Journal of Web Engineering, Vol. 13, No. 3&4 (2014) 243–262
© Rinton Press

MODEL-BASED RICH INTERNET APPLICATIONS CRAWLING: "MENU" AND "PROBABILITY" MODELS

SURYAKANT CHOUDHARY, EMRE DINCTURK, SEYED MIRTAHERI
EECS, Faculty of Engineering, University of Ottawa, 800, King Edward Avenue
Ottawa, Ontario, K1N 6N5, Canada
{schou062, mdinc075, staheri}@uottawa.ca

GREGOR v. BOCHMANN[a], GUY-VINCENT JOURDAN[a]
EECS, Faculty of Engineering, University of Ottawa, 800, King Edward Avenue
Ottawa, Ontario, K1N 6N5, Canada
{bochmann, gvj}@eecs.uottawa.ca

IOSIF VIOREL ONUT
IBM Canada Software Lab Canada, Research and Development, IBM® Security AppScan® Enterprise
Ottawa, Ontario, Canada
vioonut@ca.ibm.com

Strategies for "crawling" Web sites efficiently have been described more than a decade ago. Since then, Web applications have come a long way both in terms of adoption to provide information and services and in terms of technologies to develop them. With the emergence of richer and more advanced technologies such as AJAX, "Rich Internet Applications" (RIAs) have become more interactive, more responsive and generally more user friendly. Unfortunately, we have also lost our ability to crawl them.

Building models of applications automatically is important not only for indexing content, but also to do automated testing, automated security assessments, automated accessibility assessment and in general to use software engineering tools. We must regain our ability to efficiently construct models for these RIAs. In this paper, we present two methods, based on "Model-Based Crawling" (MBC) first introduced in [1]: the "menu" model and the "probability" model. These two methods are shown to be more effective at extracting models than previously published methods, and are much simpler to implement than previous models for MBC. A distributed implementation of the probability model is also discussed. We compare these methods and others against a set of experimental and "real" RIAs, showing that in our experiments, these methods find the set of client states faster than other approaches, and often finish the crawl faster as well.

Keywords: Crawling, RIAs, AJAX, Modelling

1 Introduction

Building models automatically for software is an important aspect of software engineering. Many tools require such a model as input, for example automated software testing tools or reverse engineering tools. In the case of Web sites and Web applications, the most obvious motivation for automated model inference is indexing the content of the sites, which is done through "crawling". Indexing is obviously a central feature of the Web, but it is by no means the only reason why inferring models is important. We also require models for tasks related

[a] Fellow of IBM Canada CAS Research, Canada

to good software engineering: models are needed as input for automated testing of applications ("model-based testing"), models are also needed for automated security assessments, for automated usability assessments, or simply as a way to better understand the structure of the website.

Nearly two and a half decades of research in the area of model extraction and crawling has produced a large body of work with many powerful solutions [2,3]. The majority of the studies, however, focus on traditional web applications, where the HTML view of the page is generated on the server side. In this model, there is a one-to-one relation between the URL of the page and the state of its *Document Object Model* (DOM) [4]. Thus, many of the proposed web crawlers use the URL to identify the state of the DOM. This assumption reduces the basic task of crawling the Web to the task of finding all the valid and reachable URLs from a set of seed URLs.

However, the so-called *Rich Internet Applications* (RIAs) break the one-to-one relationship between the URL and the state of the DOM. In RIAs, DOMs are partially updated by client-side script execution (such as JavaScript®), and asynchronous calls to the server are done through technologies such as AJAX [5]. Such sophisticated client-side applications create a one-to-many relation between the URL and the reachable DOM states associated with that URL.

This evolution is positive, but comes at a cost which has been underestimated: the crawling techniques developed for traditional web applications just do not work on RIAs. We have lost our ability to crawl and model web applications as they are typically created today.[b] Even simple websites are not immune to the problem since common tools to create and maintain website content are increasingly adding AJAX-like scripts to the page. We need to address this issue, which means to develop web crawlers that do not rely solely on the URL to uniquely identify the state of the application, but also take into consideration the DOM structure and its properties to identify different states of the application. There is some work being done in that domain (see [6] for an overview), but more must be done. This paper is one step in this direction.

Crawling RIAs is much more complex than crawling traditional web applications. The one-to-many relation between a URL and states of the DOM can be modeled as a directed graph referred to as the *application graph*. In the application graph, each state of the DOM is a node, and each JavaScript event is a directed edge. To construct such a graph, one must differentiate between different states of the DOM, which is a challenge in itself, but outside the scope of this paper (see e.g. [7] for a discussion on the topic). In this model, taking an edge from a node means executing a JavaScript event from the DOM that the node represents.

After defining the application graph, the task of crawling a RIA is reduced to the task of discovering every state in the application graph. The state that is reached when a given URL is loaded is called the "initial state of the URL". For a crawler to ensure that all states reachable from a given URL are discovered, the crawler has to start from the initial state of the URL, take every possible transition, and do this for every newly discovered state recursively. This often takes a long time. It is thus interesting for a crawler to discover as many states as

[b] See e.g. `https://developers.google.com/webmasters/ajax-crawling/docs/getting-started`, in which Google suggests to create static URLs to index the pages that will not be reached by the crawler because of AJAX.

possible during early stages of the crawl, and postpone executing events that most probably lead to states that have already been visited.

To this end, we have introduced a general approach called *model-based* crawling [1, 8], where a crawling strategy aims at discovering the states of the application as soon as possible by making predictions based on an anticipated model for the application. In this paper, we explore two new strategies using the model-based crawling approach. The first one, called "Menu" model, uses a very simply paradigm for the crawled Web application: some events will always lead to the same client-state, no matter what the source client-state is. This new algorithm is simpler and more efficient to discover all reachable DOM states in a RIA than the other known strategies. The second one, called the "probability" model, can be seen as a generalization of the first one. In this approach, we try to compute the probability that a given event will lead to a new state at the next execution, based on how often it did lead to a new state during previous executions. The "probability" model is even simpler to implement, and does not have much overhead. It is comparable to the "menu" model in terms of efficiency in our test cases. It can also be efficiently distributed, and we describe a concurrent implementation of the model. Both models, as all model-based crawling strategies, are complete, which mean that they will eventually discover all possible client-states and all possible transitions between these client-states (under some assumptions described in Section 2). What distinguishes these strategies is not the model being eventually built, but how fast the model is built, and specifically how early in the process the client-states are found. The "menu" model was first introduced in [9] and the "probability" model in [10].

The rest of this paper is organized as follows: in Section 2, we give an overview of model-based crawling. In Section 3, we explain the "Menu" strategy in details. In Section 4, we explain the "Probability" strategy, and descrive a concurrent implementation for it. We then present the experimental study in Section 5 . In Section 6, a summary of related works is presented. In Section 7, we conclude the paper.

