Crawling Rich Internet Applications: The State of the Art

Suryakant Choudhary[1], Mustafa Emre Dincturk[1], Seyed M. Mirtaheri[1], Ali Moosavi[1], Gregor von Bochmann[1,2], Guy-Vincent Jourdan[1,2], Iosif Viorel Onut[3,4]

[1]EECS, University of Ottawa, Ottawa, Ontario, Canada
[2]Fellow of IBM Canada CAS Research, Markham, Ontario, Canada
[3]R&D, IBM® Security AppScan® Enterprise, Ottawa, Ontario, Canada
[4]IBM Canada Software Lab, Canada
{schou062, mdinc075, smirt016, smoos078}@uottawa.ca
{bochmann, gvj}@eecs.uottawa.ca
vioonut@ca.ibm.com

Abstract

Web applications have come a long way, both in terms of adoption to provide information and services, and in terms of the technologies to develop them. With the emergence of richer and more advanced technologies such as AJAX, web applications have become more interactive, responsive and user friendly. These applications, often called Rich Internet Applications (RIAs), changed the web applications in two ways: (1) dynamic manipulation of client-side state and (2) asynchronous communication with the server. However, at the same time, such techniques also introduced new challenges. One important challenge is the difficulty of automatically crawling these new applications. Without crawling, RIAs cannot be indexed nor tested automatically. Traditional crawlers are not able to handle these newer technologies. This paper surveys the research on addressing the problem of crawling RIAs and provides some experimental results to compare existing crawling strategies. In addition, we provide some future directions for research on crawling RIAs.

1 Introduction

Over the last decade, web applications have evolved from simple applications which presented limited user experience into more responsive and interactive applications, so-called Rich Internet Applications (RIAs).

A traditional web application consists of HTML pages that are exclusively generated on the server-side. Each HTML page is addressed by a URL and may contain URLs (hyperlinks) to other pages. In traditional applications, the client-side (web browser) is only used for viewing a page received from the server and for requesting new pages by following the hyperlinks on the current page. There is no processing related to the application logic on the client-side. Also, in traditional web applications the communication is synchronous, meaning that when a request is sent, the user interaction is blocked until the response is received. Each response is a complete HTML page that replaces

the current one shown to the user.

RIAs improve traditional web applications in two ways: First, RIAs add more processing capability on the client-side through client-side scripts, such as JavaScript. That is, the server now sends, along with the HTML page, the scripts that could be executed when an event occurs. An event is a user interaction (such as a mouse click) or a time-out. The scripts running on the client-side are able to access and manipulate the page using a special interface defined in a platform and language-independent convention, called Document Object Model (DOM) [1]. A DOM instance (or simply DOM) is a tree data structure that is built at the client-side following the hierarchical structure of the corresponding HTML page. It basically contains all the information required to display the page as well as the events. Client-scripts can generate new pages, by modifying the DOM of the currently displayed page, without contacting the server. Second, the RIA client-scripts can initiate asynchronous communication with the server, using technologies like AJAX (Asynchronous JavaScript and XML) [2]. That is, client-scripts can generate requests to the server without blocking the user interaction and the responses are processed whenever they arrive.

With these improvements, it is now possible to modify the displayed page partially (or completely) to transform it into a new page on the client-side, possibly using the data retrieved asynchronously from the server. Thus, RIAs provide continuous user experience by having two programs running concurrently, one on the server-side and one on the client-side.

However, this evolution of web applications has a serious impact on the ability to crawl RIAs since RIAs violate the assumptions on which traditional crawling algorithms are based. These are the assumptions that each page is addressed by some URL and the synchronous communication.

Crawling is the process of exploring a web application to discover the pages of the application. There are two important motivations for crawling. First, crawling is required to index the content of the application to make it searchable. Second, crawling is required for the automated testing of web applications for various purposes, such as detecting security vulnerabilities or accessibility issues. The result of crawling is called "a model" of the application that is crawled. A model consists of

- *states* that represent the distinct pages and
- *transitions* that represent the possible ways to move from one state to another.

Crawling a traditional application is relatively easy since each state can be identified by its URL. To discover the existing pages, it is enough to find the different URLs in the application and download the corresponding pages. That is, the crawler starts from a given URL, downloads the corresponding page and extracts all the URLs embedded in the page. Then it processes each newly discovered URL the same way until all the pages are visited. Once a page is discovered, it can be analyzed according to the purpose of the crawl.

Using the traditional algorithm alone is not enough for crawling RIAs since the client-script execution can change the state of the application without changing the URL. As a result, we cannot in general rely on URLs to identify states, except for a few that can be reached by a URL. For this reason, the crawler has to identify the states based on the DOMs. In order for a crawler to discover new states, it should simulate each of the possible user-interactions on a page to see if a state change occurs, that is, if an event results in a "different" DOM. This is achieved by identifying the possible events in the DOM of the given page and then executing each event from that particular DOM (executing an event means to execute the corresponding script). This type of exploration is referred to as "event-based crawling". Event-based crawling should be applied separately for each distinct URL whose corresponding page has events. Event-based crawling for a given URL terminates when all the states that can be reached from the initial state of the URL are discovered. This requires the crawler to explore all the possible events from all the discovered states, as it is not possible to know, a priori, if executing an event from a state leads to a new state or not.

