. The main steps of the crawl procedure are then explained.

3.3. Detecting Clickables

There is no feasible way to automatically obtain a list of all clickable elements on 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). Each element on the DOM tree that meets the labeling requirements is selected as a candidate element.

In automated mode, the candidate clickables are labeled as such based on their HTML tag element name. In our implementation, all elements with a tag `<A>`, `<BUTTON>`, or `<INPUT type=submit>` are considered as candidate clickables, by default. The selection of candidate clickables can be relaxed or constrained by the user as well, by defining element properties such as the XPATH position on the DOM tree, attributes and their values, and text values. For instance, the user could be merely interested in examining `DIV` elements with attribute `class='article'`. It is also possible to exclude certain elements from the crawling process.

Based on the given definition of the candidate clickables, our algorithm scans the DOM tree and extracts all the DOM elements that meet the requirements of the definition (line 12). For each extracted candidate element, the crawler then instructs the robot to fill in the detected data entry points (line 14) and fire an event (line 15) on the

ALGORITHM 1: Crawling AJAX

```
   input : URL, tags, browserType
1  Procedure MAIN()
2  begin
3      global $browser \leftarrow$ INITEMBEDDEDBROWSER($URL$, $browserType$)
4      global $robot \leftarrow$ INITROBOT()
5      global $sm \leftarrow$ INITSTATEMACHINE()
6      CRAWL(null)

7  Procedure CRAWL($State\ ps$)
8  begin
9      $cs \leftarrow sm$.GETCURRENTSTATE()
10     $\Delta update \leftarrow$ DIFF($ps$, $cs$)
11     $f \leftarrow$ ANALYSEFORMS($\Delta update$)
12     $Set\ C \leftarrow$ GETCANDIDATECLICKABLES($\Delta update$, $tags$, $f$)
13     for $c \in C$ do
14         $robot$.ENTERFORMVALUES($c$)
15         $robot$.FIREEVENT($c$)
16         $dom \leftarrow browser$.GETDOM()
17         if STATECHANGED($cs$.GETDOM(), $dom$) then
18             $xe \leftarrow$ GETXPATHEXPR($c$)
19             $ns \leftarrow sm$.ADDSTATE($dom$)
20             $sm$.ADDEDGE($cs$, $ns$, EVENT($c$, $xe$))
21             $sm$.CHANGETOSTATE($ns$)
22             if STATEALLOWEDTOBECRAWLED($ns$) then
23                 CRAWL($cs$)
24             $sm$.CHANGETOSTATE($cs$)
25             BACKTRACK($cs$)
```

element in the browser. The robot is currently capable of using either self-generated random values or custom values provided by the user to fill in the forms (for more details on the form-filling capabilities see [Mesbah and van Deursen 2009]).

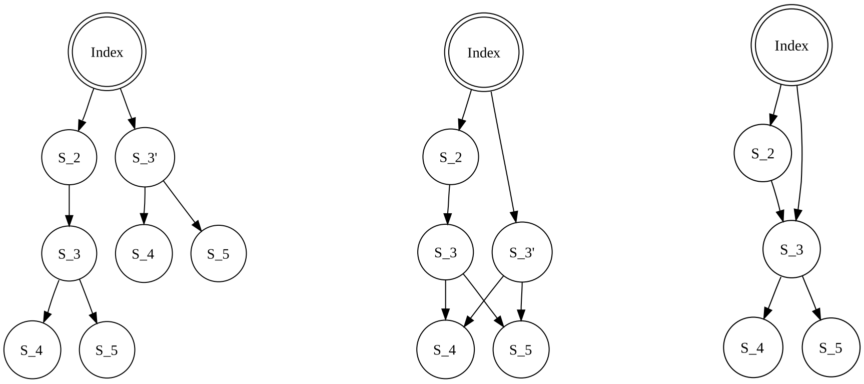
Definition 2. Let D_1 be the DOM tree of the state before firing an event e on a candidate clickable cc, and D_2 the DOM tree after e is fired, then cc is a clickable if and only if D_1 differs from D_2.

3.4. State Comparison

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 (line 17).

To detect a state change, the DOM trees need to be compared. One way of comparing them is by calculating the *edit distance* between two DOM trees, using the Levenshtein [1996] method. A similarity threshold τ is used, under which two DOM trees are considered clones. This threshold ($0.0 - 1.0$) can be given as input. 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.

Another way of comparing the states we have proposed in Roest et al. [2010], is the use of a series of *comparators*, each capable of focusing on and comparing specific as-



(a) Full state machine.

(b) Merging identical states.

(c) Merging similar states.

Fig. 4. State machine optimization.

pects of two DOM trees. In this technique, each comparator filters out specific parts of the DOM tree and passes the output to the next comparator. For instance, a `Datetime` comparator looks for any date/time patterns and filters those. This way, two states containing different timestamps can be automatically marked as similar.

If a state change is detected, according to our comparison heuristics, we create a new state and add it to the state-flow graph of the state machine (line 19).

The ADDSTATE call works as follows. In order to recognize an already met state, we compare every new state to the list of already visited states on the state-flow graph. If we recognize an identical or similar state in the state machine (based on the same similarity notion used for detecting a new state after an event), that state is used for adding a new edge, otherwise a new state is created and added to the graph.

As an example, Figure 4(a) shows the full state space of a simple application before any similarity comparison heuristics. In Figure 4(b), the states that are identical (*S_4*) are merged, and Figure 4(c) presents the state space after similar states (*S_3* and *S_3′*) have been merged.

For every detected state, a new edge is created on the graph between the state before the event and the current state (line 20). Using properties such as the XPATH expression, the clickable causing the state transition is also added as part of the new edge (line 18).

Moreover, the current state pointer of the state machine is updated to this newly added state at that moment (line 21).

3.5. Processing Document-Tree Deltas

After a clickable has been identified and its corresponding state transition created, the CRAWL procedure is recursively called (line 23) to find possible states reachable from the newly detected state.

Upon every new (recursive) entry into the CRAWL procedure, the first action taken (line 10) is computing the differences between the previous document tree and the current one, by means of an enhanced *Diff* algorithm [Chawathe et al. 1996; van Deursen 2007]. The resulting differences are used to find new candidate clickables, which are then further processed by the crawler. Such may be due, for example, to a server request call that injects new elements into the DOM.

ALGORITHM 2: Backtracking

```
   input :
1  Procedure BACKTRACK(State s)
2  begin
3      $cs \leftarrow s$
4      while cs.HASPREVIOUSSTATE() do
5          $ps \leftarrow$ cs.GETPREVIOUSSTATE()
6          if ps.HASUNEXAMINEDCANDIDATECLICKABLES() then
7              if browser.history.CANGOBACK() then
8                  browser.history.GOBACK()
9              else
10                 browser.RELOAD()
11                 List $E \leftarrow$ sm.GETPATHTO(ps)
12                 for $e \in E$ do
13                     $re \leftarrow$ RESOLVEELEMENT(e)
14                     robot.ENTERFORMVALUES(re)
15                     robot.FIREEVENT(re)
16             return
17         else
18             $cs \leftarrow ps$
```

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-level number can be defined to constrain the depth level of the recursive function.

3.6. Backtracking to the Previous State

Upon completion of the recursive call, the browser should be put back into the state it was in before the call, at least if there are still unexamined clickable elements left on that state.

Unfortunately, navigating (back and forth) through an AJAX site is not as easy as navigating a classical multi-page one. A dynamically changed DOM state does not register itself with the browser history engine automatically, so triggering the 'Back' function of the browser usually does not bring us to the previous state. Saving the whole browser state is also not feasible due to many technical difficulties. This complicates traversing the application when crawling AJAX. Algorithm 2 shows our backtracking procedure.

The backtracking procedure is called once the crawler is done with a certain state (line 25 in Algorithm 1). Algorithm 2 first tries to find the relevant previous state that still has unexamined candidate clickables (lines 4–18 in Algorithm 2).

If a relevant previous state is found to backtrack to (line 6 in Algorithm 2), then we distinguish between two situations.