2 Problem Overview and Model-Based Crawling

2.1 Goals and Assumptions

The end goal is to build a "model" of a RIA; in our case, the model is a graph $G = (V, E)$ where the set of nodes V represents the client-states and the set of edges E represents the transitions between client-states, labeled by JavaScript events. Building such a model is potentially a very lengthy process, because of the large number of states and transitions involved. Because of this, many of the existing strategies do not try to build a complete model of the application being crawled. Our approach is different: we insist that under some assumptions and given enough time, the strategy should produce a complete model of the RIA (more specifically, given enough time any given client state or any given transition will be found). On the other hand, we acknowledge that, most of the time, we will not have enough time to complete the crawl, so we want to achieve as much as possible, as early as possible. Thus, we have two goals for our strategies:

1. Our first goal is to produce a complete model as efficiently as possible, which means that we want to minimize the number of events we need to execute to produce such a model.

2. Our second goal is that, as we produce this model, we should discover as many states as possible as early as possible during the crawl.

We say that a strategy is more *efficient* than another if it is better at reaching these two goals. The second goal is particularly important because, in most cases, it is more important to find the states than it is to find the transitions. Indeed, for many purposes, one is only interested in the states, not in the transitions. The transitions are only important as a means to show that we haven't missed any state. If we are not going to run the crawl to the end, we want to ensure that the partial model being built will contain as many states as possible.

In order to achieve the first goal above, the completeness of the model, we need to make some assumptions regarding the behaviour of the Web application being crawled.

1. The first assumption is that the RIA being crawled is deterministic from the point of view of the crawler. This means that, from the same state, the same action will always produce the same result (go to the same state). It is thus sufficient to execute each event from each state only once in order to infer a full model.

2. The second assumption is that we assume that if user inputs are involved, we have access to a collection of sample inputs that are good enough to build the model. We do not address here the question of how to generate such inputs. In practice, it means that the model we are building is complete with respect to this collection of sample inputs only.

3. Finally, we assume that we can always "reset" the RIA by reloading the initial URL, and thus the underlying application graph is strongly connected.

Note that these assumptions are rather strong, especially the first one: in reality, in practice even if the application is overall deterministic, it may not appear deterministic from the point of view of the client: the client doesn't have access to the server state, for example, so two client-states could be identical from the point of view of the client, but be in fact two different application-wide states. Although these assumptions are fairly commonly made in the literature, we recognize that they are limiting and that more work will have to be done to relax them in the future.

2.2 Model-Based Crawling

In general, it is not possible to devise a strategy that would be efficient at finding the states early, since the graph of the application could be any graph. We have introduced model-based crawling as a solution to this problem [1]. With model-based crawling, we work from a particular behavioural model, referred to as *meta-model.* This meta-model provides some indication on how the application will behave under some particular circumstances (the circumstances and the behaviour depend on the meta-model). An efficient (ideally, optimal) strategy is designed to crawl applications that are an instance of this meta-model. A second issue is to adapt the strategy to cope with "violations", that is, how to adapt the crawling strategy on-the-fly when the application does *not* behave as predicted by the meta-model. It is of course very important to deal with such violations, and deal with them efficiently if possible, since in practice, very few RIAs, if any, will be an exact instance of the meta-model.

Once such a strategy is defined, any RIA can be crawled with it. If the strategy is efficient, then the more the RIA behaves as predicted by the meta-model, the better the result. It is thus important to create meta-models that are realistic, in that they capture real behaviours of RIAs.

A model-based strategy usually consists of two phases:

1. **State exploration phase** where the objective is to discover all the application states as predicted by the meta-model of the strategy.

2. **Transition exploration phase** where the objective is to execute all remaining events, to complete the model and find states not predicted by the model.

It is possible that, during the second phase, new states are discovered (which is typically an instance of a violation), in which case we will switch back to the first phase. Thus, a model-based crawling strategy may alternate between these two phases multiple times before finishing the crawl. The strategy finishes the crawl when it has executed all the events in the application, which guarantees that all states of the application have been discovered.

Our first model-based crawling strategy is the "Hypercube" strategy where the meta-model predicts that when an event is executed from a state, this event is "consumed" and is not enabled in the resulting state, while the other events are still enabled. The meta-model also predicts that events are independent, that is, executing a subset of events on any order will always yield the same state. In that case, the RIA will have a hypercube structure [1]. The Hypercube strategy is an optimal strategy for RIAs that fully follow the hypercube meta-model. However, in practice, few RIAs follow this model, and the algorithms involved are rather complex. Even though the results were better than other strategies even for RIAs that do not follow this model, we present here two strategies that are better still, much easier to understand and that are based on meta-models more commonly found in RIAs. Refer to [8, 11] for a detailed overview of model-based crawling and the hypercube strategy.

3 The "Menu" Model

The "Menu" meta-model is based on the idea that some events will always lead the application to the same resulting state, regardless of the source state from which the event is executed. These kind of events are referred to as the "menu events". We called this model *menu model* because our menu events are often the intended model behind application menus. Such behaviour is realized by the menu items present in a web application such as *home, help, about us* etc.

Once an event is identified as a menu event, we can use it to anticipate some part of the application graph, and use this anticipated graph to build an efficient strategy. Thus, the core of the strategy is to identify these menu events, and then execute the events that are not menu events sooner than the menu events (since menu events are anticipated to produce known states). In practice, we prioritize the events based on the execution history:

1. **Globally unexecuted events**: This category represents the events that have not yet been executed at any state discovered so far. Events in this category have the highest priority.

2. **Locally unexecuted events**: This category represents the events that have been executed at some discovered state but have not been executed at the current state of the application. Events in this category are further divided into the following subcategories:

    (a) **Non-classified events**: Events in this subcategory have been executed only once globally. A second execution is necessary to classify the event. Events in this subcategory have the second highest priority next to the globally unexecuted events.

    (b) **Menu events**: Events are classified as menu events when their first two executions from two different states have led to the same state. They have the lowest priority.

    (c) **Self-Loop events**: Events are classified as self-loop events when their first two executions from two different states have not changed the state of the application. They have the same priority as the menu events.

    (d) **Other events**: All the remaining events belong to this category. These are the events that have shown neither menu nor self-loop behaviour in their first two executions. These events have the same priority as non-classified events.

Since the events in the menu and self-loop categories are not expected to lead to a new state, they have the lowest priority. The categorization of the events is done throughout the crawl. The priority sets are updated as new events are found in newly discovered states and as more information about results of the execution instances of the events become available.

3.1 State Exploration Phase

The primary goal of the state exploration phase is to discover all the states of the application as soon as possible. To do so, the strategy constructs a weighted directed graph model of the application An edge may be an executed event, a *reset*[c], or a *predicted* transition. For simplicity, we assume that each event has the same unit cost, but the cost of reset is different and it depends on the application being crawled. Predicted transitions correspond to unexecuted events that are classified either as menu event, in which case the predicted resulting state is the resulting state of the menu event, or as non-classified event, in which case the predicted resulting state is the state which was reached on the first execution of the event[d], or as self-loop event, in which case the predicted resulting state is the starting state of the self-loop edge. Figure 1 shows an instance of a graph.