It is an important challenge in event-based crawling to recognize whether the DOM

reached after an event exploration corresponds to a new state or whether the state was discovered before. This is necessary to build a meaningful model and avoid redundant exploration. For this purpose, the crawler needs a DOM equivalence relation. A DOM equivalence relation partitions the DOMs such that each subset of DOMs (equivalence class) represents a state. One may consider using the equality as the DOM equivalence relation, which considers two DOMs as equivalents only if they are identical. However, web pages often contain parts that change frequently, such as advertisements, timestamps or counters. Such information is usually considered non-relevant for crawling, so they should be ignored when identifying states [3]. Simple equality would fail to ignore such non-relevant parts and may lead to unnecessarily large models and redundancy in the exploration. We note that the DOM equivalence relation to be used will depend on the purpose of crawling. For instance, for content indexing the text content is important to consider in the DOM equivalence, but for security analysis the text content is often non-relevant since two pages with the same structure but with different text will most probably have the same security vulnerabilities. Thus, it is important to choose an appropriate DOM equivalence relation for crawling.

A crawling strategy is an algorithm that decides what should be explored next during the crawl. In event-based crawling, the crawling strategy is important for crawling efficiency. In [4], we defined an efficient crawling strategy as a strategy that is able to discover as many states as possible, as soon as possible. This is important, as for some large applications, the crawl may take a long time to finish and we would like to have as many pages as possible to be available for indexing or analysis, early on in the crawl.

It is important to note that (under the right assumptions) loading a URL in the RIA leads to the initial state of that URL. The action of loading the URL to go back to the initial state of the URL is called a "reset". Resets can be considered a special transition in the model, which means that there is such a transition originating from each state, leading to the initial state (of the URL). A reset transition does not need to be traversed by the crawler (as it will not discover a new state); nevertheless, if needed it can always be used as a means to reach the initial state. The states discovered through event exploration cannot be reached directly in this manner. To move from the current state to another known state, the crawler has to execute an already explored sequence of events (possibly after a reset). For crawling efficiency, the crawling strategy should try to minimize the number of resets and event executions used for such relocation purposes.

There are many other challenges in event-based crawling. A common assumption is that executing the same event from the same client-state always leads to the same state. This assumption can be violated if the state of the application has a server-side state. In this case, the state of the application cannot be defined solely on the client-side, and the result of the event executions depends on the server-side state. Also, the state reached might depend on the values of the user-inputs. Since in theory, there could be an infinite number of user-input values (consider, for example, the text that the user can enter in a text field), it is usually assumed that a finite set of representative input values (used during the crawl) would be enough to discover all the states. In addition, the unique identification of events across an application is yet another challenge. The same event could be found in different DOMs and the crawling strategy may benefit from knowing if an event that exists in one state also exists in another in order to make its decisions. The last challenge that we mention here is the intermediate states caused by asynchronous server requests [5]. In RIAs, the client-scripts that use asynchronous requests during the processing of an event may lead to intermediate states that are neither the state before the user request is made, nor the state after the response is processed completely. The application is in such an intermediate state when the request is sent to the server but the response has not yet processed. Note that during this time interval, the application still allows user-interaction. Interleaving of several concurrent asynchronous requests increases the number of such states. Because of the cost and complexity of capturing them, the intermediate states are usually

ignored.

The problem of crawling RIAs is a problem that needs to be addressed in order to keep web applications searchable and testable. In this paper, we provide a survey of the work related to RIA crawling and point to some future directions.

The paper is organized as follows. We start with the survey of the research work in the field of crawling RIAs in Section 2 and 3. In Section 4, we describe the experimental results of the available RIA crawling strategies. Finally, we conclude with some future directions in the field of crawling RIAs.

2 General RIA Crawling

The research area of web application crawling has made significant progress over the last 15 years. This facilitated the development of crawlers used by search engines and the automated web application scanners that analyze web applications for security vulnerabilities and accessibility issues. However, the majority of the research until recently has focused on crawling traditional web applications.

Traditional web application crawling is a well-researched field with multiple efficient solutions [6]. However, today none of the search engines and web application scanners are sufficient for RIAs [7]. In the case of RIAs, the current research is still trying to address the fundamental problem of crawling, i.e. automatically discovering the web pages of the application. This is not surprising given the short history of RIAs, over less than a decade.

2.1 Centralized Crawling

In the last few years, several papers have been published to solve the problem of RIA crawling mostly focusing on AJAX based applications. For example, [8, 9] focus on crawling strategies for RIAs; [10, 11] focus on crawling for the purpose of indexing and search. In [12], the aim is to make RIAs accessible to search engines that are not AJAX-friendly. In [13] the focus is on regression testing of AJAX applications, whereas [14] is concerned with security testing of web widget interactions, [15] focuses on invariant-based testing. However, except for the work done in [8, 9] most of the research is concerned with their ability to crawl RIAs and not much attention has been given to the actual efficiency of crawling. Crawling RIAs in its naive form seems to favor the standard Breadth-First and Depth-First strategies, which have been used in most of the published research with some modifications.

One of the earliest attempts for an AJAX crawling algorithm and optimization is presented in [11]. The authors proposed an AJAX crawler that crawls the application based on user events and builds a model of the application. The application is modeled using transition graphs which contain all the application entities (states, events and transitions). The crawler uses the Breadth-First search strategy to trigger all the events present in the page. If the DOM of the page changes then a new state and corresponding transition is added to the transition graph. After a new state is reached, the crawler uses a reset to go back to the initial state and invoke the next event in the initial state. Once all the events in the initial state have been explored, the crawler explores in a similar fashion the discovered states in the order they are discovered.

In addition to the crawling strategy, the authors also proposed few optimizations to improve the efficiency of the crawling process. They suggested caching of JavaScript function execution results to save expensive server calls. If the same JavaScript function is invoked again along with the same parameters, then the cached results are used instead of executing the function again.

In [10], the authors introduced an AJAX-aware search engine for indexing the contents of RIAs. Similar to traditional search engines, it contains a crawler, indexer and query processor, but the components are adapted to handle RIAs. The AJAX crawler has the role of identifying events in the application states. The crawler starts with identifying and executing events on the first page. The crawler uses a standard Breadth-First search. The crawler identifies a new state if an event execution generated a new DOM tree and the content of the DOM is different from already discovered states.

