A Strategy for Efficient Crawling of Rich Internet Applications

Kamara Benjamin[1], Gregor von Bochmann[1], Mustafa Emre Dincturk[1], Guy-Vincent Jourdan[1], and Iosif Viorel Onut[2]

[1] SITE, University of Ottawa. 800 King Edward Avenue, K1N 6N5, Ottawa, ON, Canada
[2] Research and Development, IBM® Rational® AppScan® Enterprise, IBM, 1 Hines Rd, Ottawa, ON, Canada
{bochmann,gvj}@site.uottawa.ca,
{kbenj067,mdinc075}@uottawa.ca,
vioonut@ca.ibm.com

Abstract. New web application development technologies such as Ajax, Flex or Silverlight result in so-called Rich Internet Applications (RIAs) that provide enhanced responsiveness, but introduce new challenges for crawling that cannot be addressed by the traditional crawlers. This paper describes a novel crawling technique for RIAs. The technique first generates an optimal crawling strategy for an anticipated model of the crawled RIA by aiming at discovering new states as quickly as possible. As the strategy is executed, if the discovered portion of the actual model of the application deviates from the anticipated model, the anticipated model and the strategy are updated to conform to the actual model. We compare the performance of our technique to a number of existing ones as well as depth-first and breadth-first crawling on some Ajax test applications. The results show that our technique has a better performance often with a faster rate of state discovery.

Keywords: Rich Internet Applications, Crawling, Web Application Modeling.

1 Introduction

Web Applications have been used for nearly as long as the Web itself. In the formative years, a Web Application was an application running entirely on the server side, and which used a dynamically created web page rendered inside a classical web browser as a client. Following the model of static Web sites, these web applications used the usual GET and POST HTTP requests to communicate between the client and the server, and each response sent by the server was meant as a complete replacement of the currently displayed client side. These applications were not very user-friendly due to the lack of flexibility at the client side, but they were pretty similar to static web sites in terms of crawling (except for possible user inputs): it was sufficient to scan the current page to find out what the possible next states could be, and the current state was entirely defined by the last server response. Crawling these web

S. Auer, O. Diaz, and G.A. Papadopoulos (Eds.): ICWE 2011, LNCS 6757, pp. 74–89, 2011.
© Springer-Verlag Berlin Heidelberg 2011

applications was easy, and many commercial tools can do it, e.g. for the purpose of automated testing, usability assessment, security evaluation or simply content indexing.

This situation was first modified with the widespread support in Web browsers for scripting languages such as JavaScript, and the ability given to these scripts by browsers to modify directly the current page (the current DOM). After this, it was no longer true that the state of the client could be inferred simply from the last server response, since script embedded in this response could have modified the state, perhaps based on user input. This new situation seriously impaired the ability of existing tools to truly crawl these applications. But it was the introduction of another possibility for these scripts that was a real challenge for crawlers: the ability for scripts to asynchronously exchange messages with the server (still using GET and POST commands) without having to reload the page entirely. This technique, most commonly used with Ajax, means that these Web Applications can really be seen as two programs running concurrently in the server and the browser, communicating by asynchronous message exchanges. We call these types of Web Applications "Rich Internet Applications" (RIAs). To the best of our knowledge, there are no tools today to efficiently and systematically crawl RIAs. Some academic tools have been developed [11,17] but the results are not entirely satisfactory yet. Some large commercial software do require crawling abilities, such as vulnerabilities scanners (e.g. see [3] for a recent survey of security scanners), but again their ability to really crawl RIAs is very limited in practice.

It is important to note that the difficulties introduced by RIAs for crawling have a much deeper impact than preventing security testing tools to perform well. First, crawling is the basis for indexing, which is necessary for searching. Our inability to crawl RIAs means that these RIAs are not properly indexed by search engines, arguably one of the most important features of the Web.[1] Second, because RIAs are often more user-friendly than simple sites and provide an enhanced experience for end-users, a growing number of web authoring tools automatically add RIA-type code (usually Ajax) into the produced site. So the problem extends much beyond advanced Web Applications. Even the most classical Web site, built without any programming ability by some content editor, might end up being non-crawlable and thus non-searchable. This is clearly an important problem that must be addressed. This paper is a first step in this direction. For simplicity, we only consider Ajax based RIAs in the following, but the same concepts can be applied to other technologies.