The state exploration phase categorizes the events. Each event initially belongs to the globally unexecuted category. Unexecuted events are then picked according to the priority sets. All the instances of the events from a higher priority set are exhausted before executing an event from a lower priority set. Among the events within the same priority set, priority is given to the event which is closer to the current state than the others (closeness is in terms of number of transitions that are needed to be taken to reach a state where the event is enabled and unexecuted), otherwise one is chosen at random. During the state exploration phase, we execute all the unexecuted events in the application, except for categorized menu and self-loop events.

[c] A reset is the action of resetting the application to its initial state by reloading the URL.

[d] For the purpose of predicting the next state of transitions, all non-classified events are assumed to be menu events.

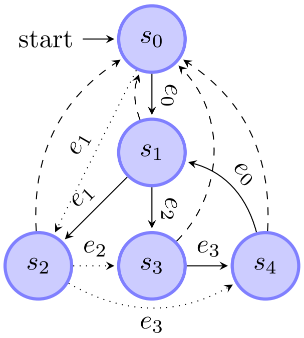

Fig. 1. An example of application graph G under construction: solid lines are executed transitions, dashed lines are resets, dotted lines are predicted transitions.

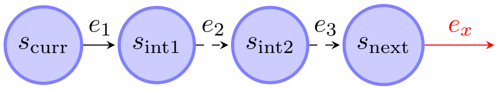

Fig. 2. Path from the current state to state s_{next} where the next event can be executed. Solid lines represent known transitions, and dotted lines represent predicted transitions.

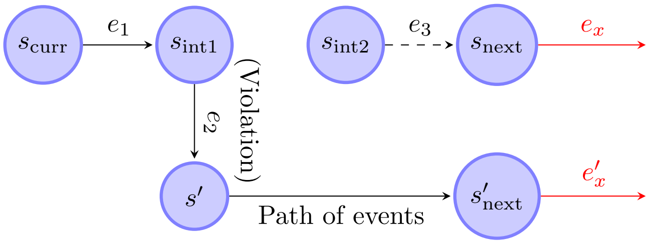

Fig. 3. Example of a violation for the path in Figure 2

Once an event is picked for execution, the shortest known path from the current state to the state where the event is going to be executed. This path may contain predicted transitions, which may of course be wrong, and the application may end up in a state that is not the predicted one. When this happens, the crawled RIA contradicts the menu model (at least from that state, and for this event). To adapt to such a violation, the strategy discards the current path and looks for the next unexecuted event from the state reached.

For instance, considering the execution of the example path shown in Figure 2, let us assume that there is a violation when the predicted transition e_2 (originating from s_{int1}) is taken. After executing event e_2 on s_{int1}, we reach state s' instead of s_{int2}. After this violation, the menu strategy builds a new path from the current state (s') to a next state with an unexecuted event.

During the execution of a path, traversing predicted transition leads to the execution of an event that had not been executed from that state before, which permits the categorization of that event if it is not already categorized.

3.2 Transition Exploration Phase

The state exploration phase executes all the events discovered during that phase, except for the events in the menu and the self-loop categories that have not been executed as part of a predicted edge on a path to the next selected event. Once the state exploration phase is over, the menu strategy moves to the transition exploration phase. The transition exploration phase verifies the validity of the assumptions made at the state exploration phase by executing all these remaining events. If the application follows the menu model, all the states of the application are found by the end of the state exploration phase. Any violating menu or self-loop events, however, may lead to the discovery of a new state.

During the transition exploration phase, the strategy tries to find the least costly path to execute all the remaining events in the application. The cost of this path is measured in terms of the total number of events and resets required. If we define *a walk of the graph* as a sequence of adjacent edges, the transition exploration problem can be mapped to the problem of finding the least costly walk of the graph that traverses all the edges representing the unexecuted events at least once.

3.2.1 Graph Walk

The transition exploration phase uses a walk generator function to calculate a walk that covers all of the unexecuted events. During the calculation of the graph walk, the application graph includes predicted transitions. Hence, executing the event sequence in the walk might not result in the expected state, in which case we have a violation as explained in Section 3.2.2. Considering the example in Figure 1 where the next state of all the unexecuted events have been assumed, a possible walk that covers every unexecuted event is the sequence $< e_1, e_3, e_0, e_1, e_2 >$, which starts at the initial state, s_0, and terminates at s_3.

Our immediate situation is similar to the problem known as the Rural Chinese Postman Problem (RCPP) [12] where, given a graph, we want a least cost tour covering only a subset of the edges. We need a least cost tour to execute all the remaining unexecuted events. Unfortunately, the RCPP is an NP-complete problem, so we do not attempt to solve this problem. Instead, we use the Chinese Postman Problem (CPP). In CPP, given a graph we want a least cost tour of all the edges. Unlike the RCPP, there are polynomial algorithms for the CPP. However, this is not a perfect analogy to our situation: in the current graph, we have both executed and predicted transitions, and we only want to execute the predicted ones. If we consider the subgraph containing only the predicted transitions, this subgraph may not be connected, and a tour may not exist. To address this problem, we augment this subgraph with a few of the known transitions (including resets if necessary), until the graph is strongly connected again. We then use a CPP algorithm to create a tour that goes over all transitions (see [13] for details). This solution gives reasonably good results (although clearly non optimal) at a small computational cost.

3.2.2 Violation and Strategy Adaptation

When going over the tour, each predicted transition may lead to a violation of the assumption, and the application can end up in a state that is not the one predicted. There are two cases to handle :

1. **Wrong known state:** This is the case where the resulting state has been discovered previously, but it is not the expected one. When this happens, the predicted edge is removed from the graph, replaced with the newly executed transition. At this point, we end up in a wrong state in the tour. Instead of recomputing a tour, we have opted for a simpler solution: the strategy keeps the original walk, and brings the application back to the state that was expected to be reached initially. To do this, we simply find the

shortest known path that does not contain any predicted edges from the current state to that next state, and execute it first.

2. **New state:** Should a violation lead to the discovery of a new state, the crawling strategy switches back to the state exploration phase if there are events unclassified as "other events". Otherwise, the existing CPP is augmented to include any additional discovered unexecuted events.

4 The "Probability" Model

The Menu model explained above is rather simple to understand, but the implementation is not necessarily immediate, in particular when it comes to the computation of the graph walk described in Section 3.2.1. Moreover, the classification of events is not very specific: set of "other events" contains anything that is not menu or self loop. Finally, it may be challenging to distribute this strategy over several concurrent crawlers, especially when in comes to handling the violations (Section 3.2.2). The "probability" model helps with these issues.

The Probability strategy is a strategy based on the belief that an event which was often observed to lead to new states in the past is more likely to lead to new states in the future. In this strategy an event's probability of discovering a new state on its next exploration is estimated based on how successful it was in its past explorations and this estimated probability is used as a priority measure.

The Probability Strategy chooses the next event to explore considering two aspects of an event: the event's estimated probability of discovering a new state on the next exploration, and the length of the shortest transfer sequence to a state where the event can be executed.