Browser History Support. It is possible to programmatically register each state change with the browser history through frameworks such as the jQuery history/remote plugin[4], the Really Simple History library,[5] If an AJAX application has support for the browser history (line 7 in Algorithm 2), then to change the state in the browser, we can simply use the built-in history back functionality to move backwards (line 8 in Algorithm 2).

Clicking Through from Initial State. In case the browser history is not supported, which is the case with many AJAX applications currently, the approach we propose to get to a previous state is to save information about the clickable elements, the event type (e.g., `click`), and the order in which the events fired on the element results in reaching a particular state. Once we possess such information, we can reload the application (line 10 in Algorithm 2) and fire events on the clickable elements from the initial state to the desired state, using the exact path taken to reach that state (line 11 in Algorithm 2). However, as an optimization step, it is also possible to use Dijkstra's shortest path algorithm [Dijkstra 1959] to find the shortest element/event path on the graph to a certain state. For every element along the path, we first check whether the element can be found on the current DOM tree and try to resolve it using heuristics to find the best match possible (line 13 in Algorithm 2). Then, after filling in the related input fields (line 14 in Algorithm 2), we fire an event on the element (line 15 in Algorithm 2).

We adopt XPath, along with its attributes, to provide a better, more reliable, and persistent element identification mechanism. For each clickable, we reverse engineer the XPath expression of that element, which gives us its exact location on the DOM (line 18 in Algorithm 1). We save this expression in the state machine (line 20 in Algorithm 1) and use it to find the element after a reload, persistently (line 13 in Algorithm 2).

Figure 5 shows an example of how our backtracking mechanism operates. Lets assume that we have taken the `(E_1, E_3, E_4, E_5)` path and have landed on state `S_5`. From `S_5`, our algorithm knows that there are no candidate clickables left in states `S_4` and S_3, by keeping the track of examined elements. `S_1`, however, does contain an unexamined clickable element. The dotted line annotated with `1` shows our desired path for backtracking to this relevant previous state. To go from `S_5` to `S_1`, the algorithm reloads the browser so that it lands on the `index` state, and from there it fires an event on the clickable `E_1`.

4. CONCURRENT AJAX CRAWLING

The algorithm and its implementation for crawling AJAX, as just described, is sequential, depth-first, and single-threaded. Since we crawl the Web application dynamically, the crawling runtime is determined by the following factors.

(1) The speed at which the Web server responds to HTTP requests.
(2) Network latency.
(3) The crawler's internal processes (e.g., analyzing the DOM, firing events, updating the state machine).
(4) The speed of the browser in handling the events and request/response pairs, modifying the DOM, and rendering the user interface.

We have no influence on the first two factors and already have many optimization heuristics for the third step (See Section 3). Therefore, in this section we focus on the last factor, the browser. Since the algorithm has to wait some considerable amount of time for the browser to finish its tasks after each event, our hypothesis is that

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

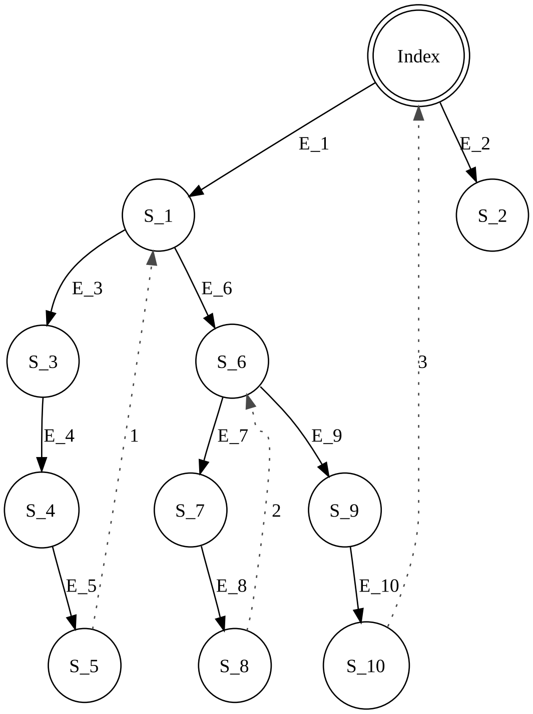


Fig. 5. Backtracking to the previous relevant state.

we can decrease the total runtime by adopting concurrent crawling through multiple browsers.

4.1. Multi-Threaded, Multi-Browser Crawling

Figure 6 shows the processing view of our concurrent crawling. The idea is to maintain a single state machine and split the original controller into a new controller and multiple *crawling nodes*. The controller is the single main thread monitoring the total crawl procedure. In this new setting, each crawling node is responsible for deriving its corresponding robot and browser instances to crawl a specific path.

Compared with Figure 3, the new architecture is capable of having multiple crawler instances, running from a single controller. All the crawlers share the same state machine. The state machine makes sure every crawler can read and update the state machine in a synchronized way. This way, the operation of discovering new states can be executed in parallel.

4.2. Partition Function

To divide the work over the crawlers in a multi-threaded manner, a partition function must be designed. The performance of a concurrent approach is determined by the quality of its partition function [Garavel et al. 2001]. A partition function can be either static or dynamic. With a static partition function, the division of work is known in advance, before executing the code. When a dynamic partition function is used, the decision of which thread will execute a given node is made at runtime. Our algorithm infers the state-flow graph of an AJAX application dynamically and incrementally. Thus, due to this dynamic nature, we adopt a dynamic partition function.

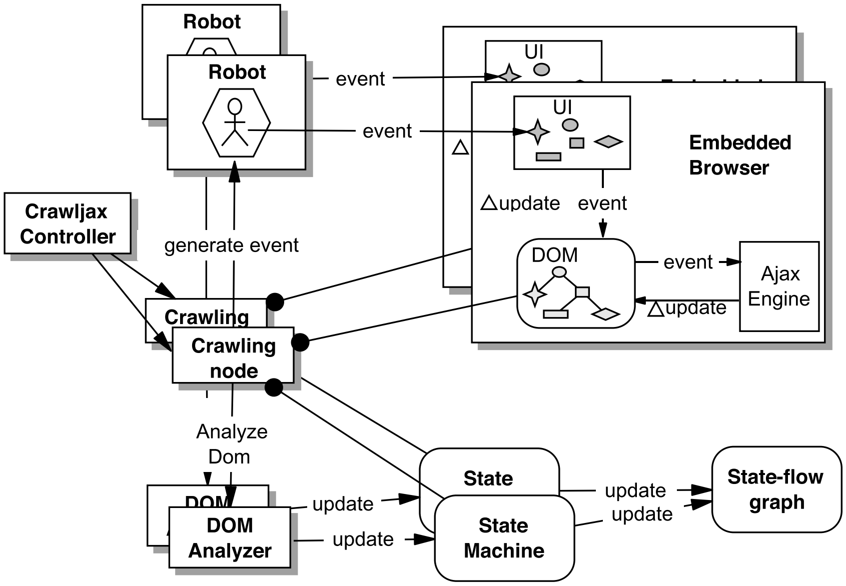


Fig. 6. Processing view of the concurrent crawling architecture.

The task of our dynamic partition function is to distribute the work equally over all the participating crawling nodes. While crawling an AJAX application, we define *work* as bringing the browser back into a given state and exploring the first unexplored candidate state from that state. Our proposed partition function operates as follows. After the discovery of a new state, if there are still unexplored candidate clickables left in the previous state, that state is assigned to another thread for further exploration. The processor chosen will be the one with the least amount of work left.

Figure 7 visualizes our partition function for concurrent crawling of a simple Web application. In the Index state, two candidate clickables are detected that can lead: `S_1` and `S_11`. The initial thread continues with the exploration of the states `S_1`, `S_2`, `S_3`, `S_4`, and finishes in `S_5`, in a depth-first manner. Simultaneously, a new thread is branched off to explore state `S_11`. This new thread (`thread #2`) first reloads the browser to `Index` and then goes into `S_11`. In state `S_2` and `S_6`, this same branching mechanism happens, which results in a total of five threads.