The paper is organized as follows: in Section 2, we provide a general overview of our crawling strategy. In Section 3, we give a high-level description of the set of algorithms we have developed for this purpose. In Section 4, we show how all of these algorithms are tied together to lead to our crawling strategy. In Section 5, we provide experimental results obtained with our prototype, compared with existing tools and other, simpler crawling strategies. Related works is reviewed in Section 6, and we conclude in Section 7 with some future research directions.

---

[1] To observe the current RIA crawling capabilities of the search engines, we created various experimental RIAs, each with 64 states reachable via AJAX calls. To date, Google and Yahoo visited our applications: neither was able to explore beyond the initial state!

2 Overview of Strategy

The purpose of crawling a RIA is to infer automatically an accurate model of the application. We conceptualize the model of the application as a Finite State Machine. The states of the state machine are the distinct DOMs that can be reached on the client side and a transition is an event execution (e.g. any JavaScript event such as onClick, onMouseOver, onSubmit etc.) on the current DOM. To produce an accurate and useful model, the crawling strategy must satisfy certain requirements. First, the produced model must be complete. That is, the model must represent all reachable states and all transitions. In addition to the event being executed, the next state of a transition may depend on other data, such as user inputs and variable values. Therefore, building a complete model requires capturing all the states and all such events along with the relevant data. Second, the model must be built in a deterministic way. That is, crawling the application twice under the same conditions (including server-side data) should always produce the same model. Third, the model should be produced efficiently. By this, we mean that the model construction strategy should produce as much of the model as possible within the shortest amount of time. Crawling an application may take a very long time, and even can be infinite theoretically; thus it may not be practical or even feasible to wait until the crawling ends. In such a case, if the crawl is stopped before it is complete, the partial model produced should contain as much information as possible.

Satisfying the completeness requirement is not too difficult on its own. There are several simple strategies that will achieve this. One must simply be careful not to overlook any of the intermediate states [4], even though most state-of-the-art crawling strategies fail to meet this requirement, usually on purpose since it is considered to take too much time. The determinism requirement is also not very difficult to achieve, although many RIAs may not easily provide the necessary uniform responses to the same requests. The efficiency requirement is obviously the most challenging one. Even the most simple-minded strategies, such as enumerating and executing one by one all possible orderings of events enabled at the state being explored, may satisfy the first two requirements, but not the efficiency requirement. Considering a single state with n enabled events, there are $n!$ possible different orders for executing these events, so blindly executing events in every possible order is not efficient. However, note that this simple strategy makes no hypothesis about the application beforehand, so there is actually not much room for generating a more efficient strategy as such.

To generate an efficient strategy, we use the following methodology which we call "model-based crawling":

- First, general hypotheses regarding the behavior of the application are elaborated. The idea is to assume that, in general, the application will usually behave in a certain way, described by the hypotheses. With these hypotheses, one can anticipate what the final model (or part of it) will be, if they are indeed valid. The crawling activity is thus morphed from a discovery activity ("what is the model?") into a confirmation activity ("is the anticipated model correct?"). Note that the hypotheses do not need to be actually valid.
- Once reasonable hypotheses have been specified, an efficient crawling strategy can be created based on these hypotheses. Without any assumptions about the behavior of the application, it is impossible to have a strategy that is expected to be efficient.

But if one anticipates the behavior, then it becomes possible to draw an efficient strategy to confirm the hypothesis. In our case, we provide a crawling strategy that is in fact optimal for our hypotheses.

- Finally, a strategy must also be established to reconcile the predicted model with the reality of the application that is crawled. Whenever the application's behavior doesn't conform to the hypotheses, some corrective actions must be taken to adapt the model and to update the crawling strategy. In our case, we always assume that the part of the application that has not yet been crawled will obey the initial set of hypotheses, and the crawling strategy is adapted accordingly.