The result of the crawling process is maintained in a special application model which is annotated with new information as the crawling proceeds. The authors recommended the exploration of a limited number of different events and different states or having a maximum limit on the depth of the crawl. The other components of the search engine, such as the indexer, read the information from the application model discovered by the crawler.

In [16], the authors introduced AjaxRank which is an adaptation of PageRank [17] to states in RIAs. Similar to the PageRank, the AjaxRank is connectivity-based, but instead of links the transitions are considered. In the AjaxRank, the initial state of the URL is given more importance (as it is the only state reachable from anywhere directly), hence states that are closer to the initial state also get higher ranks.

In [12, 18], the authors proposed an approach to analyze and reconstruct the states of RIAs automatically. They also introduced the tool "Crawljax" for the purpose of crawling RIAs. The crawler is capable of exercising client-side code and can identify events that change the state of the application. The information discovered by the tool is maintained as a state-flow graph capturing the client-side states and the possible transitions between them.

To distinguish between two states, an edit distance is calculated between the DOM trees using the Levenshtein method [19]. They use a similarity threshold to determine whether a reached DOM is a new state or not.

The crawler uses a crawling strategy similar to the Depth-First strategy. To identify the events in the current state, the differences between the previous DOM tree and the current one is calculated. The resulting delta is used to find new events that are then further processed in a Depth-First manner. In other words, only the HTML elements that changed from the previous state to the current one are analyzed to identify new events to be explored from the current state. To keep track of which events are explored the tag name, the list of attribute names and values, and the XPath expression of the corresponding element is used.

Upon completion of the recursive Depth-First call, the application should be put back into the state it was before the call. This can be done by using a reset and then click through from the initial state. To optimize this operation, the authors state it is possible to use Dijkstra's shortest path algorithm to find the shortest event sequence from the initial state to the desired state [18].

The authors also developed a Domain Specific Language called Crawling AJAX Specification Language (CASL) to define the elements to be clicked during crawling along with the exact order in which the crawler should crawl the application.

After the crawling process is finished the created state-flow graph can be used to generate a multi-page static version of the original AJAX application to enable search engines to crawl and index contents of the RIA.

In [20, 21], the authors focus on modeling and testing RIAs using execution traces. Their initial work [20] is based on first obtaining execution traces from user-sessions (a manual method). Once the traces are obtained, they are analyzed and an FSM model is formed by grouping together the equivalent user interfaces according to an equivalence relation. The proposed technique was also implemented by a software tool called "RE-RIA" to perform these activities. In a later paper [21], they introduced a tool, called "CrawlRIA", which automatically generates execution traces using a Depth-First strategy. That is, starting from the initial state, events are executed in a depth-first manner until a DOM that is equivalent to a previously visited DOM is reached. Then the sequence of states and events is stored as a trace in a database, and after a reset, crawling continues from the initial state to record another trace. These automatically generated traces are later used to form an FSM model using the same technique that is used in [20] for user-generated traces.

In [22], the authors introduced "CreRIA" an integrated reverse engineering environment for dynamic analysis of RIA for supporting the comprehension of processed RIAs. They use an agile, iterative process for this purpose. However, the model must be validated by a human expert on the basis of his knowledge about the application.

In [23], the authors presented a tool called

"DynaRIA" that has been designed to support the comprehension of RIAs in different contexts, such as maintenance, reverse engineering and testing processes. The tool is based on dynamic analysis of the application and provides functionalities for capturing and tracing the user sessions, analyzing the data captured and producing several types of abstractions and visualizations about the run-time behavior of the application such as UML sequence diagrams at various levels of detail. Such information might be useful for obtaining an overview of information exchanged between the server and the client, the timing behavior, the events executed etc.

In [24], the authors proposed the extraction of an FSM representation of an AJAX application through dynamic analysis, complemented by information coming from static code analysis. One important point here is that dynamic analysis is by definition partial, and hence they recommended a manual validation step after the model extraction to ensure that the states and transitions of the extracted model are valid. Any missing states or transitions must be added manually. The amount of manual work required in this step depends on the number and quality of the available information after the dynamic analysis.

In [25], the authors suggested using a greedy crawling strategy. That is, the strategy is to explore an event from the current state if there is an unexplored event. Otherwise, the crawler moves to the closest state with an unexplored event. They also suggested two other variations of this strategy. In these variations, the most recently discovered state and the state closest to the initial state are chosen when there is no event to explore in the current state. They concluded all three variations have similar performance on their test applications in terms of the total number of event executions completing the crawl.

2.2 Distributed Crawling

2.2.1 Traditional Models

Due to the large size of the web, it is often the case that crawlers use several nodes (i.e. computers) to crawl the web simultaneously. Distributed crawling of the web has been extensively described in the literature [6]. [26] classifies distributed crawlers based on their work assignment method into three classes:

Independent Assignment: Different crawlers start from different URLs and crawl the web independently. This approach may lead to overlap and duplication of work.

Dynamic Assignment: This approach is based on one or more units that keep track of discovered and executed tasks. Upon discovering a task, the node will inform these units and if it is a new task, the unit will add it to the task queue. Nodes then ask the unit for workload, and the unit assign tasks to the probing nodes [27]. The first prototype of Google roughly followed this architecture: A centralized unit, called URLserver, stores the list of URLs and orders a set of slave nodes to download each URL. All downloaded pages are stored in a unit, called Storeserver. The retrieved pages are then indexed distributively. Both the downloading and indexing tasks requires centralized units of coordination [28].