4.1 Probability of an Event

Using the so-called "rule of succession" [14], we can estimate the probability, $P(e)$, of discovering a new state on an event e's next exploration using the following Bayesian formula:

$$P(e) = \frac{S(e) + p_s}{N(e) + p_n} \tag{1}$$

- $N(e)$ is the "exploration count" of e, that is, the number of times e has been explored (executed from different states) so far,
- $S(e)$ is the "success count" of e, that is, the number of times e discovered a new state out of its $N(e)$ explorations,
- p_s and p_n are called the pseudo-counts and they represent the "initial success count"' and the "initial exploration count", respectively. These terms are preset and define the a priori (initial) probability for an event.

To use this formula we assign values to p_s and p_n to set an initial probability. For example, $p_s = 1$ and $p_n = 2$ can be used to set an event's initial probability to 0.5 (note that $N(e) = S(e) = 0$ initially). After each exploration of an event, the event's probability is updated by taking into account the result of this recent exploration.

4.2 Choosing The Next Event to Explore

When selecting the next event to explore, simply choosing an event with the highest probability is not efficient, since that event may be some distance away from the current state. To make a decision, we should also take into account the length of the transfer sequence to be executed to reach the candidate state.

For a state $s \in S$, we define the probability of the state, $P(s)$, as the probability of the event that has the highest probability among all the unexplored events of s. If s has no unexplored event then $P(s) = 0$. $l_T(s)$ is the length of the shortest known transfer sequence from the current state s_{current} to s.

From a given current state, we therefore have the task of selecting among the neighbourhood a state s with high value for $P(s)$ and a small value for $l_T(s)$. This task is non trivial in the case that there is a state s_1 with a higher probability than another state s_2, but farther away from the current state. In order to determine which of the two states is more interesting, we consider the difference, $k = l_T(s_1) - l_T(s_2)$, of the lengths of the transfer sequences and we calculate the "k-iterated probability" of s_2 (the state that is closer to the current state), noted as $P(s_2, k)$. We define $P(s_2, k)$ as the probability of discovering a new state by choosing s_2 and exploring k more events after the event in s_2 is explored.

In order to calculate $P(s, k)$ for a state s, we need to estimate the probabilities of the states that we will traverse as we execute the k events. Assuming that reaching any state is equally likely, we can compute the average probability of all the state P_{avg} as

$$P_{\text{avg}} = \frac{\sum_{s \in S} P(s)}{|S|} \tag{2}$$

Using this estimation for P_{avg}, it is easy to compute $P(s, k)$:

$$P(s, k) = 1 - (1 - P(s))(1 - P_{\text{avg}})^k \tag{3}$$

In plain words, this value is one minus the probability of not discovering any state by choosing s and then not discovering any state by exploring k more events (each with probability P_{avg}).

The strategy thus consists of choosing the state with the highest $P(s, k)$ from the current state, then transfer to that state, execute the event with the highest probability on that state, and update the set of probabilities accordingly. The strategy stops once all transitions have been executed.

In practice, it is not necessary to always compute $P(s, k)$ for all pair of states. See [11] for an algorithm implementing this strategy with an overall complexity of $O(|E|^2 + |E||V| \log n)$.

4.3 Distributing the Probability Strategy Across Concurrent Crawlers

In order to speed up the execution of the strategy, we can distribute the work over several concurrent processes. In our context, based on our experiments, the most time consuming task is events execution, and subsequent DOM update and analysis. Consequently, we have implemented a distributed version of the algorithm that works as follows: a centralized coordinator collects the information found so far and builds the application graph. A certain number of "crawlers" are in charge of executing the events and report back the results (new states, new events found) to the coordinator. In this distributed version, each crawler implements its own version of the probability strategy, so the computation of the strategy is distributed as well.

The coordinator is in charge of informing the crawlers about the graph as it is being built. Based on that knowledge, each crawler decides what event to execute next (using the probability strategy), and reports back the result of the execution to the coordinator. The two activities are carried out concurrently: every time a crawler executes an event, it sends the result of that execution to the coordinator using an asynchronous message (which we assume here to be reliable) and continues the crawl by choosing the next event to execute based on the probability strategy. On the other hand, the coordinator accumulates global information about the graph and, from time to time, informs the crawlers about it.

In order to avoid duplication of work, each event should be assigned to a single crawler, who is in charge of executing that event [15]. For efficiency reasons, this should be done transparently, without requiring the crawlers to coordinate among themselves. This can be achieved by assigning the ownership of an event as a function of the identifier of the crawler. The difficult part, for the probability strategy, is to efficiently compute the probability of an event as well as choosing the next event to execute, without centralizing the decision nor introducing coordination between the crawlers.

Because the probability of an event is based on the result of that event's execution across several states, a good way to distribute the algorithm is to assign an event to a single crawler, so that one crawler is in charge of the execution of all instances of that event across all states in which that event is enabled. This way, the crawler in charge of that event can easily compute the probability of that event locally using Equation (1).

Computing the next event to execute is more difficult, since a given crawler may not be the "owner" of the event that has the best probability. In order to resolve that issue, we adapt the probability strategy by restricting the computation of the next event to execute to the events that are assigned to the crawler. In other words, our distributed implementation of the strategy can be seen as a partitioning of the events among crawlers, with each crawler computing the original probability strategy on the subset of the events assigned to it.[e] The concurrent implementation is thus not the exact replica of the centralized version, but a close approximation.

The last point to clarify is the partitioning of the events amongst crawlers. We use an approach where each crawler is assigned an identifier between 0 and $n-1$ (for n crawlers). Each event identifier is hashed into a integer value h, and the value $k = h \bmod n$ is computed. The event is assigned to crawler k. That computation can be done independently and locally by each crawler, as well as by the coordinator.

5 Implementation and Evaluation

In this section, we present our experimental results, comparing the efficiency of the Menu and Probability strategies against many other existing crawling strategies on several AJAX-based RIAs.

[e] Each crawler can still execute an event assigned to another crawler once the coordinator did broadcast the information about the result of the execution of that event.

5.1 Measuring the Efficiency of a Strategy

As explained before, our definition of an efficient strategy is a strategy that builds the entire model quickly, while finding the states as early as possible in the process. In order to determine efficiency, measure speed, instead of measuring time, we measure the number of event executions and the number of resets required by each strategy to complete two tasks: finding the states and finding the complete model. This is reasonable since the time spent for event executions and resets dominates the crawling time and the numbers depend only on the decisions of the strategy. And this way, the results do not depend on the hardware that is used to run the experiments and are not affected by the network delays which can vary in different runs.