Now that the partition function has been introduced, the original sequential crawling algorithm (Algorithm 1) can be changed into a concurrent version.

4.3. The Concurrent Crawling Algorithm

The concurrent crawling approach is shown in Algorithm 3. Here we briefly explain the main differences with respect to the original sequential crawling algorithm, as presented in Algorithm 1 and discussed in Section 3.

Global State-flow Graph. The first change is the separation of the state-flow graph from the state machine. The graph is defined in a global scope (line 3), so that it can be centralized and used by all concurrent nodes. Upon the start of the crawling process, an initial crawling node is created (line 5) and its RUN procedure is called (line 6).

Browser Pool. The robot and state machine are created for each crawling node. Thus, they are placed in the local scope of the RUN procedure (lines 10–11).

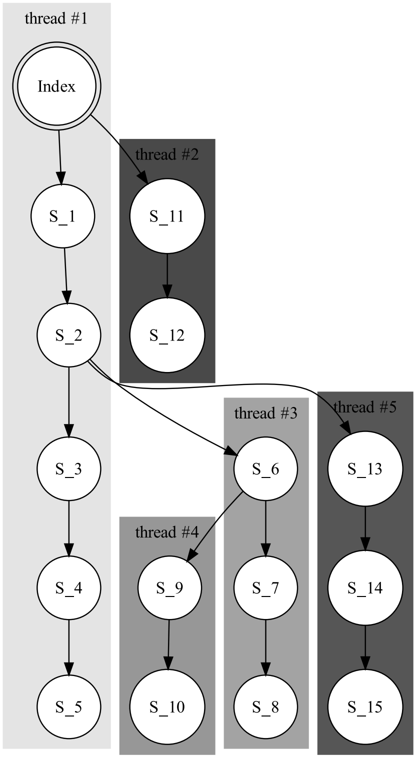


Fig. 7. Partitioning the state space for multi-threaded crawling.

Generally, each node needs to acquire a browser instance, and after the process is finished, the browser is killed. Creating new browser instances is a process-intensive and time-consuming operation. To optimize, a new structure is introduced: the `BrowserPool` (line 4), which creates and maintains browsers in a pool of browsers to be reused by the crawling nodes. This reduces start-up and shut-down costs. The `BrowserPool` can be queried for a browser instance (line 9), and when a node is finished working, the browser used is released back to the pool.

In addition, the algorithm now takes the desired number of browsers as input. Increasing the number of browsers used can decrease the crawling runtime, but it also comes with some limitations and trade offs that we will discuss in Section 6.5.

Forward-Tracking. In the sequential algorithm, after finishing a crawl path, we need to bring the crawler to the previous (relevant) state. In the concurrent algorithm, however, we create a new crawling node for each path to be examined (see Figure 7). Thus, instead of bringing the crawler back to the desired state (backtracking), we must take the new node forward to the desired state, hence, forward-tracking.

This is done after the $browser$ is pointed to the `URL` (line 12). The first time the RUN procedure is executed, no forward-tracking is taking place, since the event-path (i.e., the list of clickable items resulting to the desired state) is empty, so the initial crawler starts from the `Index` state. However, if the event path is not empty, the clickables are used to take the browser forward to the desired state (lines 13–16). At that point, the CRAWL procedure is called (line 17).

ALGORITHM 3: Concurrent AJAX Crawling

```
input : URL, tags, browserType, nrOfBrowsers
Procedure MAIN()
begin
    global $sfg \leftarrow$ INITSTATEFLOWGRAPH()
    global $browserPool \leftarrow$ INITBROWSERPOOL($nrOfBrowsers, browserType$)
    $crawlingNode \leftarrow$ CRAWLINGNODE()
    $crawlingNode$.RUN($null, null$)

Procedure RUN(State s, EventPath ep)
begin
    $browser \leftarrow browserPool$.GETEMBEDDEDBROWSER()
    $robot \leftarrow$ INITROBOT()
    $sm \leftarrow$ INITSTATEMACHINE($sfg$)
    $browser$.GOTO($URL$)
    for $e \in ep$ do
        $re \leftarrow$ RESOLVEELEMENT($e$)
        $robot$.ENTERFORMVALUES($re$)
        $robot$.FIREEVENT($re$)
    CRAWL($s, browser, robot, sm$)


Procedure CRAWL(State ps, EmbeddedBrowser browser, Robot robot, StateMachine sm)
begin
    $cs \leftarrow sm$.GETCURRENTSTATE()
    $\Delta update \leftarrow$ DIFF($ps, cs$)
    $f \leftarrow$ ANALYSEFORMS($\Delta update$)
    Set $C \leftarrow$ GETCANDIDATECLICKABLES($\Delta update, tags, f$)
    for $c \in C$ do
        SYNCH($c$)
        begin
            if $cs$.NOTEXAMINED($c$) then
                $robot$.ENTERFORMVALUES($c$)
                $robot$.FIREEVENT($c$)
                $cs$.EXAMINED($c$)
                $dom \leftarrow browser$.GETDOM()
                if STATECHANGED($cs$.GETDOM(), $dom$) then
                    $xe \leftarrow$ GETXPATHEXPR($c$)
                    $ns \leftarrow sm$.ADDSTATE($dom$)
                    $sm$.ADDEDGE($cs, ns,$ EVENT($c, xe$))
                    $sm$.CHANGETOSTATE($ns$)
                    PARTITION($cs$)
                    if STATEALLOWEDTOBECRAWLED($ns$) then
                        CRAWL($cs$)
                    $sm$.CHANGETOSTATE($cs$)

Procedure PARTITION(State cs)
begin
    while SIZEOF($cs$.NOTEXAMINEDCLICKABLES()) $> 0$ do
        $crawlingNode \leftarrow$ CRAWLINGNODE($cs$, GETEXACTPATH())
        DISTRIBUTEPARTITION($crawlingNode$)
```

Crawling Procedure. The first part of the CRAWL procedure is unchanged (lines 21–24). To enable concurrent nodes accessing the candidate clickables in a thread-safe manner, the body of the `for` loop is synchronized around the candidate element to be examined (line 26). To avoid examining a candidate element multiple times by multiple nodes, each node first checks the examined state of the candidate element (line 28). If the element has not been examined previously, the robot executes an event on the element in the browser and sets its state as `examined` (line 31). If the state is changed, before going into the recursive CRAWL call, the PARTITION procedure is called (line 38).

Partition Procedure. The partition procedure, called on a particular state `cs` (line 44), creates a new crawling node for every unexamined candidate clickable in `cs` (line 46). The new crawlers are initialized with two parameters, namely, (1) the current state `cs`, and (2) the execution path from the initial `Index` state to this state. Every new node is distributed to the work queue participating in the concurrent crawling (line 47). When a crawling node is chosen from the work queue, its corresponding RUN procedure is called in order to spawn a new crawling thread.

5. TOOL IMPLEMENTATION

We have implemented the crawling concepts in a tool called CRAWLJAX. The development of CRAWLJAX originally started in 2007. There have been many extension and improvement iterations since the first release in 2008. CRAWLJAX has been used by various users and applied to a range of industrial case studies. It is released under the Apache open source license and is available for download. In 2010 alone, the tool was downloaded more than 1,000 times. More information about the tool can be found on `http://crawljax.com`.

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 currently supports three browsers (IE, Chrome, and Firefox) and has been implemented on top of the Selenium 2.0 (WebDriver) APIs.[6] The state-flow graph is based on the JGrapht[7] library.

CRAWLJAX has a `Plugin`-based architecture. There are various extension points for different phases of the crawling process. The main interface is `Plugin`, which is extended by the various types of plugins available. Each plugin interface serves as an extension point that is called in a different phase of the crawling execution, for example, `preCrawlingPlugin` runs before the crawling starts, `OnNewStatePlugin` runs when a new state is found during crawling, and `PostCrawlingPlugin` runs after the crawling is finished. More details of the plugin extension points can be found on the project homepage.[8] There is a growing list of plugins available for CRAWLJAX,[9] examples of which include a static mirror generator, a test suite generator, a crawl overview generator for visualization of the crawled states, a proxy to intercept communication between client/server while crawling, and a cross-browser compatibility tester.