The event-based crawling strategy exposed in this paper follows this methodology. To form an anticipated model, we make following two hypotheses only:

- **H1:** The events that are enabled at a state are pair-wise independent. That is at a state with a set $\{e_1,e_2,\ldots, e_n\}$ of n enabled events, executing a given subset of these events leads to the same state regardless of the order of execution.
- **H2:** When an event e_i is executed at state s, the set of events that are enabled at the reached state is the same as the events enabled at s except for e_i.

We believe that these hypotheses are usually reasonable for most of the web applications in the sense that executing different subsets of events is more likely to lead to new states, whereas executing a given subset of events in different orders is more likely to lead to the same state. More precisely, it is obviously not always the case that concurrently enabled events are independent of each other, and that executing an event will not enable new ones or disable existing ones. Still, it is reasonable to assume that, in general, the events that are concurrently enabled are "roughly" independents, as a whole. In fact, if a client state tends to have a large number of events, then it seems reasonable to expect that many of these events are independent of each other, and so even if the hypotheses are often violated for chosen pairs of events, they are usually verified overall. And of course, when this is not the case, the expected model and the corresponding strategy is updated without throwing away the whole strategy.

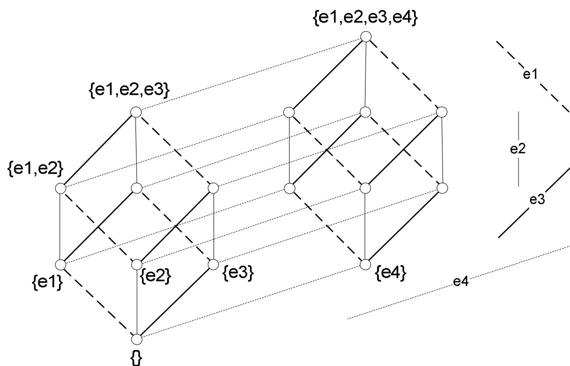

Fig. 1. Hypercube of dimension 4

With these hypotheses, the expected model for a state with n enabled events is a hypercube of dimension n. There are 2^n possible subsets of n events and a hypercube is a partial order of all possible subsets ordered by inclusion. Figure 1 shows an example hypercube of dimension $n = 4$. In a hypercube of dimension n, each of the 2^n states is characterized by a subset of n events executed to reach it. Each edge corresponds to a transition that is triggered by execution of one of the events. There are $n!$ different paths from the bottom state to the top state. Each one of them represents an order of execution containing n events.

Based on the hypercube hypothesis, a strategy should be generated. To satisfy our efficiency requirement, the strategy must aim at visiting each state in the expected model as soon as possible. Once each state has been visited, then the strategy completes the crawl by aiming at efficiently traversing every transition that has not yet been traversed. In the following section, we introduce an algorithm that generates an optimal strategy which completes crawling in (n choose (n/2))*⌈n/2⌉ paths instead of going over $n!$ paths. What is more only (n choose (n/2)) of these paths are enough to visit all the states in the hypercube. Both numbers are optimal.

Another important factor for the efficiency of the crawling algorithm is the choice of the state equivalence relation. Equivalence relation is used to determine whether a state should be regarded as being the same as another, already known. This avoids exploring the same state multiple times. Our crawling algorithm assumes that some appropriate equivalence relation is provided. Different equivalence relations can be employed depending on the purpose of model. For example, for security scanning, an equivalence relation that ignores text content of pages may be appropriate whereas for indexing the text must be taken into account. An in-depth discussion of this topic is beyond the scope of this paper, but it is important to note that the equivalence function we are working with is assumed to be a real equivalence relation (from the mathematical viewpoint) and that this function should at least be coherent with state transitions, that is, two equivalent states must at least have the same set of embedded URLs (an exception can be made if some specific parts of the page are ignored during the crawl, e.g. a part containing ads for example. URLs and events embedded into the ignored parts can also be ignored) and the same set of events. One obvious such equivalence relation is simple equality. Throughout this paper $\approx$ is used to denote the equivalence relation that is provided to the crawling algorithm.