In order to determine the relative cost of event executions and resets, we measure for each application the following two values. $t(e)_{avg}$: the average event execution time obtained by measuring the time for executing the events in a randomly selected set of events in the application and taking the average, and $t(r)_{avg}$: the average time to perform a reset. For simplicity, we assume that each event execution takes $t(e)_{avg}$ and this becomes our cost unit. Then, we calculate "the cost of reset": $c_r = t(r)_{avg}/t(e)_{avg}$. Finally, the cost of a strategy is calculated by $n_e + n_r \times c_r$ where n_e and n_r are the total number of events executed and resets used by the strategy, respectively.[f]

5.2 Other Crawling Strategies Used for Comparison

- Optimized Standard Crawling Strategies: The standard crawling strategies are Breadth-First and Depth-First. We use "optimized" versions of these strategies, meaning that when there is a need to move from the current state to another known state, the shortest known path from the current state to the desired state is used. This is in contrast to using systematic resets. The results presented here with the optimized versions are much better than the ones obtained using the standard, non-optimized Breadth-First and Depth-First strategies.

- Greedy Strategy [16]: This is a simple strategy that prefers to explore an event from the current state, if there is one. Otherwise, it chooses an event from a state that is closest to the current state.

- Another Model-based Crawling Strategy: The Hypercube strategy [1] is based on the anticipation that the application has a hypercube model.

- The Optimal Cost: We also present the optimal cost of discovering all the states for each application. This is not a strategy. This cost is calculated once the model is known. Finding an optimal path that visits all states in a known graph is the Asymmetric Traveling Salesman Problem (ATSP). We use an exact ATSP solver [17] to find this path. This gives us an idea of how far from the optimal speed each strategy is (for the first phase, find all the states).

[f] We measure the value of c_r before crawling an application and give this value as a parameter to each strategy. A strategy, knowing how costly a reset is compared to an average event execution, can decide whether to reset or not when moving from current state to another known state.

	Name	*#states*	*#transitions*	Cost of Reset
Real Applications	Bebop	1,800	145,811	2
	Elfinder	1,360	43,816	10
	FileTree	214	8,428	2
	Periodic Table	240	29,034	8
	Clipmarks	129	10,580	18
	DynaTable	448	5,376	19
Test Applications	TestRIA	39	305	2
	Altoro Mutual	45	1,210	2

Table 1. Applications tested, along with their number of states, transitions, and the cost of reset.

5.3 Applications Tested

We compare the strategies using two test RIAs and six real RIAs.[g] This number is not as large as we would like, but we are limited by the tools that are available to us.[h] With the increasing exposure to this problem, better tools will be made available, and we will be able to test our strategies on a much broader test set. The following table provides, for each test RIA, the number of states, transitions as well as the cost of reset.

5.4 Experimental Setup

We have implemented all the mentioned crawling strategies in a prototype of IBM® Security AppScan® Enterprise.[i] Each strategy is implemented as a separate class in the same code base, so they use the same DOM equivalence mechanism [18], the same event identification mechanism [7], and the same embedded browser. For this reason, all strategies find the same model for each application.

We crawl each application with a given strategy ten times and present the average of these crawls. In each crawl, the events of each state are randomly shuffled before they are passed to the strategy. The aim here is to eliminate influences that may be caused by exploring the events of a state in a given order since some strategies may explore the events on a given state sequentially.

5.5 Costs of Discovering States (Strategy Efficiency)

The box plots in Figure 4 show the results. For each application and for each strategy, the figure contains a box plot: the lower edge, the line in the middle and the higher edge of the box show the cost of discovering 25%, 50% and 75% of the states, respectively. The maximum point of the line shows the cost of discovering all the states. The plots are drawn in logarithmic scale for better visualization. Each horizontal dotted line shows the optimal cost for the corresponding application.

The results show that for almost all applications, the menu and the probability models are more efficient than the other strategies, sometimes by a wide margin. It can be seen that

[g] `http://ssrg.eecs.uottawa.ca/testbeds.html`

[h] We stress that the work in question is not related to the strategy described here, but to the limitation of the available tools.

[i] Details are available at `http://ssrg.eecs.uottawa.ca/docs/prototype.pdf` Since our crawler is built on top of the architecture of a commercial product, we are not currently able to provide open-source implementations of the strategies.

the Menu strategy has the best performance to discover all the states except for the Bebop application where it is beaten by the Probability strategy. In fact, in all cases but for the periodic table, both menu and probability strategy beat the other strategies to discover 75% and 100% of the states (and 50% as well, except the Probability strategy on DynaTable). This is particularly important if one assumes that the crawl will not be executed to the end and that in most cases it will be cut short. It shows that the Menu and the Probability strategies will provide the most information after the least amount of time, with a slight edge to the Menu strategy.

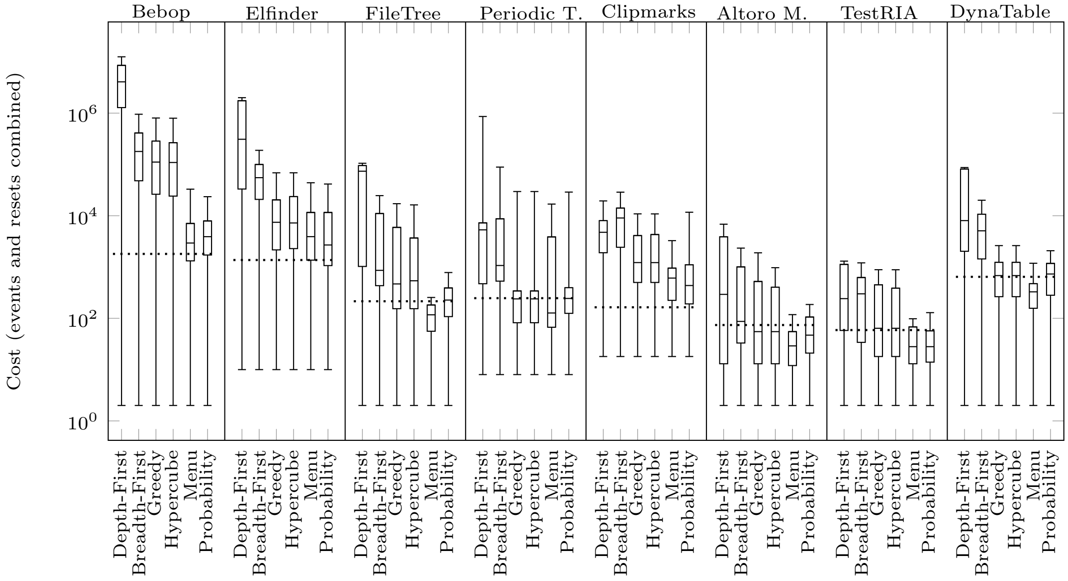

Fig. 4. Costs of Discovering the States (Strategy Efficiency), in logarithmic scale. Each horizontal dotted line shows the optimal cost for the corresponding application.

5.6 Costs of the Complete Crawl

The previous results show the costs of discovering all the states. However, the crawl does not end at this point since a crawler cannot know that all states are discovered until all events are explored from each state (in other words, we could provide this information only because we have run the tests to the end). In Table 2, the total number of events and the total number of resets during the crawl are shown as well as the costs calculated based on these numbers.