Through an API (`CrawljaxConfiguration`), the user is able to configure many crawling options, such as the elements that should be examined during crawling (e.g., clicked on), elements that should be ignored (e.g., logout), crawling depth and time,

[6] `http://code.google.com/p/selenium/wiki/GettingStarted`
[7] `http://jgrapht.sourceforge.net`
[8] `http://crawljax.com/documentation/writing-plugins/`
[9] `http://crawljax.com/plugins/`

Table I. Experimental Benchmarks and Examples of their Clickable Elements

Case	AJAX site	Sample 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="#">2nd 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 onclick="Shop.selectSort(this); return false" class="booties" href="#">booties</a> <div id="thumbnail_7" class="thumbnail highlight"><img src="...001_thumb.jpg" /><div <div class="darkening">...</div>

the maximum number of states to examine, the state comparison method, the plugins to be used, and the number of desired browsers that should be used during crawling.

6. EVALUATION

Since 2008, we and others have used CRAWLJAX for a series of crawling tasks on different types of systems. In this section, we provide an empirical assessment of some of the key properties of our crawling technique. In particular, we address the accuracy (are the results correct?), scalability (can we deal with realistic sites?), and performance, focusing in particular on the performance gains resulting from concurrent crawling.

We first present our findings concerning accuracy and scalability, for which we study six systems, described next (Section 6.1). Or analyze the performance gains from concurrent crawling, we apply CRAWLJAX to Google’s ADSENSE application (Section 6.5).

6.1. Subject Systems

In order to assess the accuracy (Section 6.3) and scalability (Section 6.4), we study the six systems C1–C6 listed in Table I. For each case, we show the site under study, as well as a selection of typical clickable elements. We selected these sites because they

Table II. Results of Running CRAWLJAX on Six AJAX Applications

Case	DOM string size (byte)	Candidate Clickables	Detected Clickables	Detected States	Crawl Time (s)	Depth	Tags
C1	4590	540	16	16	14	3	A, DIV, SPAN, IMG
C2	24636	1813	33	34	26	2	A, IMG
C3	262505	150	148	148	498	1	A
		19247	1101	1071	5012	2	A, TD
C4	40282	3808	55	56	77	2	A, DIV, INPUT, IMG
C5	165411	267	267	145	806	1	A
		32365	1554	1234	6436	2	A, DIV
C6	134404	6972	83	79	701	1	A, DIV

adopt 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 tree.

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 a case in which we are in full control of the AJAX features used, allowing us to introduce different types of dynamically set clickables as shown in Figure 1 and Table I.

Our second case, C2, is Sun's, Ajaxified PETSTORE 2.0,[10] which is built on 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 real-world external public AJAX applications. Thus, we have no access to their source code. C4 is an AJAX-based application 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 applications with numerous clickables and dynamic states.

6.2. Applying CRAWLJAX

The results of applying CRAWLJAX to C1–C6 are displayed in Table II. The table lists key characteristics of the sites under study, such as the average DOM size and the total number of candidate clickables. Furthermore, it lists the key configuration parameters set, most notably the tags used to identify candidate clickables and the maximum crawling depth.

The performance measurements were obtained on a a laptop with Intel Pentium M 765 processor 1.73GHz, 1GB RAM, and Windows XP.

6.3. Accuracy

6.3.1. Experimental Setup. Assessing the correctness of the crawling process is challenging for two reasons. First, there is no strict notion of "correctness" with respect to state equivalence. The state comparison operator part of our algorithm (see

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

Section 3.4) can be implemented in different ways: the more states it considers equal, the smaller and the more abstract the resulting state-flow graph is. The desirable level of abstraction depends on the intended use of the crawler (regression testing, program comprehension, security testing, to name a few) and the characteristics of the system being crawled.

Second, no other crawlers for AJAX are available, making it impossible to compare our results to a "gold standard." Consequently, an assessment in terms of *precision* (percentage of correct states) and *recall* (percentage of states recovered) is impossible to give.

To address these concerns, we proceed as follows. For the cases in which we have full control—C1 and C2—we inject specific clickable elements.

- For C1, 16 elements were injected, out of which 10 were on the top-level index page. Furthermore, to evaluate the state comparison procedure, we intentionally introduced a number of identical (clone) states.
- For C2, we focused on two product categories, CATS and DOGS, from the five available categories. We annotated 36 elements (product items) by modifying the JAVASCRIPT method, which turns the items retrieved from the server into clickables on the interface.

Subsequently, we manually create a reference model, to which we compare the derived state-flow graph.

To assess the four external sites C3–C6, we inspect a selection of the states. For each site, we randomly select ten clickables in advance, by noting their tag names, attributes, and XPath expressions. After crawling of each site, we check the presence of these ten elements among the list of detected clickables.

In order to do the manual inspection of the results, we run CRAWLJAX with the *Mirror* plugin enabled. This post-crawling plugin creates a static mirror, based on the derived state-flow graph, by writing all DOM states to file and replacing edges with appropriate hyperlinks.

6.3.2. Findings. Our results are as follows.

- For C1, all 16 expected clickables were correctly identified, leading to a precision and recall of 100% for this case. Furthermore, the clone states introduced were correctly identified as such.
- For C2, 33 elements were detected correctly from the annotated 36. The three elements that were not detected turn out to be invisible elements requiring multiple clicks on a scroll bar to appear. Since our default implementation avoids clicking the same element multiple times (see Section 3.5), these invisible elements cannot become visible. Hence, they cannot be clicked in order to produce the required state change when the default settings are used. Note that multiple events on the same element is an option supported in the latest version of CRAWLJAX.
- For C3–C6, 38 out of the $4 * 10 = 40$, corresponding to 95% of the states, were correctly identified. The reason for not creating the missing two states is similar to the C2-case: the automatically derived navigational flow did not permit reaching the two elements that had to be clicked, in order to generate the required states.

Based on these findings, we conclude that (1) states detected by CRAWLJAX are correct; (2) duplicate states are correctly identified as such, and but that (3) not all states are necessarily reached.

6.4. Scalability

6.4.1. Experimental Setup. In order to obtain an understanding of the scalability of our approach, we measure the time needed to crawl, as well as a number of site characteristics that will affect the time needed. We expect the crawling performance to be directly proportional to the input size, which is composed of (1) the average DOM string size, (2) number of candidate elements, and (3) number of detected clickables and states, which are the characteristics that we measure for the six cases.

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.

6.4.2. Findings. Concerning the time needed to crawl the internal sites, we see that it takes CRAWLJAX 14 and 26 seconds to crawl C1 and C2, respectively. The average DOM size in C2 is five times bigger, and the number of candidate elements is three times higher.

In addition to this increase in DOM size and in the number of candidate elements, the C2 site does not support the browser's built-in Back method. Thus, as discussed in Section 3.6, 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 naturally has a negative effect on the performance.

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, which increases the performance measurement numbers in our experiment.

CRAWLJAX was able to run smoothly on the external sites. Except a few minor adjustments (see Section 7) we did not witness any difficulties. C3 with depth level 2 was crawled successfully in 83 minutes, resulting in 19,247 examined candidate elements, 1,101 detected clickables, and 1,071 detected states. For C5, CRAWLJAX was able to finish the crawl process in 107 minutes on 32,365 candidate elements, resulting in 1,554 detected clickables, and 1,234 states. As expected, in both cases, increasing the depth level from 1 to 2 greatly expands the state space.