3 Minimal Transition Coverage (MTC) of a Hypercube

In this section, we present an algorithm to find a minimal set of paths that traverse each transition in a given hypercube. We call such a set a Minimal Transition Coverage (MTC) of the hypercube. Since we also want to be able to visit each state of the hypercube as early as possible, we need more than just any MTC; we need one that will also reach all the states as fast as possible. For this reason, our MTC algorithm can be constrained by a set of disjoint chains of transitions given as input. Each chain of the constraint set will be included in one of the produced MTC chains. When the produced MTC is constrained with a minimal set of chains that visit each state in the hypercube, the resulting MTC is exactly what we are aiming for: a minimum number of paths containing every possible transition of the hypercube and

including as a subset a minimum number of paths visiting every state. We then simply have to give priority to the paths visiting the states to obtain our strategy.

In [5], it was stated that a Minimal Chain Decomposition (MCD) of a hypercube, which is the minimal set of paths visiting every state of the hypercube, could be used as part of a crawling strategy. Here, we use an MCD as a constraint for MTC to generate our strategy. First, we give a short review of the MCD for completeness.

3.1 Minimal Chain Decomposition

Since a hypercube represents a partially ordered set, a path in the hypercube consists of pair-wise comparable elements. A path is also called as a chain of the order. Finding a set of chains covering every element of the order is known as chain decomposition of the order. A minimal chain decomposition of the order is a chain decomposition containing the minimum number of chains (see [1] for an overview of the concepts). In 1950, Dilworth proved that the cardinality of any minimal chain decomposition is equal to the maximum number of pairwise incomparable elements of the order, also known as the width of the order [9]. For a hypercube of size n, the width is (n choose $n/2$). An algorithm that can be used for decomposing a hypercube is given in [8]. Below we present the algorithm as it is exposed in [12] (adapted to our hypercube definition).

Definition (adapted from [12]): The Canonical Symmetric Chain Decomposition (CSCD) of a hypercube of dimension n is given by the following recursive definition:

1. The CSCD of a hypercube of size 0 contains the single chain (Ø).
2. For $n \geq 1$, the CSCD of a hypercube of dimension n contains precisely the following chains:

- For every chain $A_0 < \ldots < A_k$ in the CSCD of a hypercube of dimension $n - 1$ with $k > 0$, the CSCD of a hypercube of dimension n contains the chains:
$$A_0 < \ldots < A_k < A_k \cup \{n\} \text{ and}$$
$$A_0 \cup \{n\} < \ldots < A_{k-1} \cup \{n\}.$$
- For every chain A_0 of length 1 in the CSCD of a hypercube of dimension n - 1, the CSCD of a hypercube of dimension n contains the chain:
$$A_0 < A_0 \cup \{n\}$$

3.2 MTC Algorithm

In the rest of this paper, we represents a chain C as alternation of states and events in the form $C = s_1 - e_1 - s_2 - e_2 - s_3 - \ldots - e_m - s_{m+1}$. Each chain starts and ends with a state. The first state in C is noted by first(C) and the last state in C is noted by last(C). Each event e_i in C corresponds to a transition ($s_i - e_i - s_{i+1}$) from s_i to s_{i+1}. The length of the chain C is defined to be the number of states in C and denoted as |C|. $\text{pref}_C(s_i)$ denotes the chain that is the prefix of C such that $\text{last}(\text{pref}_C(s_i))$ is s_i. $\text{suff}_C(s_i)$ denotes the chain that is the suffix of C such that $\text{first}(\text{suff}_C(s_i))$ is s_i. Two chains C_1 and C_2 can be concatenated if $\text{last}(C_1) = \text{first}(C_2)$. The resulting chain is represented as $C_1 + C_2$.

For a given hypercube of dimension n, we present an algorithm to generate an MTC in three stages. The algorithm allows MTC to be optionally constrained by a set

(C_C) of disjoint chains of the transitions. We call level 0 the bottom of hypercube, level *n* the top of the hypercube and level $\lfloor n/2 \rfloor$ middle of the hypercube.