It can be seen that the model-based strategies and the Greedy strategy finish crawling with significantly less cost compared with Breadth-First and Depth-First. The strategies presented here are in the same ballpark but not significantly better than the Hypercube model-based or greedy strategies. However, the complete crawl is not as important a factor as finding all the states, as explained before.

5.7 Time measurements

In Sections 5.5 and 5.6, the costs are expressed in terms of number of event executions and resets, which is independent from external factors such as network conditions or load on the

	Bebop			FileTree			Periodic Table			Clipmarks		
	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost
Depth-First	13,386,210	27	13,386,264	99,336	13	99,362	897,358	236	899,246	19,569	72	20,868
Breadth-First	943,001	8,732	960,466	26,375	1,639	29,652	64,850	14,633	181,916	15,342	926	32,015
Greedy	826,914	27	826,968	20,721	13	20,747	29,926	236	31,814	11,396	56	12,398
Hypercube	816,142	27	816,196	19,865	13	19,891	29,921	236	31,809	11,350	56	12,356
Probability	816,922	27	816,976	19,331	13	19,357	29,548	236	31,436	11,456	62	12,563
Menu	814,220	27	814,274	19,708	13	19,734	37,489	236	39,377	11,769	71	13,043
	Altoro Mutual			TestRIA			Elfinder			DynaTable		
	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost
Depth-First	6,876	34	6,944	1,433	1	1,435	1,998,329	194	2,000,559	156,499	1	156,518
Breadth-First	3,074	334	3,742	1,216	55	1,326	115,702	7,033	186,028	20,368	1512	49,096
Greedy	2,508	34	2,576	1,001	1	1,003	67,187	190	69,085	9,059	1	9,078
Hypercube	2,489	34	2,557	994	1	996	67,453	195	69,402	9,059	1	9,078
Probability	2,451	34	2,520	972	1	974	49,731	134	51,076	9,208	1	9,227
Menu	2,457	35	2,527	974	1	976	48,438	203	50,466	11,192	1	11,211

Table 2. Total Costs of Complete Crawling

	Bebop	FileTree	Periodic Table	Clipmarks	Altoro Mutual	TestRIA	Elfinder	DynaTable
Depth-First	93 : 42 : 14	01 : 43 : 06	05 : 37 : 45	00 : 14 : 44	00 : 02 : 37	00 : 00 : 27	55 : 15 : 56	01 : 10 : 15
Breadth-First	09 : 00 : 31	00 : 38 : 29	00 : 21 : 48	00 : 07 : 17	00 : 00 : 54	00 : 00 : 25	04 : 42 : 47	00 : 43 : 45
Greedy	08 : 17 : 07	00 : 29 : 57	00 : 14 : 44	00 : 04 : 11	00 : 00 : 44	00 : 00 : 20	02 : 18 : 56	00 : 02 : 07
Hypercube	08 : 10 : 36	00 : 29 : 10	00 : 15 : 20	00 : 04 : 12	00 : 00 : 24	00 : 00 : 19	02 : 19 : 29	00 : 02 : 01
Probability	00 : 12 : 50	00 : 00 : 36	00 : 14 : 15	00 : 04 : 32	00 : 00 : 06	00 : 00 : 04	01 : 09 : 14	00 : 01 : 25
Menu	00 : 18 : 49	00 : 00 : 15	00 : 08 : 49	00 : 01 : 55	00 : 00 : 04	00 : 00 : 03	02 : 01 : 29	00 : 00 : 56

Table 3. Total time required to find all states (hh:mm:ss).

computer. For completeness, we provide here some timing information as well. In Table 3 we provide time measurements for finding all the states (corresponding to Figure 4), and in Table 4 the time required to complete the crawl (corresponding to Table 2). The experiments where executed on a a laptop equipped with an Intel$^{\circledR}$, Core$^{\text{TM}}i7$ processor and 4MB of RAM, accessing copies of the RIAs installed on a local network.

As expected, the time measurements reflect the fact that strategies which execute less events and less resets often require proportionately less time. The major source of difference between the time measurements and our cost measurements is our simplifying assumption that each event execution takes the same average time. As a result of this simplification, for Clipmarks our measured value for cost of reset is a bit higher than the real time delays incurred by the resets during crawling. This is why the Breadth-First has more cost than Depth-First (since Breadth-First uses many resets), but actually Breadth- First finishes earlier than Depth-First. Except for this case, the cost measurements are in line with the time measurements. In addition, since the Menu strategy calculates a Chinese Postman Path for its transition exploration phase, for larger models (Bebop, ElFinder) the Menu strategy requires more time to complete the crawl than the other strategies that execute a comparable number of events and resets.

	Bebop	FileTree	Periodic Table	Clipmarks	Altoro Mutual	TestRIA	Elfinder	DynaTable
Depth-First	97 : 05 : 22	01 : 43 : 28	05 : 38 : 02	00 : 14 : 52	00 : 12 : 00	00 : 00 : 34	55 : 17 : 30	02 : 04 : 47
Breadth-First	09 : 04 : 57	00 : 43 : 05	00 : 42 : 19	00 : 08 : 14	00 : 15 : 07	00 : 00 : 27	04 : 53 : 40	00 : 44 : 05
Greedy	08 : 32 : 04	00 : 33 : 42	00 : 15 : 06	00 : 04 : 28	00 : 04 : 28	00 : 00 : 21	02 : 21 : 47	00 : 06 : 44
Hypercube	08 : 22 : 54	00 : 33 : 02	00 : 15 : 42	00 : 04 : 29	00 : 04 : 19	00 : 00 : 21	02 : 22 : 05	00 : 06 : 31
Probability	07 : 54 : 03	00 : 31 : 37	00 : 14 : 42	00 : 04 : 39	00 : 04 : 46	00 : 00 : 21	02 : 03 : 43	00 : 06 : 08
Menu	12 : 48 : 55	00 : 33 : 40	00 : 20 : 06	00 : 04 : 22	00 : 04 : 24	00 : 00 : 21	08 : 23 : 26	00 : 11 : 02

Table 4. Total time required to finish crawling (hh:mm:ss).

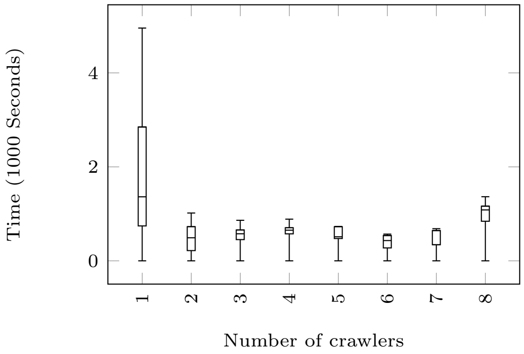

Fig. 5. Cost of discovering FileTree application states with multiple nodes.