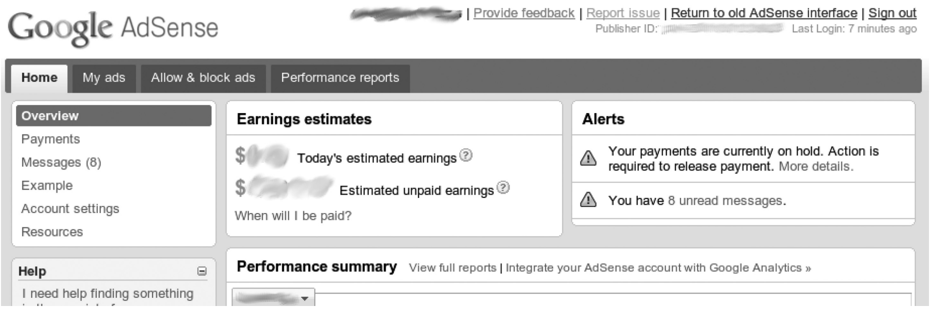
6.5. Concurrent Crawling

In our final experiment, the main goal is to assess the influence of the concurrent crawling algorithm on the crawling runtime.

6.5.1. Experimental Object. Our experimental object for this study is Google ADSENSE[11], an AJAX application developed by Google, which empowers online publishers to earn revenue by displaying relevant ads on their Web content. The ADSENSE interface is built using GWT (Google Web Toolkit) components and is written in Java.

Figure 8 shows the index page of ADSENSE. On the top, there are four main tabs (`Home`, `My ads`, `Allow & block ads`, `Performance reports`). On the top left side, there is a box holding the anchors for the current selected tab. Underneath the left-menu box, there is a box holding links to help-related pages. On the right of the left-menu we can see the main contents, which are loaded by AJAX calls.

[11] https://www.google.com/adsense/



[redacted] | Provide feedback | Report issue | Return to old AdSense interface | Sign out
Publisher ID: [redacted] Last Login: 7 minutes ago

$[redacted] Today's estimated earnings

$[redacted] Estimated unpaid earnings

When will I be paid?

Your payments are currently on hold. Action is required to release payment. More details.

You have 8 unread messages.

I need help finding something

View full reports | Integrate your AdSense account with Google Analytics »

[redacted]

Fig. 8. Google ADSENSE.

6.5.2. Experimental Design. Our research questions can be presented as follows.

RQ1. Does our concurrent crawling approach positively influence the performance?
RQ2. Is there a limit on the number of browsers that can be used to reduce the runtime?

Based on these two research questions, we formulate our two null hypotheses as follows.

$H1_0$. The availability of more browsers does not impact the time needed to crawl a given AJAX application.
$H2_0$. There is no limit on the number of browsers that can be added to reduce the runtime.

The alternative hypotheses that we use in the experiment are the following.

H1. The availability of more browsers reduces the time needed to crawl a given AJAX application.
H2. There is a limit on the number of browsers that can be added to reduce the runtime.

Infrastructure. To derive our experimental data, we use the Google infrastructure which offers the possibility of runnning our experiments either on a local workstation or on a distributed testing cluster.

On the distributed testing cluster, the number of cores varies between clusters. Newer clusters are supplied with six or eight core CPU's, while older clusters include two or four cores. The job distributor ensures a minimum of two GB of memory per job. The cluster is shared between all development teams, so our experiment data was gathered while other teams were also using the cluster. To prevent starvation on the distributed testing cluster, a maximum runtime of one hour is specified, that is, any test running longer than an hour is killed automatically.

To achieve a repeatable experiment, we initiate a new ADSENSE front end with a clean database server for every experimental test. The test data is loaded into the database during the initialization phase of the ADSENSE front end.

Tool Configuration. To crawl ADSENSE, we configured CRAWLJAX 2.0 to click on all anchor-tags and fill in form inputs with custom data.

To inform the user that the interface is being updated, ADSENSE displays a loading-icon. While crawling, to determine whether the interface was finished with loading the content after each fired event, CRAWLJAX analyzed the DOM tree to check for this icon.

Table III. Descriptive Statistics of the Runtime (in Minutes) for the 10 Categories of Browsers

	N	Mean	Std. Dev.	Std. Err.	Lower Bound*	Upper Bound*	Min	Max
1	2953	7.4871	3.70372	.06816	7.3535	7.6207	2.33	22.15
2	299	5.4721	1.14160	.06602	5.3422	5.6020	2.33	10.55
3	295	5.5521	1.26716	.07378	5.4069	5.6973	3.38	11.83
4	297	5.6404	1.11101	.06447	5.5135	5.7673	3.78	11.12
5	291	5.6744	1.15004	.06742	5.5417	5.8070	3.69	10.75
6	290	5.6920	.99718	.05856	5.5767	5.8072	4.20	11.01
7	294	5.9806	1.05696	.06164	5.8593	6.1020	4.13	10.72
8	297	6.2338	1.03936	.06031	6.1151	6.3525	4.74	10.32
9	292	7.4651	1.04823	.06134	7.3444	7.5858	5.35	10.56
10	299	9.9243	1.28598	.07437	9.7779	10.0707	7.05	13.37
Total	299	17.0613	2.31540	.13390	16.7978	17.3249	10.17	22.15

Note: * 95% Confidence Interval for Mean.

Due to the infrastructure, we were restricted to using Linux as our operating system, and we chose Firefox 3.5 as our embedded browser.

Variables and Analysis. The *independent variable* in our experiment is the number of browsers used for crawling. We use the same crawl configuration for all the experiments. The only property that changes is the number of browsers used during crawling, from 1–10.

The *dependent variable* that we measure is the time needed to crawl the given crawl specification, calculated from the start of the crawling until the (last) browser finishes. To compare, we also measure the actual number of examined clickables, crawled states, edges, and paths.

We run every experiment multiple times and take the average of the runtime. On the distributed cluster all resources are shared. Hence, to get reliable data we executed every experiment 300 times.

Since we have ten independent samples of data with 2,953 data points, we use the One-Way ANOVA statistical method to test the first hypothesis ($H1_0$). Levene's test is used for checking the homogeneity of variances. Welch and Brown-Forsythe are used as tests of equality of means [Maxwell and Delaney 2004].

If $H1_0$ turns out to be true, we proceed with our second hypothesis. To test $H2_0$, we need to compare the categories to find out which are responsible for runtime differences. Thus, we use the Post Hoc Tukey test if the population variances are equal, otherwise, the Games-Howell test. We use SPSS[12] for the statistical analysis and R[13] for plotting the graphs.

6.5.3. Results and Evaluation. We present our data and analysis on the data from the distributed infrastructure. We obtained similar results with different configurations. Our experimental data can be found on the following link.[14]

Figures 9, 10, and 11 depict boxplots of the detected states (Figure 9), detected edges (Figure 10), and runtime (minutes) (Figure 11), respectively, versus the number of browsers used during crawling, on the distributed infrastructure. The number of

[12] `http://www.spss.com`

[13] `http://www.r-project.org`

[14] `http://tinyurl.com/3d5km3b`

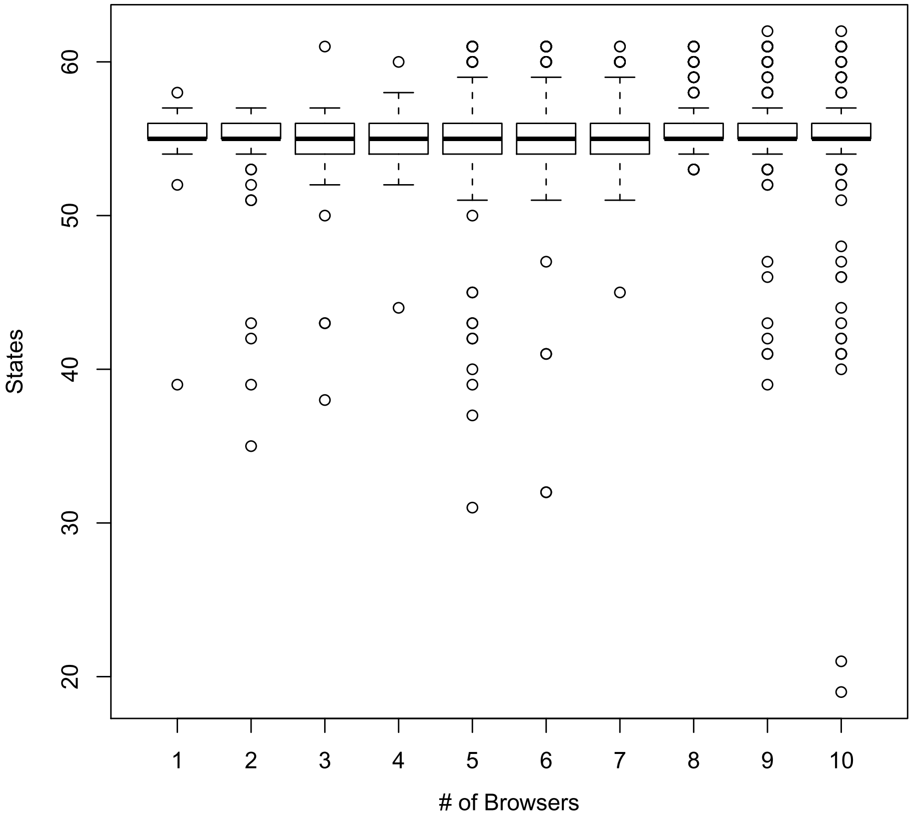


Fig. 9. Boxplots of detected states versus the number of browsers. Distributed setting (300 measurements for each category of browsers).