Static Assignment: In this approach a set of homogeneous workers are allocated unique IDs. The mapping function maps each task to one of the assigned IDs. Upon encountering a task the crawler examines the task and decides whether the task falls under its jurisdiction or belongs to another node. In the first case, the node takes care of the task autonomously. Otherwise, the node will inform the node responsible for the task [26]. Different proposals suggest different matrices and algorithms to derive the mapping function. In [29] the distribution of the task of crawling of the different URLs is performed by hashing the URL (either only the host-name part, or the entire URL) and distributing the resulting hash values to the different crawlers, for instance, using the distributed hash table (DHT) of a peer-to-peer system. [30] also includes the geographic information about the crawlers and the searched servers into the task distribution algorithm in order to allocate a crawler that is geographically close to the server to be crawled. UbiCrawler [31] uses the so-called consistent hashing approach to allocate the tasks to the different crawlers in such a way that there are only minimal changes when some crawler disappears or new crawlers come in. This approach can be

used to obtain better fault tolerance.

2.2.2 Limitations

Partitioning the search space based on the URL may not lead to the best performance in a RIA. In a traditional web application, there is often a one to one relation between the state of the DOM and the URL of the page. This allows the crawlers to balance the work load among them by partitioning the search space based on the URLs using the Static Assignment method. This assumption is however broken in a RIA in which one seed URL may be associated with a large number of DOM states. Furthermore, the cost of reaching a DOM state in the traditional web applications is often constant: one simply has to download the target URL. This is not the case in a RIA. In order to get to a DOM state one first has to download a seed URL, then execute a set of events, some involving communication with the server, to reach the DOM state. Deviation of the RIAs from the traditional web applications with respect to these fundamental issues makes the current strategies less suitable for this purpose.

The only parallel method of crawling RIAs we know of is that of Mesbah et al. who use threads to concurrently crawl RIAs [18]. This approach is based on shared memory among the threads which is given in a multi-core processing unit. However, this approach cannot be deployed over multiple computers.

3 Model-Based Crawling

At the first glance, the use of Breadth-First or Depth-First strategies might look like a good solution to the problem of event-based crawling. This is more or less the approach taken in the relevant research works as discussed above. But none of this research has focused on the efficiency of the crawling strategy. Efficiency is an important factor when it comes to crawling RIAs, as most RIAs are complex web applications with a very large state space. The Breadth-First and Depth-First strategies in their standard form and under specified circumstances will eventually be able to crawl a given RIA. However, there are two important shortcomings of these strategies when used for RIAs. First, they do not predict which event executions are more likely to lead to new states. Therefore they do not have any mechanism to prioritize some events over others. Second, both strategies explore the states in a strict order. That is, in Depth-First crawl, the most recently discovered state is explored first, and in the Breadth-First crawl the state that is discovered earlier is explored first. Note that, exploring a state means to explore all the events in the state, hence when these strategies end up in a state that is different than the one currently being explored, they need to go back to that state in order to finish the remaining events. That increases the number of event executions and resets used for relocation purposes. In [8] the authors indicate opportunities to be able to design more efficient strategies by identifying general patterns in the actual RIAs being crawled and using these patterns to come up with reasonable assumptions about the model of the application. This approach has been called "Model-Based Crawling". The assumptions can be justified by the user by interacting informally with the RIA in order to gain a general understanding of how it is structured and how it interacts with the user. In [8] the concept of model-based crawling is defined as follows:

1. "First, a general hypothesis about the behavior of the application is conceptualized. The idea is to assume that the application will behave in a certain way. Based on this hypothesis, one can define the anticipated model of the application, which is called the "meta-model". This will transform the process of crawling from the discovery activity to determine "what the model is" to the activity of validating whether the assumed model is correct".

2. "Once a hypothesis is elaborated and an assumed model is defined, the next step is to define an efficient crawling strategy to verify the model. Without having an assumption about the behavior of the application, it is impossible to define any strategy that will be efficient".

3. "However, it is important to note that any real world application will never follow the assumed model to its entirety. Therefore, it is also important to define strategies which will reconcile the differences discovered between the

assumed model and the real model of the application in an efficient way".

Thus, model-based crawling defines the goal of a crawling strategy as being able to anticipate automatically an accurate model of the application.

In [8], a two-stage approach to confer to the primary goal of finding all states as soon as possible is also defined. The first phase is the "state exploration phase". It aims at discovering all the states of the RIA being crawled. Once the strategy predicts, based on its assumptions, that there are no new states to discover, it proceeds to the second phase, the "transition exploration phase" which tries to execute the remaining transitions that was not explored in the first phase, to confirm that nothing has been overlooked. The motivation for defining these two phase is that in many cases the application is too complex to be crawled completely, and it is important to explore, in the given time, as many states as possible, but unless all transitions have been explored, one cannot be sure that all states are found.

3.1 Hypercube

The first meta-model that was introduced in this context was the Hypercube meta-model [8]. The Hypercube meta-model is based on two assumptions:
(1) Given a set of events enabled at a state, executing these events in different orders does not change the state reached.
(2) When an event is executed; it becomes disabled but does not disable or enable other events.
The model of an application that follows these assumptions is a hypercube structure. An optimal strategy to crawl the applications that follow the Hypercube meta-model is designed. That is, if an application has a hypercube model, the Hypercube strategy uses a minimal number of resets and event executions to first visit every state and later to make sure that every transition is explored once. Of course, the Hypercube strategy adapts itself in case the application does not follow the hypercube assumptions.

3.2 Probability

In the probability strategy [9], some statistical information about the behavior of the events in the application is collected during the crawl. That is, events are prioritized based on their probability of discovering a new state. Initially, each event has the same default probability and when an event is explored its probability is updated based on the result of exploration. Then, the strategy is simply to take the action that will maximize the probability of finding a new state while trying to minimize the number of events and resets used for relocation purposes (that is, the cost of moving from the current state to another known state for exploring an event).