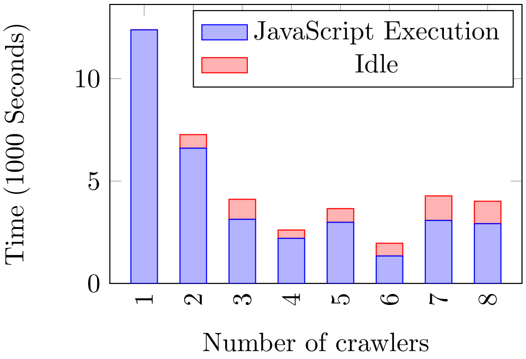

Fig. 6. The total time to Crawl the FileTree application with multiple nodes.

5.8 Distributed Crawl

Figure 5 show the cost of discovering states in the FileTree application in distributed mode. As can be seen, distributing the work reduces the time it takes to discover application states up to a certain point. Figure 6 shows the total time needed to crawl the application. As the number of crawlers increases, a good speed up is achieved until six crawlers and it starts degrading afterwards. Several possible factors contributing to this degradation are the coordinator becoming slower to distribute the knowledge of the graph among the nodes, so the path computed by the crawlers is not as good and the events that are split between too many crawlers, so each crawler does not have enough data to compute statistically significant probabilities. Distributing the probability strategy is an effective solution to speed up the crawl, but the maximum number of useful crawlers that can be used is driven by the number of events in the application and can be relatively small.

6 Related Works

A survey of traditional crawling techniques is presented in [2]. For RIA crawling, recent surveys are presented in [6] and [3]. Except for [1, 8, 10, 18] which present other model-based crawling strategies and [16] which presents the Greedy strategy, the published research uses Breadth-First or Depth-First strategies for crawling RIAs. As we have seen, Breadth-First and Depth-First strategies are less efficient than the Greedy and the model-based strategies.

[19] and [20] suggest algorithms to index a RIA. [20] offers an early attempt in crawling AJAX applications based on user events and building the model of the application. The application model is constructed as a graph using the Breadth-First strategy. [19] introduces an AJAX-aware search engine for indexing the contents of RIAs. In this model components are adapted to handle RIAs. The crawler identifies JavaScript events and runs a standard Breadth-First search on them. [21] offers an algorithm, called *AjaxRank*, similar to *PageRank* [22] tailored to RIAs, to give weight to different states based on the connectivity. *Dist-RIA Crawler* suggests an event-based partitioning algorithm to run a distributed Breadth-First strategy [15]. Wu and Liu [23] seek to reduce the number of JavaScript events executed to crawl an application. They do so by using HTML elements XPaths expressions as an

indication to weather an element has been clicked before or not. In a more advanced attempt Moosavi [24] identifies independent client-side widgets. In this model, called *Component-based Crawling*, a state is not defined as state of DOM, but as an individual independent widget.

[25–27] seek to automate regression and other testing of a RIA. *Crawljax* [28,29] constructs a state-flow graph of the application by exercising client-side code and identifying the events that change the state of the application. Crawljax differentiates states using Levenshtein distance method [30], and uses a Depth-First strategy. [31] describes the derivation of test sequences from the application model obtained by crawling. [32] is similar, but takes a white-box testing approach where the program fragments of the states are analyzed. *PYTHIA* [33] generates unit tests at JavaScript function level using a greedy strategy to maximize code coverage. It then generate test oracles using mutation testing.

FeedEx [34] extends [16] by defining a matrix to measure the impact of executing an event. Four factors to prioritize the events in FeedEx matrix are: code coverage, navigational diversity, page structural diversity and test model size. Event execution order is derived by calculating different events impact priority and sorting it.

Several other tools exist to create a graph model of the application. *RE-RIA* [35] uses execution traces to create the graph model of the application. As an improvement to *RE-RIA*, *CrawlRIA* [36] generates the execution traces by running a Depth-First strategy. *CreRIA* [37] facilitate reverse engineering of a RIA for dynamic analysis. *DynaRIA* [38] offers a tool to comprehend a RIA better for testing and other purposes. It also helps to visualize the run-time behavior of the application. More generally, the concept of "GUI ripping" is introduced in [39]. This is a different, but related question to the one we are addressing here. Some efforts are being done to apply to mobile applications what is done on RIAs. This field is quite new, but some work has already been done in the context of model inference for native android apps [40, 41] and native iOS apps [42].

In the context of session transfer, *Imagen* [43] improves definition of RIAs client-side state by considering factors such as: JavaScript functions closure, JavaScript event listeners, and HTML5 elements such as *Opaque Objects* and *Stream Resources.* As [43] explains, these factors are not reflected in the DOM and require extra effort to be reflected into client-side session.

7 Conclusion and future studies

Two new model-based crawling algorithms were introduced: the Menu model and the Probability model. We models RIAs as a graph, based on the JavaScript events in each state of the DOM. The approach consists of making assumptions about the category of events in order to derive a strategy, then learn, and adapt these categories as the crawling proceeds. In the Menu model, the events are assumed to either always lead to the same state (menu events), or to always lead to new states. In the probability strategy, an actual probability of leading to a new state is calculated for each event, based on how successful it was in the past at finding new states. A concurrent implementation of the probability model was also described. A prototype implementation for these strategies was built and the performance of these strategies is compared with several other algorithms. We have shown empirically that our new strategies are better than other known strategies when it comes to finding all the states of the application being modeled. We have also evidences showing that the concurrent implementation of the probability strategy provides an good additional speed up, opening the door to fast crawling of RIA using concurrent crawlers.

Some directions for future studies are:

- *Better definition of states*: Learning about independent widgets, as suggested by [24], can substantially improve performance of both Menu and Probabilistic model by reducing the number of events to execute.
- *Adaptive Crawling*: starting from a *generic* meta-model and recognizing the meta model of a web-application on-the-fly is another area of improvement.
- *Relaxing assumptions*: In future work, it will be important to relax or waive some of the assumptions made in regard to this work. In particular the assumptions that the applications is deterministic from the viewpoint of the model builder.

Acknowledgments

This work is partially supported by the IBM Center for Advanced Studies, and the Natural Sciences and Engineering Research Council of Canada (NSERC). The views expressed in this article are the sole responsibility of the authors and do not necessarily reflect those of IBM.[j] A special thank to Salman Hooshmand, Sara Baghbanzadeh and Ali Moosavi.

References

1. Kamara Benjamin, Gregor von Bochmann, Mustafa Emre Dincturk, Guy-Vincent Jourdan, and Iosif Viorel Onut. A strategy for efficient crawling of rich internet applications. In *Proceedings of the 11th international conference on Web engineering*, ICWE'11, pages 74–89, 2011.
2. Christopher Olston and Marc Najork. Web crawling. *Found. Trends Inf. Retr.*, 4(3):175–246, March 2010.
3. Seyed M Mirtaheri, Mustafa Emre Dincturk, Salman Hooshmand, Gregor V Bochmann, Guy-Vincent Jourdan, and Iosif-Viorel Onut. A brief history of web crawlers. In *CASCON*, 2013.
4. World Wide Web Consortium (W3C). Document Object Model (DOM). `http://www.w3.org/DOM/`, 2005. [Online].
5. Jesse James Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/publications/essays/archives/000385.php`, 2005. [Online].
6. Suryakant Choudhary, Mustafa Emre Dincturk, Seyed M. Mirtaheri, Ali Moosavi, Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut. Crawling rich internet applications: the state of the art. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '12, pages 146–160, 2012.
7. Suryakant Choudhary, Mustafa Emre Dincturk, Gregor V. Bochmann, Guy-Vincent Jourdan, Iosif Viorel Onut, and Paul Ionescu. Solving some modeling challenges when testing rich internet applications for security. *Software Testing, Verification, and Validation, 2012 International Conference on*, pages 850–857, 2012.
8. Mustafa Emre Dincturk, Guy-Vincent Jourdan, Gregor von Bochmann, and Iosif Viorel Onut. A model-based approach for crawling rich internet applications. *ACM Transactions on the WEB*, page to appear, 2014.