Upper Chains Stage: In this stage, a set of chains (C_U) covering all the transitions above the middle of the hypercube is calculated. For each transition leaving a state at the middle level, a chain is constructed by extending the transition toward the top of the hypercube. Let U be the chain that is currently being constructed and s be last(U). Initially, U is a chain containing only the middle level state. Then we pick an unvisited transition (t = s - e - s') leaving s and attempt to extend U with t. If t is not part of any constraint chain then we can extend U with t. Otherwise t is included in a constraint chain $C \in C_C$. There can be two cases,

- If |U| = 1 or s is first(C) → extend U with $suff_C(s)$.
- Otherwise we cannot use t to extend U. So, we mark t as not usable and we attempt to extend U by picking another unvisited transition.

The extension of U continues until no transition can be found to extend it further.

Lower Chains Stage: In this stage, a set of chains (C_D) covering all the transitions below the middle level of the hypercube is calculated. This stage is simply the symmetric of the upper chains stage.

Chain Combination Stage: In the final stage, the chains in C_U and C_D are combined into larger chains. Note that, when the hypercube has an odd dimension, the number of transitions entering a middle-level state is one less than the number of transitions leaving it. In that case, for some upper chains, we cannot find a lower chain to combine with. For this reason, in this stage we iterate over the lower chains and try to combine each with an upper chain. After all lower chains combined, any uncombined upper chains are simply added to the resulting set. Again the only consideration while combining a lower chain with an upper chain is to keep every constraint chain as it is. That is, if is a constrain chain $C \in C_C$ has a prefix in the lower half and a suffix in the upper half, then the lower chain D containing the prefix of C can only be combined with the upper chain U containing the suffix of C and vice versa.

Note that, the number of chains in an MTC of a hypercube of size *n* equals (*n* choose (*n*/2))*$\lceil n/2 \rceil$ which is the number of states in the middle level (or the width of the hypercube) multiplied by the number of transitions leaving each middle level state.

4 Overall Strategy for Crawling RIAs

There are two methods for reaching new client states while crawling a RIA. One is through making synchronous HTTP requests to the server via URLs embedded in the DOM. The other is executing events in the DOM. An overall strategy for crawling RIAs should be a mix of these two and possibly running them alternatingly. For lack of space, we do not ellaborate here on the traditional, URL crawling part of the strategy, nor on the way we alternate between URL crawling and event crawling (see [4] for some details), and focus on event-based crawling only.

During URL-based crawling, whenever a page with a non-empty set of events is discovered, it is forwarded to the event based crawler for exploration. If such a state (called a base state) has *n* events, then the expected model we use to generate the crawling strategy is a hypercube of dimension *n* based on that state. To explore the states reachable from a base state s, a strategy Chains(s) that consists of a set of chains is generated. Let MTC(s) denote a set of MTC chains constrained by an MCD of the hypercube based on s. To generate the "initial" strategy, we use MTC(s) as a basis. Note that the chains in MTC(s) may not be executed in their original form because a chain in MTC(s) does not necessarily start at the base state s. To be able to begin executing events in a chain C∈MTC(s), we have to reach the state first(C). For this reason each C must be prepended with a (possibly empty) transfer chain, denoted T(first(C)), from the base state s to first(C). Any chain can be used as a transfer chain for C as long as it starts from the base state s and reaches first(C). Thus the initial event execution strategy for the hypercube based on state s will simply be the set Chains(s) = {C' = T(first(C))+C | C∈MTC(s)}. After, we begin exploration of the base state s, by executing the chains in Chains(s). Since we aim to reach every state first, we execute the chains that contain MCD chains first.

Executing a chain simply means executing the events of the chain one after the other. After an event is executed, if the state reached is a new state then this state is scanned and the newly found URLs are added to the traditional crawler. Then, we need to check if the state reached has the expected characteristics. If it is not, the current strategy and expected model must be revised.

The expected characteristics of a reached state are determined by the model hypotheses (in our case H1 and H2). H1 has two possible violations (non-mutually exclusive)

- Unexpected Split: In this case, we are expecting to reach a state that has already been visited but the actual state reached is a new state.
- Unexpected Merge: In this case, we unexpectedly reach a known state.

H2 can also be violated in two ways (non- mutually exclusive)

- Appearing Events: There are some enabled events that were not expected to be enabled at the reached state.
- Disappearing Events: Some events that were expected to be enabled at the reached state are not enabled.