3.3 Menu

The menu meta-model is formed based on the following hypothesis about the application: The result of an event execution is independent of the state (source state) where the event has been executed and always results in the same resultant state. This can be conceptualized as a mapping between the event and the resulting state.

Such behavior for example is realized by the menu items present in a web application or other common applications such as "home", "help", "about us" etc. Executing these menu items will result in the same state.

The menu crawling strategy categorizes the events into multiple priorities based on the event's likelihood of discovering a new state and whether the event has followed the menu assumption in its previous execution instances. The menu crawling strategy then executes the events in order of their priority. The priorities of the events are updated as more information is discovered about their execution results from different states and when the assumptions about the event execution results are violated.

4 Experimental Results

In this section, we compare the performance of the Breadth-First, Depth-First, Greedy, Menu, Hypercube, Probability and Crawljax strategies. We use two real RIAs and two test RIAs.

The following metrics are used for performance evaluation.
(1) Number of events and resets required to discover all states
(2) Number of events and resets required to explore all transitions
A reset is loading the URL of the application to go to the initial state. Resets are typically costlier (in terms of time of execution) than event executions. For simplicity, we have combined the events and resets used by a strategy into a single cost factor. For this purpose, we have expressed the cost of reset in terms of number of event executions (the actual value used is application-dependent and measured by experiments). We believe that the number of events executions is a good metrics for performance evaluation since the time to crawl is proportional to the number of events executed during the crawl.

We also present the optimal number of event executions necessary to explore all the states of an application. Note that, this optimal value is calculated after the fact, once the model of the application is obtained. This number is found by an Asymmetric Traveling Salesman Problem (ATSP) solver [32] on the extracted models.

We are interested in two factors to define the efficiency of the crawl [9]:
1. State Discovery Cost: The cost required to discover all the states of the application. This cost is important as it might not be feasible to finish the crawl (factors such as timing constraints) and hence, we would want to explore as much states as possible within the given runtime of the algorithm. Thus, it is very important to find what percentage of the total state space has been discovered by the crawling algorithm at any given time during the crawl.
2. Total Crawling Cost: The total cost required to complete the crawl i.e. to discover the complete application model.

4.1 Comparison with Crawljax

It is important to note that the default strategy used by Crawljax does not explore every event from every state [1]. That is, an event is only executed from the state where the event is first encountered. When its default strategy is used, Crawljax might not discover all the states. For this reason, it is hard to compare the performance of our strategies (which aim at crawling the application completely) with the results obtained with the default strategy of Crawljax (which does not have such a goal). Nevertheless, we present the number of states discovered and the corresponding costs using the default strategy of Crawljax which will be referred to as Crawljax_Default in the remainder.

Crawljax can also be configured (by disabling the "clickOnce" option) such that it explores the application completely like the other strategies. In that case, the Crawljax's strategy is a Depth-First strategy and is able to discover all the states. However, this Depth-First strategy is an "unoptimized" one, meaning that each time Crawljax needs to relocate to another state to explore an event, it uses a reset and follows an event sequence that will lead to the destination state. Our implementation of the Depth-First strategy is "optimized" such that the shortest path from the current state to the destination state is used for relocations (the same is true for our Breadth-First implementation). The results we present as Depth-First strategy is obtained using our implementation. The "unoptimized" version of the Depth-First strategy gives worse results than what we present here.

4.2 Test Applications

The first real RIA we consider is an AJAX-based periodic table[2]. In total 240 states and 29034 transitions are identified by our crawler and the reset cost is 8.

The second real application considered is Clipmarks[3]. For this experimental study we have used a partial local copy of the website. It consists of 129 states and 10580 transitions

[1] In this study, Crawljax version 2.0. is used and except for specifying which elements to click for each application, no limits are imposed in the crawl configurations regarding the parameters like the crawl depth, the maximum number of states to explore, the maximum runtime etc. The details of the configurations we have used for the experiments can be found at `http://ssrg.eecs.uottawa.ca/docs/crawljax/config.pdf`

[2] http://code.jalenack.com/periodic/
(Local version: http://ssrg.eecs.uottawa.ca/periodic/)

[3] http://www.clipmarks.com/
(Local version: http://ssrg.eecs.uottawa.ca/clipmarks/)

and the reset cost is 18.

The third application, TestRIA[4] is a test application that we developed using AJAX. It has 39 states and 305 transitions and a reset cost of 2.

The fourth application is a demo web application maintained by the IBM® AppScan® Team [5]. We have used the AJAX-fied version of the website. The application has 45 states and 1210 transitions and a reset cost of 2.

4.3 State Discovery Results

We first present the cost required by each strategy to discover all the states of an application. It is important to mention that the cost of finding all the states does not necessarily correspond to a complete crawl since the crawler does not know that all the states have been discovered until it has executed all the events. The cost of complete crawl is described in Section 4.4.

For compactness of the presentation of our results we use box plots: the top of a vertical line shows the cost required to discover all the states. The lower edge of the box, the line in the box and the higher edge of the box indicate the number required to discover a quarter, half and three quarters of all the states of the application, respectively. The position of the box and the horizontal lines in the plot is used to assess whether a method is able to discover new states faster than others. The presented box plots does not contain results for Crawljax_Default for the reasons explained in Section 4.1.

In addition, the Table 1 shows for each application and for each strategy, the number of states discovered, the total number of event executions and resets required by the strategy to discover the states.

The results show that model-based crawling strategies discover all the states of the application more efficiently than the Greedy [25], the Depth-First and the Breath-First strategies. Except for AltoroMutual, Crawljax_Default did not discover all the states hence a comparison with Crawljax_Default is not possible in general.

[4]http://ssrg.eecs.uottawa.ca/TestRIA/

[5] http://www.altoromutual.com/

4.4 Costs of Complete Crawls

We also present in Table 2 the costs required for the complete crawl of each application i.e. the cost required to discover the complete application model. The table does not include data for Crawljax for the reasons explained in Section 4.1.