[j] **Trademarks:** IBM and AppScan are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at *Copyright and trademark information* at www.ibm.com/legal/copytrade.shtml. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

9. Suryakant Choudhary, Mustafa Emre Dincturk, Seyed M. Mirtaheri, Guy-Vincent Jourdan, Gregor v. Bochmann, and Iosif Viorel Onut. Building rich internet applications models: Example of a better strategy. In Florian Daniel, Peter Dolog, and Qing Li, editors, *Web Engineering*, volume 7977 of *Lecture Notes in Computer Science*, pages 291–305. Springer, 2013.
10. Mustafa Emre Dincturk, Suryakant Choudhary, Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut. A statistical approach for efficient crawling of rich internet applications. In *Proceedings of the 12th international conference on Web engineering*, ICWE'12, pages 74–89, 2012.
11. Mustafa Emre Dincturk. *Model-based Crawling - An Approach to Design Efficient Crawling Strategies for Rich Internet Applications.* PhD thesis, EECS - University of Ottawa, 2013. `http://ssrg.site.uottawa.ca/docs/Dincturk_MustafaEmre_2013_thesis.pdf`.
12. H. A. Eiselt, Michel Gendreau, and Gilbert Laporte. Arc routing problems, part ii: The rural postman problem. *Operations Research*, 43(3):pp. 399–414, 1995.
13. Suryakant Choudhary. M-crawler: Crawling rich internet applications using menu meta-model. Master's thesis, EECS - University of Ottawa, 2012. `http://ssrg.site.uottawa.ca/docs/Surya-Thesis.pdf`.
14. Sandy L. Zabell. The rule of succession. *Erkenntnis*, 31:283–321, 1989.
15. Seyed M. Mirtaheri, Di Zou, Gregor V. Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut. Dist-ria crawler: A distributed crawler for rich internet applications. In *In Proc. 8TH International Conference On P2P, Parallel, Grid, Cloud And Internet Computing*, 2013.
16. Zhaomeng Peng, Nengqiang He, Chunxiao Jiang, Zhihua Li, Lei Xu, Yipeng Li, and Yong Ren. Graph-based ajax crawl: Mining data from rich internet applications. In *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, volume 3, pages 590 –594, march 2012.
17. G. Carpaneto, M. Dell'Amico, and P. Toth. Exact solution of large-scale, asymmetric traveling salesman problems. *ACM Trans. Math. Softw.*, 21(4):394–409, December 1995.
18. Kamara Benjamin, Gregor v. Bochmann, Guy-Vincent Jourdan, and Iosif-Viorel 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, pages 403–409, Washington, DC, USA, 2010. IEEE Computer Society.
19. Cristian Duda, Gianni Frey, Donald Kossmann, and Chong Zhou. Ajaxsearch: crawling, indexing and searching web 2.0 applications. *Proc. VLDB Endow.*, 1(2):1440–1443, August 2008.
20. Cristian Duda, Gianni Frey, Donald Kossmann, Reto Matter, and Chong Zhou. Ajax crawl: Making ajax applications searchable. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 78–89, Washington, DC, USA, 2009. IEEE Computer Society.
21. Gianni Frey. Indexing ajax web applications. Master's thesis, ETH Zurich, 2007. `http://e-collection.library.ethz.ch/eserv/eth:30111/eth-30111-01.pdf`.
22. Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web, 1998. Standford University, Technical Report.
23. Guoshi Wu and Fanfan Liu. Web crawler for event-driven crawling of ajax-based web applications. In *Emerging Technologies for Information Systems, Computing, and Management*, pages 191–200. Springer, 2013.
24. Ali Moosavi. Component-based crawling of complex rich internet applications. Master's thesis, EECS - University of Ottawa, 2014. `http://ssrg.site.uottawa.ca/docs/Ali-Moosavi-Thesis.pdf`.
25. Danny Roest, Ali Mesbah, and Arie van Deursen. Regression testing ajax applications: Coping with dynamism. In *ICST*, pages 127–136. IEEE Computer Society, 2010.
26. Cor-Paul Bezemer, Ali Mesbah, and Arie 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, pages 81–90, New York, NY, USA, 2009. ACM.
27. Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In

Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, pages 210 –220, may 2009.

28. Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling ajax by inferring user interface state changes. In *Proceedings of the 2008 Eighth International Conference on Web Engineering*, ICWE '08, pages 122–134, Washington, DC, USA, 2008. IEEE Computer Society.
29. Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the WEB*, 6(1):3, 2012.
30. VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.
31. Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of ajax web applications. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 121–130, Washington, DC, USA, 2008. IEEE Computer Society.
32. Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of JavaScript web applications. In *Proc. 33rd International Conference on Software Engineering (ICSE)*, May 2011.
33. Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Pythia: Generating test cases with oracles for javascript applications. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE), New Ideas Track*, pages 610–615. IEEE Computer Society, 2013.
34. Amin Milani Fard and Ali Mesbah. Feedback-directed exploration of web applications to derive test models. In *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, page 10 pages. IEEE Computer Society, 2013.
35. Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Reverse engineering finite state machines from rich internet applications. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, WCRE '08, pages 69–73, Washington, DC, USA, 2008. IEEE Computer Society.
36. Domenico Amalfitano, Anna Rita Fasolino, and Porfirio 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, pages 274–283, Washington, DC, USA, 2010. IEEE Computer Society.
37. Creria. `http://wpage.unina.it/ptramont/downloads.htm`.
38. Domenico Amalfitano, Anna Rita Fasolino, Armando Polcaro, and Porfirio Tramontana. Dynaria: A tool for ajax web application comprehension. In *ICPC*, pages 46–47. IEEE Computer Society, 2010.
39. Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, November 2003.
40. Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A gui crawling-based technique for android mobile application testing. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 252–261, Washington, DC, USA, 2011. IEEE Computer Society.
41. Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.
42. M. Erfani and A. Mesbah. Reverse engineering ios mobile applications. In *19th Working Conference on Reverse Engineering, (WCRE'12)*, 2012.
43. James Lo, Eric Wohlstadter, and Ali Mesbah. Imagen: Runtime migration of browser sessions for javascript web applications. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 815–825. ACM, 2013.