detected states and edges is constant, which means our multi-browser crawling and state exploration is stable.

Figure 11 shows that there is a decrease in the runtime when the number of browsers is increased. Table III presents the descriptive statistics of the runtime for the ten categories of browsers.

Table IV shows the main ANOVA results. The significance value comparing the groups is < .05. The significance value for homogeneity of variances, as shown in Table V, is < .05, which means the variances are significantly different. The Welch and Brown-Forsythe are both 0, so we can reject the first null hypothesis. Thus, we can conclude that our concurrent crawling approach positively influences the performance.

To test for the second hypothesis, we need to compare the groups to find out if the differences between them is significant. Table VI shows our post hoc test results. A "$*$" means that the difference in runtime is significant. It is evident that there is a limit on the number of browsers that can significantly decrease the runtime. The optimal number for our ADSENSE study is five browsers. By increasing the number of browsers from one to five, we can achieve a decrease of up to 65% in runtime. Increasing the number of browsers beyond five has no significant influence on the runtime in our current impementation.

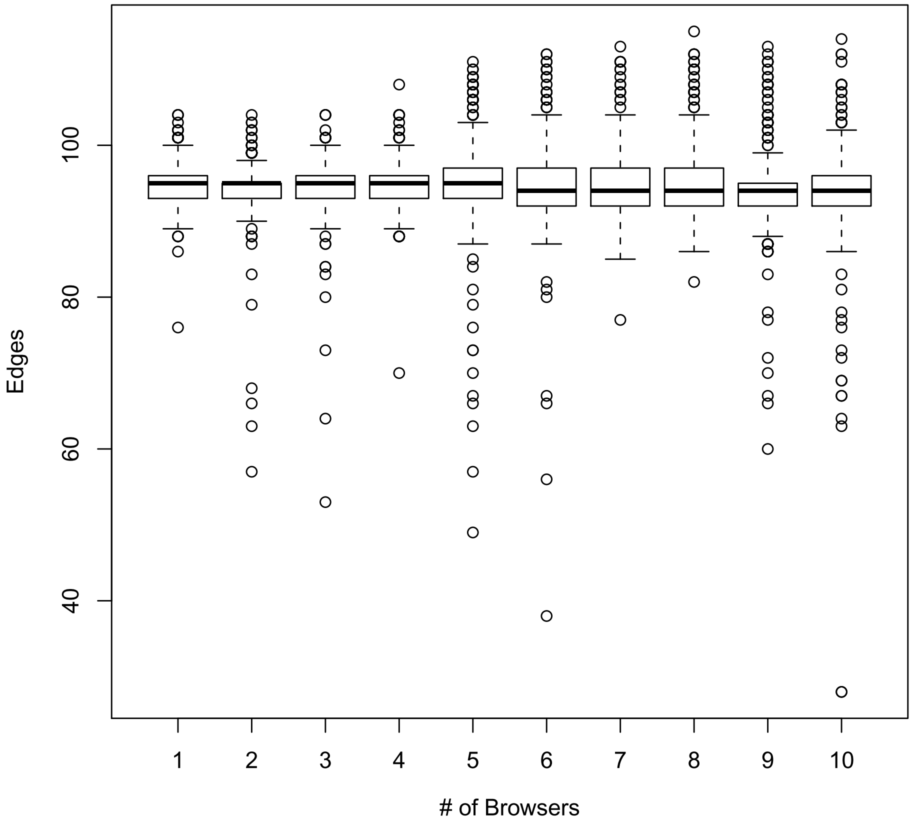


Fig. 10. Boxplots of detected edges vs. the number of browsers. Distributed setting (300 measurements for each category of browsers).

6.6. Threats to Validity

As far as the repeatability of the studies is concerned, CRAWLJAX is open source and publicly available for download. The experimental applications in Section 6.1 are composed of open source and public domain websites. In the concurrent crawling experiment (Section 6.5), the study was done at Google using Google's ADSENSE, which is also publicly accessible. More case studies are required to generalize the findings on correctness and scalability. One concern with using public domain Web applications as benchmarks is that they can change and evolve over time, making the results of the study irreproducible in the future.

7. DISCUSSION

7.1. Detecting DOM Changes

An interesting observation in C2, in the beginning of the experiment, was that every examined candidate element was detected as a clickable. Further investigation revealed that this phenomenon was caused by a piece of JAVASCRIPT code (banner) which constantly changed the DOM tree with textual notifications. Hence, every time a DOM comparison was done, a change was detected. We had to use a higher similarity threshold so that the textual changes caused by the banner were not seen as a relevant state change for detecting clickables. In CRAWLJAX, it is also possible to ignore certain parts of the DOM tree, for instance, regular expressions that capture

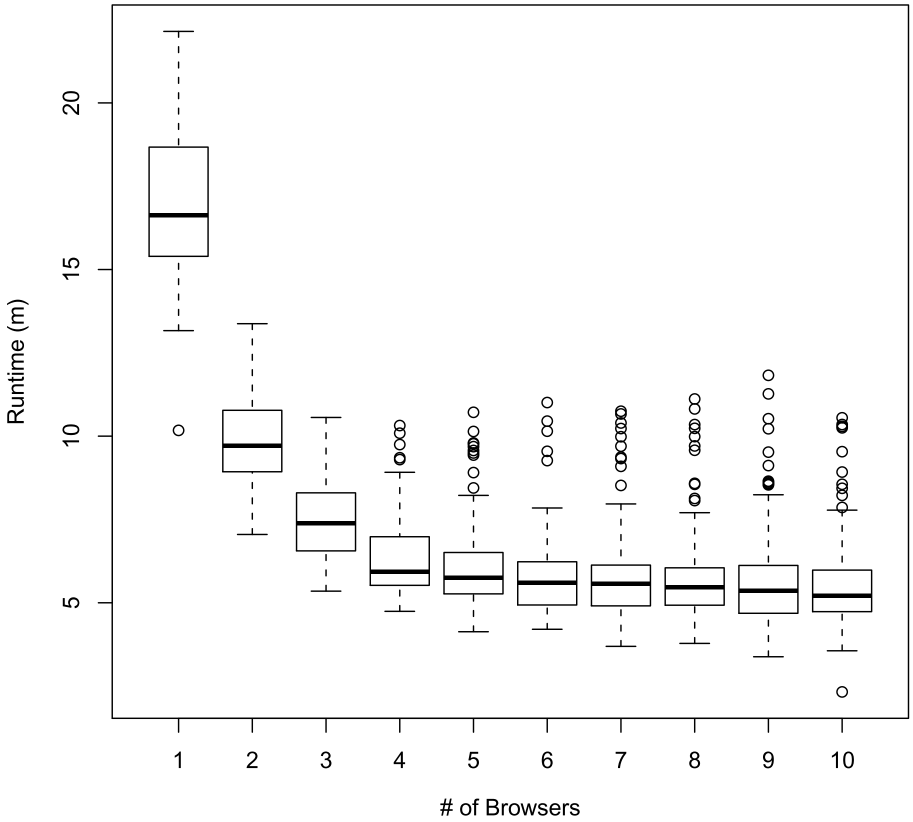


Fig. 11. Boxplots of runtime vs. the number of browsers. Distributed setting (300 measurements for each category of browsers).