We can again see that the model-based crawling approach has better performance than the other strategies, especially the Breadth-First and the Depth-First strategies.

4.5 Results Evaluation

As an overall evaluation, model-based crawling seems to be a promising approach for designing efficient crawling strategies for RIAs. The model-based crawling strategies aim at finding most of the states of the application as soon as possible and eventually find all the states and transitions of the web application. Experimental results show that the model-based crawling strategies perform very well and outperform the Depth-First and Breadth-First crawling strategies by significant margins. Further, they also outperform the Greedy strategy in most cases while being comparable in the least favorable example.

5 Future of RIA Crawling

As web applications are evolving and expanding rapidly, merely relying on model-based crawling is not always sufficient to guarantee a good coverage and freshness with limited computational resources. The complexity of the emerging RIAs requires one to deploy a variety of techniques to crawl RIAs efficiently.

Such techniques should help the crawling engine to learn about the characteristics of the application on-the-fly so that the engine can optimize its strategies, avoid taking unnecessary transitions and achieve a good coverage and freshness with the available resources. Furthermore an unavoidable step in the future of RIA crawling is to harvest the power of parallel processing and cloud computing to reduce the long time it takes to crawl an application. This section elaborates on some of these emerging techniques.

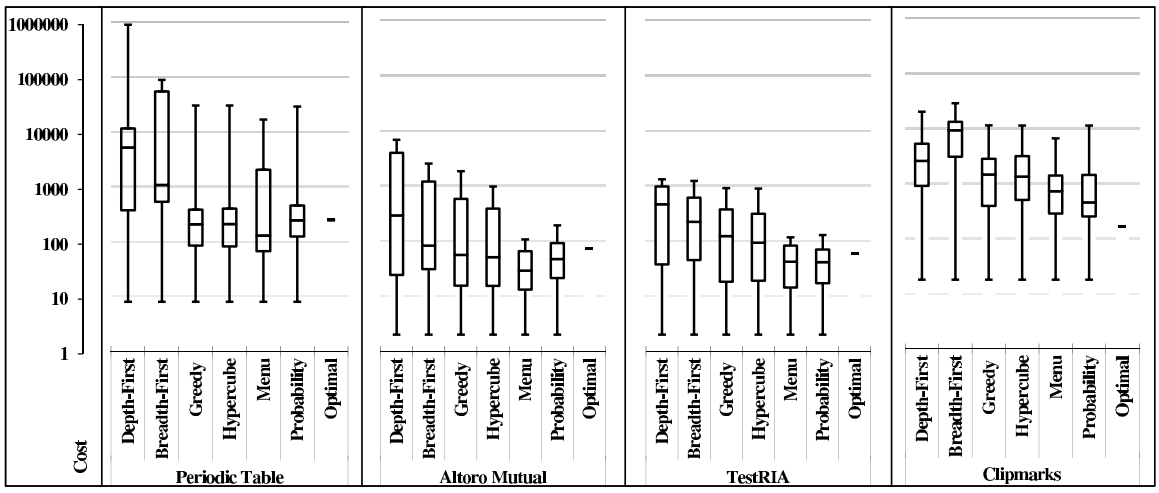

Figure 1: Costs of State Discovery (in log scale)

	Periodic Table				Clipmarks				TestRIA				Altoro Mutual			
	States	Events	Resets	Cost	States	Events	Resets	Cost	States	Events	Resets	Cost	States	Events	Resets	Cost
Crawljax_Default	121	146	121	1,114	22	79	29	601	21	52	16	84	45	109	54	217
Depth-First	240	897,080	78	897,704	129	19,300	45	20,106	39	1,312	1	1,314	45	6,822	15	6,853
Breadth-First	240	28,888	7,504	88,922	129	12,619	894	28,711	39	1,118	54	1,226	45	1,868	333	2,534
Greedy	240	29,653	83	30,316	129	11,029	21	11,414	39	910	1	912	45	1,828	12	1,851
Hypercube	240	29,643	78	30,267	129	10,985	16	11,271	39	894	1	896	45	919	28	976
Menu	240	16,543	26	16,747	129	6,008	33	6,596	39	115	1	117	45	100	3	107
Probability	240	28,935	2	28,951	129	10,820	25	11,266	39	126	1	128	45	180	7	193
Optimal	240	239	1	247	129	128	2	164	39	57	1	59	45	72	1	74

Table 1: State Discovery Results

5.1 New States without New Information

The states of the web application are usually defined based on their DOM structures and contents. Whenever the crawler encounters a DOM that is not seen before, it is regarded as a new state that needs to be investigated. However, a drawback of this definition of state is that a new DOM might not necessarily contain new information. A promising direction to improve RIA crawling is to detect and avoid such states. A good example of a web application in which such situation occurs frequently is a web application that consists of several independent widgets. Since each widget shows different contents at different times independently of others, they can easily combine with each other in different ways and make up new DOMs which actually contain no new data. Failure to detect such states as known states may lead to an unnecessary state space explosion and can cause the crawling time to increase exponentially. A new definition of state can help identifying them. A future direction of RIA crawling is to make a new definition of web application states which is not based on DOM equivalence, but rather based on interesting information that they contain.

One possible approach in this direction is to make the crawler smart enough to guess which portions of a web application interact with the user independently of other portions and assign states to each portion separately, called sub-states. This way, a new DOM which is basically a new combination of already seen sub-states would not be considered as a new state of the web application.