Table IV. One-Way ANOVA

	Sum of Squares	**df**	**Mean Square**	**F**	**Sig.**
Between Groups	35,540.225	9	3,948.914	2345.919	000
Within Groups	4,953.987	2,943	1.683		
Total	40,494.212	2,952			

the recurring patterns. How the notion of a dynamic state change is defined can potentially influence the crawling behaviour. The automatic crawler ignores subtle changes in the DOM that we believe are not of significant importance (such as case sensitivity and time stamps). We also provide the user with different mechanisms for defining their own notion of state similarity. Push-based techniques, such as Comet [Russell 2006], in which data is constantly pushed from the server, could also cause state comparison challenges. Such push-based updates are usually confined to a specific part of the DOM tree, and hence, can be controlled using custom DOM change filters.

7.2. Back and Forward Tracking

Because of side effects of back-end state changes, there is no guarantee that we reach the exact same state when we traverse a click path a second time. This nondeterminism characteristic is inherent in dynamic Web applications. Our crawler uses the

Table V. Tests for Homogeneity of Variances and Equality of Means

Method	Statistic	df1	df2	Sig.
Levene	55.884	9	2,943	0.000
Welch	1,060.017	9	1,198.048	0.000
Brown-Forsythe	2,355.528	9	1,914.845	0.000

Table VI. Post Hoc Games-Howell Multiple Comparisons

—	1	2	3	4	5	6	7	8	9	10
1	—	*	*	*	*	*	*	*	*	*
2	*	—	*	*	*	*	*	*	*	*
3	*	*	—	*	*	*	*	*	*	*
4	*	*	*	—	—	*	*	*	*	*
5	*	*	*	—	—	*	*	*	*	*
6	*	*	*	*	*	—	—	—	—	—
7	*	*	*	*	*	—	—	—	—	—
8	*	*	*	*	*	—	—	—	—	—
9	*	*	*	*	*	—	—	—	—	—
10	*	*	*	*	*	—	—	—	—	—

Note: * Indicates the Mean Difference is Significant at the 0.05 Level.

notion of state similarity, thus as long as the revisited state is similar to the state visited before, the crawling process continues without side effects. In our experiments, we did not encounter any problems with this nondeterministic behaviour.

When the crawling approach is used for testing Web applications, one way to ensure that a state revisited is the same as the state previously visited (e.g., for regression testing [Roest et al. 2010]), is by bringing the server-side state to the previous state as well, which could be challenging. More research is needed to adopt ways of synchronizing the client- and server-side state during testing.

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. The new features of HTML5, such as Web storage [W3C], could possibly cause the same problems by making parts of the client-side state persistent between backtracking sessions. It would be interesting future work to explore ways to get around these issues.

7.3. State Space

The set of found states and the inferred state machine are not complete, that is, CRAWLJAX creates an instance of the state machine of the AJAX application but not necessarily *the* instance. Any crawler can only crawl and index a snapshot instance of a dynamic Web application in a given point of time. The order in which clickables are chosen could generate different states. Even executing the same clickable twice from a 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* [Valmari 1998]. Just like 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 (i.e., different domains) and links that match some predefined set of regular expressions, for example, mail:*, *.ps, and *.pdf.

The current implementation of CRAWLJAX keeps the DOM states in the memory. As an optimization step, next to the multi-browser crawling, we intend to abstract and serialize the DOM state into the file system and only keep a reference in the memory. This saves much space in the memory and enables us to handle many 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 file system when needed, in a later stage.

7.4. DOM Settling

Determining when a DOM is fully loaded into the browser after a request or an event is a very difficult task. Partially loaded DOM states can adversely influence the state-exploration accuracy during crawling. The asynchronous nature of AJAX calls and the dynamic DOM updates make the problem even more challenging to handle. Since the major browsers currently do not provide APIs for determining when a DOM tree is fully loaded, we wait a specific amount of time after each event or page reload. This waiting time can be adjusted by the user through the CRAWLJAX configuration API. By choosing a high enough waiting time, we can be certain that the DOM is fully settled in the browser. As a side effect, a too-high waiting period can make the crawling process slow.

An more reliable alternative way is when the Web application provides a completion flag in the form of a DOM element, either visible or invisible to the end user. Before continuing with its crawling operations, CRAWLJAX can be configured to wait for that specific DOM flag to appear after each state transition. This flag-based waiting approach is, in fact, what we used during our experiments on ADSENSE.

7.5. Applications of CRAWLJAX

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

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 expose the hidden-web content induced by JAVASCRIPT, in general, and AJAX, in particular.

In their proposal for making AJAX applications crawlable,[15] Google proposes using URLs containing a special hash fragment, that is, `#!`, for identifying dynamic content. Google then uses this hash fragment to send a request to the server. The server has to treat this request in a special way and send an HTML snapshot of the dynamic content, which is then processed by Google's crawler. In the same proposal, they suggest using CRAWLJAX for creating a static snapshot for this purpose. Web developers can use the model inferred by CRAWLJAX to automatically generate a static HTML snapshot of their dynamic content, which then can be served to Google for indexing.

The ability to automatically detect and exercise the executable elements of an AJAX site and navigate between the various dynamic states gives us a powerful Web-analysis and test-automation mechanism. In the recent past, we have applied CRAWLJAX in the following Web-testing domains.

(1) Invariant-based testing of AJAX user interfaces [Mesbah and van Deursen 2009],
(2) Spotting security violations in Web widget interactions [Bezemer et al. 2009]

[15] `http://code.google.com/web/ajaxcrawling/docs/getting-started.html`

(3) Regression testing of dynamic and nondeterministic Web interfaces [Roest et al. 2010],
(4) Automated cross-browser compatibility testing [Mesbah and Prasad 2011].

8. RELATED WORK

Web crawlers, also known as Web spiders and (ro)bots, have been studied since the advent of the Web itself [Brin and Page 1998; Burner 1997; Cho et al. 1998; Heydon and Najork 1999; Pinkerton 1994].

More recently, there has been extensive research on the hidden Web behind forms [Barbosa and Freire 2007; Dasgupta et al. 2007; de Carvalho and Silva 2004; Lage et al. 2004; Madhavan et al. 2008; Ntoulas et al. 2005; Raghavan and Garcia-Molina 2001]. The main focus in this research area is to detect ways of accessing the Web content behind data entry points.

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.

Alvarez et al. [2004, 2006] discuss some challenges of crawling hidden content generated with JAVASCRIPT, but focus on hypertext links.

To the best of our knowledge, our initial work on CRAWLJAX [Mesbah et al. 2008] in 2008 was the first academic research work proposing a solution to the problem of crawling AJAX, in the form of algorithms and an open-source tool that automatically crawls and creates a finite state machine of the states and transitions.

In 2009, Duda et al. [2009] discussed how AJAX states could be indexed. The authors present a crawling and indexing algorithm. Their approach also builds finite state models of AJAX applications; however, there is no accompanying tool available for comparison. The main difference between their algorithm and ours seems to be in the way clickable elements are detected, which is through JAVASCRIPT analysis.

The work of Memon et al. [2001, 2003] 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 also exists a large body of knowledge targeting challenges in parallel and distributed computing. Specifically for the Web, Cho and Garcia-Molina [2002] discuss the challenges of parallel crawling and propose an architecture for parallel crawling the classical Web. Boldi et al. [2004] present the design and implementation of UbiCrawler, a distributed Web crawling tool. Note that these works are URL-based and, as such, not capable of targeting event-based AJAX applications.

9. CONCLUDING REMARKS

Crawling modern AJAX-based Web systems require a different approach than the traditional way of extracting hypertext links from webpages and sending requests to the server.