5.2 Adaptive Crawling

One possible significant improvement to model-based crawling would be to adapt the meta-model on-the-fly, based on the model built so far. Currently, the idea is to commit to a meta-model before the crawling starts, a commitment that must be made based on some external factors that are difficult to specify. In-

	Periodic Table			Clipmarks			TestRIA			Altoro Mutual		
	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost
Depth-First	897,358	236	899,246	19,569	72	20,868	1,433	1	1,435	6,876	34	6,944
Breadth-First	64,850	14,633	181,916	15,342	926	32,015	1,216	55	1,326	3,074	334	3,742
Greedy	29,926	236	31,814	11,396	56	12,398	1,001	1	1,003	2,508	34	2,576
Hypercube	29,921	236	31,809	11,350	56	12,356	994	1	996	2,489	34	2,557
Menu	37,489	236	39,377	11,769	71	13,043	974	1	976	2,457	35	2,527
Probability	29,548	236	31,436	11,456	62	12,563	972	1	974	2,451	34	2,520

Table 2: Costs of Complete Crawls

stead, the crawl could be started initially following a "generic" meta-model, for example the probability meta-model, which is expected to provide good results in general. Then, after having crawled enough to accumulate some information about the model of the application, an analysis of the current graph could be done to evaluate whether some other meta-model would be more suitable. If one most suitable meta-model was found, a switch to that meta-model could be done, and the crawling would continue following that model. The situation could be regularly re-evaluated, and the meta-model switched more than once. It is even conceivable to mix meta-models, using different meta-models on different parts of the application.

We can push the concept even further and in effect create the meta-model itself on-the-fly based on the information gathered. For example, if some repeating pattern was detected in the site, the graph on the pattern could be analyzed to find an efficient way to crawl that particular graph. If the graph is small enough, an optimal crawl can even be computed. Then, as more instance of the pattern is uncovered, the optimal crawling strategy is followed. This latter approach would be particularly efficient for very large web sites with itemization of some information or widgets: these items or widgets typically repeat many times. In this case, once the pattern is detected, an optimal, strategy can easily be constructed for the pattern and reused from that point on, on every new instance of the pattern.

5.3 Greater Diversity

Exhaustive crawling on a large web application might take excessively long time to finish. It is not always feasible to wait for the crawl to finish completely before getting any results. Often it is more desirable to terminate the crawl after a limited amount of time. In such cases, there is the question which states of the application are crawled and which are postponed to be crawled later. When viewing results of a partial crawl, there is an expectation of a bird-eye view of the application instead of having detailed information on a small portion of the application and nothing from other parts. Therefore, the crawler needs to have a "diversified" exploration strategy, much like a human would do, instead of getting stuck in a particular subset for a very long time. We call this goal diversifying the crawl. In order to attain this goal one may prioritize the states. Diversifying the crawl is especially useful for applications with a large number of repeating structures such as the ones with long lists of items. Prioritizing states in such a way that the crawler moves away from these lists once it examines a few items in them rather than exhausting the whole list, may be a potential solution to this problem.

5.4 Distributed Crawling

One way to tackle the complexity of crawling large RIAs is to use multiple nodes. As explained in 2.2, distributed crawling of the web is a norm in traditional web crawling. To the best of our knowledge however distributed crawling of RIAs remains mostly an unexplored area in the literature of web crawling. One can use the thread-based model proposed in [18] and scale it up to achieve distributed crawling of RIAs. To scale this model up, one may replace each thread with a full-fledged operating system process running on different nodes, and also replace the shared memory among the threads with a central coordination unit. Processes can then contact this coordinator and get work assigned to them. This model is simi-

lar to the traditional dynamic work assignment in distributed crawling.

Similar to the crawling of traditional web applications one can also deploy static work assignment in distributed crawling of RIAs. In this model however merely partitioning the search space based on the URLs discovered may result in a few nodes becoming a bottleneck, due to the fact that the total number of states associated with a URL may vary a lot from one URL to another. Stronger load balancing algorithms and more sophisticated portioning strategies have to be devised to alleviate this asymmetry.

Running the crawl over an elastic cloud is another useful strategy to reduce costs and achieve higher efficiency. In an elastic cloud environment extra nodes may become available to the system or some of the current nodes may leave the system. Taking advantage of such an environment is trivial in the dynamic work assignment model: when a new node joins the system it simply requests work from the coordinator, and if a node intends to leave it sends its remaining work assignment back to the coordinator. In the static model, one can introduce load balancing or repartitioning the task space among the new set of workers in order to achieve an adjusted work force.

As with any crawler, a distributed RIA crawler may inadvertently be seen as launching a Distributed Denial of Service (DDoS) Attack. To avoid such situation a mechanism that limits the total number of requests should be integrated into the model. In the dynamic work assignment model, the coordination unit can be used to monitor and restrict the number of requests sent to a server. This unit may return a time frame with each task to the requesting node. It is then the responsibility of the node to perform the task within the allocated time frame. In the static model, one can achieve "politeness" by having a quota allocated to each node. It is then the responsibility of the node not to send more than the specified quota of requests to the server over any given time period. Similar to the tasks, quotas may be transferred to other nodes to avoid that a node becomes a bottleneck if it exhausts its quota.

Acknowledgements

This work is supported in part by IBM and the Natural Science and Engineering Research Council of Canada.

About the Authors

Suryakant Choudhary and Ali Moosavi are Master students at the EECS at the University of Ottawa working on the topic of designing efficient crawling strategies for RIAs.

Mustafa Emre Dincturk and Seyed M. Mirtaheri are PhD candidates at the EECS at the University of Ottawa working on model-based crawling of RIAs and distributed crawling of RIAs, respectively.

Gregor v. Bochmann is a professor at the EECS at the University of Ottawa since 1998, after 25 years at the University of Montreal. He is a fellow of the IEEE and ACM and a member of the Royal Society of Canada. He is known for his work on communication protocols and software engineering. Ongoing projects include the systematic development of distributed applications from global requirements, reverse engineering of RIAs, and control protocols for optical networks.

Guy-Vincent Jourdan is a professor at the EECS at the University of Ottawa. He joined the EECS in 2004, after 7 years in the private sector. He received his PhD from l'université de Rennes/INRIA in France in 1995 in the area of distributed systems analysis. His research interests include distributed systems modeling and analysis, software engineering, software security and ordered sets.

Iosif Viorel Onut is currently affiliated with IBM Security Systems and Center for Advanced Studies at IBM. He also is Adjunct Professor at the University of Ottawa. He completed his PhD at the University of New Brunswick and specializes in topics related to network security. Currently, his main research focus is in the area of Web 2.0 application security, compliance and crawling, but also intelligent sensor technologies for context-aware security risk assessment.

References

[1] "Document Object Model (DOM)." http://www.w3.org/DOM/, 2005.

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

[3] S. Choudhary, M. E. Dincturk, G. V. Bochmann, G.-V. Jourdan, I. V. Onut, and P. Ionescu, "Solving some modeling challenges when testing rich internet applications for security," *Software Testing, Verification, and Validation, 2012 International Conference on*, pp. 850–857, 2012.

[4] K. Benjamin, G. v. Bochmann, G.-V. Jourdan, and I.-V. Onut, "Some modeling challenges when testing rich internet applications for security," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pp. 403–409, IEEE Computer Society, 2010.

[5] K. Benjamin, "A strategy for efficient crawling of rich internet applications," Master's thesis, EECS - University of Ottawa, 2010. http://ssrg.eecs.uottawa.ca/docs/Benjamin-Thesis.pdf.

[6] C. Olston and M. Najork, "Web crawling," *Found. Trends Inf. Retr.*, vol. 4, pp. 175–246, Mar. 2010.

[7] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pp. 332–345, IEEE Computer Society, 2010.

[8] K. Benjamin, G. v. Bochmann, M. E. Dincturk, G.-V. Jourdan, and I. V. Onut, "A strategy for efficient crawling of rich internet applications," in *Proceedings of the 11th international conference on Web engineering*, ICWE'11, 2011.

[9] M. E. Dincturk, S. Choudhary, G. v. Bochmann, , G.-V. Jourdan, and I. V. Onut, "A statistical approach for efficient crawling of rich internet applications," in *Proceedings of the 12th international conference on Web engineering*, ICWE'12, 2012.

[10] C. Duda, G. Frey, D. Kossmann, and C. Zhou, "Ajaxsearch: crawling, indexing and searching web 2.0 applications," *Proc. VLDB Endow.*, vol. 1, pp. 1440–1443, Aug. 2008.

[11] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou, "Ajax crawl: Making ajax applications searchable," in *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pp. 78–89, IEEE Computer Society, 2009.

[12] A. Mesbah, E. Bozdag, and A. v. Deursen, "Crawling ajax by inferring user interface state changes," in *Proceedings of the 2008 Eighth International Conference on Web Engineering*, ICWE '08, pp. 122–134, IEEE Computer Society, 2008.

[13] D. Roest, A. Mesbah, and A. van Deursen, "Regression testing ajax applications: Coping with dynamism.," in *ICST*, pp. 127–136, IEEE Computer Society, 2010.

[14] C.-P. Bezemer, A. Mesbah, and A. van Deursen, "Automated security testing of web widget interactions," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, 2009.

[15] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of ajax user interfaces," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pp. 210 –220, may 2009.

[16] G. Frey, "Indexing ajax web applications," Master's thesis, ETH Zurich, 2007.

[17] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," 1998. Standford University, Technical Report.

[18] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *TWEB*, vol. 6, no. 1, p. 3, 2012.

[19] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[20] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "Reverse engineering finite state machines from rich internet applications," in *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, WCRE '08, pp. 69–73, IEEE Computer Society, 2008.

[21] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "Rich internet application testing using execution trace data," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pp. 274–283, IEEE Computer Society, 2010.

[22] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "An iterative approach for the reverse engineering of rich internet application user interfaces," in *Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*, ICIW '10, pp. 401–410, IEEE Computer Society, 2010.

[23] D. Amalfitano, A. R. Fasolino, A. Polcaro, and P. Tramontana, "Dynaria: A tool for ajax web application comprehension.," in *ICPC*, pp. 46–47, IEEE Computer Society, 2010.

[24] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pp. 121–130, IEEE Computer Society, 2008.

[25] Z. Peng, N. He, C. Jiang, Z. Li, L. Xu, Y. Li, and Y. Ren, "Graph-based ajax crawl: Mining data from rich internet applications," in *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, vol. 3, pp. 590 –594, march 2012.

[26] J. Cho and H. Garcia-Molina, "Parallel crawlers," in *Proceedings of the 11th international conference on World Wide Web*, WWW '02, 2002.

[27] D. H. Chau, S. Pandit, S. Wang, and C. Faloutsos, "Parallel crawling for online social networks," in *Proceedings of the 16th international conference on World Wide Web*, WWW '07, 2007.

[28] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proceedings of the seventh international conference on World Wide Web 7*, WWW7, 1998.

[29] B. T. Loo, L. Owen, and C. S. Krishnamurthy, "Distributed web crawling over dhts," 2004.

[30] J. Exposto, J. Macedo, A. Pina, A. Alves, and J. Rufino, "Information networking. towards ubiquitous networking and services," ch. Efficient Partitioning Strategies for Distributed Web Crawling, pp. 544–553, Springer-Verlag, 2008.

[31] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: a scalable fully distributed web crawler," *Softw. Pract. Exper.*, vol. 34, pp. 711–726, July 2004.

[32] G. Carpaneto, M. Dell'Amico, and P. Toth, "Exact solution of large-scale, asymmetric traveling salesman problems," *ACM Trans. Math. Softw.*, vol. 21, pp. 394–409, Dec. 1995.