This article proposes an automated crawling technique for AJAX-based Web applications, which is based on dynamic analysis of the client-side Web user interface in embedded browsers. The main contributions of our work are the following.

- An analysis of the key challenges involved in crawling AJAX-based applications;
- A systematic process and algorithm to drive an AJAX application and infer a state machine from the detected state changes and transitions. Challenges addressed include the identification of clickable elements, the detection of DOM changes, and the construction of the state machine;
- A concurrent multi-browser crawling algorithm to improve the runtime performance;
- The open-source tool called CRAWLJAX, which implements the crawling algorithms;

— Two studies, including seven AJAX applications, 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.

The fact that the tool is freely available for download will help to identify exciting case studies. Furthermore, strengthening the tool by extending its functionality, improving the accuracy, performance, and the state explosion algorithms are directions we foresee for future work. We will conduct controlled experiments to systematically analyze and find new ways of optimizing the back-tracking algorithm and implementation. Many AJAX applications use hash fragments in URLs nowadays. Investigating how such hash fragments can be utilized during crawling is another interesting direction. Exploring the hidden Web induced by client-site JAVASCRIPT using CRAWLJAX and continuing with automated Web analysis and testing are other application domains we will be working on.

REFERENCES

ALVAREZ, M., PAN, A., RAPOSO, J., AND HIDALGO, J. 2006. Crawling webpages with support for client-side dynamism. In *Proceedings of the International Conference on Advances in Web-Age Information Management*. Lecture Notes in Computer Science Series, vol. 4016, 252–262.

ALVAREZ, M., PAN, A., RAPOSO, J., AND VINA, A. 2004. Client-side deep Web data extraction. In *Proceedings of the IEEE International Conference on E-Commerce Technology for Dynamic E-Business (CEC-EAST'04)*. IEEE Computer Society, Los Alamitos, 158–161.

ATTERER, R. AND SCHMIDT, A. 2005. Adding usability to Web engineering models and tools. In *Proceedings of the 5th International Conferencee on Web Engineering (ICWE'05)*. 36–41.

BARBOSA, L. AND FREIRE, J. 2007. An adaptive crawler for locating hidden-Web entry points. In *Proceedings of the 16th International World Wide Web Conference (WWW'07)*. ACM Press, New York, NY, 441–450.

BEZEMER, C.-P., MESBAH, A., AND VAN DEURSEN, A. 2009. Automated security testing of Web widget interactions. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'09)*. ACM Press, New York, NY, 81–91.

BOLDI, P., CODENOTTI, B., SANTINI, M., AND VIGNA, S. 2004. Ubicrawler: A scalable fully distributed Web crawler. *Softw. Practice Exper. 34*, 8, 711–726.

BRIN, S. AND PAGE, L. 1998. The anatomy of a large-scale hypertextual Web search engine. *Comput. Netw. ISDN Syst. 30*, 1–7, 107–117.

BURNER, M. 1997. Crawling towards eternity: Building an archive of the world wide web. *Web Techniques Mag. 2*, 5, 37–40.

CHAWATHE, S. S., RAJARAMAN, A., GARCIA-MOLINA, H., AND WIDOM, J. 1996. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*. ACM Press, New York, NY, 493–504.

CHO, J. AND GARCIA-MOLINA, H. 2002. Parallel crawlers. In *Proceedings of the 11th International World Wide Web Conference*. ACM Press, New York, NY, 124–135.

CHO, J., GARCIA-MOLINA, H., AND PAGE, L. 1998. Efficient crawling through URL ordering. *Comput. Netw. ISDN Syst. 30*, 1–7, 161–172.

DASGUPTA, A., GHOSH, A., KUMAR, R., OLSTON, C., PANDEY, S., AND TOMKINS, A. 2007. The discoverability of the Web. In *Proceedings of the 16th International World Wide Web Conference (WWW'07)*. ACM Press, New York, NY, 421–430.

DE CARVALHO, A. F. AND SILVA, F. S. 2004. Smartcrawl: A new strategy for the exploration of the hidden Web. In *Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management (WIDM'04)*. ACM Press, New York, NY, 9–15.

DIJKSTRA, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 1, 269–271.

DUDA, C., FREY, G., KOSSMANN, D., MATTER, R., AND ZHOU, C. 2009. Ajax crawl: Making Ajax applications searchable. In *Proceedings of the 25th International Conference on Data Engineering (ICDE'09)*. IEEE Computer Society, Los Alamitos, CA, 78–89.

FIELDING, R. AND TAYLOR, R. N. 2002. Principled design of the modern Web architecture. *ACM Trans. Inter. Tech. 2*, 2, 115–150.

GARAVEL, H., MATEESCU, R., AND SMARANDACHE, I. 2001. Parallel state space construction for model-checking. *Model Check. Softw. 2057*, 217–234.

GARRETT, J. 2005. Ajax: A new approach to Web applications. Adaptive path. http://www.adaptivepath.com/publications/essays/archives/000385.php.

HEYDON, A. AND NAJORK, M. 1999. Mercator: A scalable, extensible Web crawler. *Proceedings of the International World Wide Web Conference, 2*, 4, 219–229.

LAGE, J. P., DA SILVA, A. S., GOLGHER, P. B., AND LAENDER, A. H. F. 2004. Automatic generation of agents for collecting hidden Webpages for data extraction. *Data Knowl. Eng. 49*, 2, 177–196.

LEVENSHTEIN, V. L. 1996. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics Control Theory 10*, 707–710.

MADHAVAN, J., KO, D., KOT, L., GANAPATHY, V., RASMUSSEN, A., AND HALEVY, A. 2008. Google's deep Web crawl. *Proc. VLDB Endow. 1*, 2, 1241–1252.

MAXWELL, S. AND DELANEY, H. 2004. *Designing Experiments and Analyzing Data: A Model Comparison Perspective*. Lawrence Erlbaum, U.K.

MEMON, A., BANERJEE, I., AND NAGARAJAN, A. 2003. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*. IEEE Computer Society, Los Alamitos, CA, 260–269.

MEMON, A., SOFFA, M. L., AND POLLACK, M. E. 2001. Coverage criteria for GUI testing. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'01)*. ACM Press, New York, NY, 256–267.

MESBAH, A., BOZDAG, E., AND VAN DEURSEN, A. 2008. Crawling Ajax by inferring user interface state changes. In *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*. IEEE Computer Society, Los Alamitos, CA, 122–134.

MESBAH, A. AND VAN DEURSEN, A. 2007. Migrating multipage Web applications to single-page Ajax interfaces. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE Computer Society, Los Alamitos, CA, 181–190.

MESBAH, A. AND PRASAD, M. R. 2011. Automated cross-browser compatibility testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering (ICSE'11)*. ACM Press, New York, NY, 561–570.

MESBAH, A. AND VAN DEURSEN, A. 2008. A component- and push-based architectural style for Ajax applications. *J. Syst. Softw. 81*, 12, 2194–2209.

MESBAH, A. AND VAN DEURSEN, A. 2009. Invariant-based automatic testing of Ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, Los Alamitos, CA, 210–220.

NTOULAS, A., ZERFOS, P., AND CHO, J. 2005. Downloading textual hidden Web content through keyword queries. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL'05)*. ACM Press, New York, NY, 100–109.

PINKERTON, B. 1994. Finding what people want: Experiences with the Web crawler. In *Proceedings of the 2nd International World Wide Web Conference*. Vol. 94. 17–20.

RAGHAVAN, S. AND GARCIA-MOLINA, H. 2001. Crawling the hidden Web. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*. 129–138.

ROEST, D., MESBAH, A., AND VAN DEURSEN, A. 2010. Regression testing Ajax applications: Coping with dynamism. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*. IEEE Computer Society, Los Alamitos, CA, 128–136.

RUSSELL, A. 2006. Comet: Low latency data for the browser. http://alex.dojotoolkit.org/?p=545.

VALMARI, A. 1998. The state explosion problem. In *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*. 429–528.

W3C. W3C Web Storage. http://dev.w3.org/html5/webstorage/.

Received December 2010; revised June 2011; accepted August 2011