When a violation occurs, the revision method creates a brand new strategy (still based on the original hypotheses) for the state causing violations and updates any chain in the existing strategy that is affected by the violations. As a result, the strategy will again be valid for the actual model of the application discovered so far. In the following, we give an outline (see Figure 2) and a high-level overview of the revision method, details can be found in [4].

Consider a hypercube based on state s_1 and the strategy Chains(s_1) currently used for s_1. Let the current chain be C = s_1 - e_1 - s_2 - … - s_i - e_i - s_{i+1} - … - s_n in Chains(s_1) and let s_i be the current state. According to C we execute the event e_i at s_i. Let s' be the actual state reached after this execution.

First we have to make sure that s' is a state that is in accordance with the expected model. What we expect s' to be depends on whether s_{i+1} is a known state or not. s_{i+1} is a known state if we executed a chain C'≠C in Chains(s_1) such that $pref_{C'}(s_{i+1})$ executes exactly the events $\{e_1, e_2, \ldots, e_i\}$ in any order without any violations. If this is the case then we expect s' to be equivalent to s_{i+1}. Otherwise we expect s' to be a new state without any appearing/disappearing events. If the expectations are met, from now on we regard s_{i+1} in the strategy as s'.

If s' is as we expected, the current strategy is still valid but it can be revised by removing some redundant chains. This is possible when C is our first attempt to visit s_{i+1} and it turns out that we had already been in s'. This is either because s' was reached through a base state different than s_1 or s' was reached through s_1 but the first time s' was reached there was an appearing/disappearing event violation. In either case, since s' has already been discovered, there must be an existing strategy to explore it. Hence all the chains having the prefix $pref_C(s_{i+1})$ are redundant and can be removed from Chains(s_1).

Procedure reviseStrategy
Let C = s_1 - e_1 - s_2- … - s_i – e_i – s_{i+1} - … - s_n be the current chain
    e_i be the event that has just been executed in C
    s' be the state reached by executing event e_i at s_i

If s' meets the expected model characteristics then
- If the current model contains s such that s' ≈ s then
- - remove from Chains(s_1) all chains containing $pref_C(s_{i+1})$;
- stop procedure reviseStrategy

Else // s' does not meet the expected model characteristics
- For each C' in Chains(s1) such that $pref_{C'}(s_{i+1}) = pref_C(s_{i+1})$
- - Let s_k be the first state of C' after s_i that belongs to another chain C'' of Chains(s1) that does not include s_i-e_i-s_{i+1}, if any
- - - add the chain $pref_{C''}(s_k)$+$suff_{C'}(s_k)$ to Chains(s_1);
- - - remove C'' from Chains(s_1) if s_k is its last state
- - remove C' from Chains(s_1)
- If this is the first time s' is visited, generate the crawling strategy for s' based on the hypercube hypothesis, with T(s') = $pref_C(s_i)$+(s_i-e_i-s')

Fig. 2. Revising the Crawling Strategy after executing an event

In case s' is not as expected, revision procedure checks for an alternative way to complete the remaining portion of C as much as possible. In order to do that, we first look for an alternative path among the remaining chains in Chains(s_1) to reach the minimum indexed state s_k ($i < k < n$) in $suff_C(s_1)$ without traversing the problematic transition (s_i - e_i - s_{i+1}). If a chain C'' has such a prefix then we can use that prefix to reach s_k and execute $suff_C(s_k)$ after that. That is, we add a new chain $pref_{C''}(s_k)$+$suff_C(s_k)$ to the strategy. Note that if s_k is the last state in C'' then to eliminate redundancy we can remove C'' from the strategy as it is covered completely

in the newly added chain. Also we should remove C from the strategy as it is now handled by newly added chain. This revision should be applied not only to C but to any chain C' having the prefix $pref_C(s_{i+1})$, since C' will face the same violation.

After all affected chains are updated, we check if we need to generate a strategy for s'. If s' is a state that has been discovered before, then we do not have to do anything as we must have already generated a strategy for s' at the time it was first discovered. Otherwise, a strategy (Chains(s')) should be generated for a new hypercube based on s'. Note that, currently the only known way to reach s' is using chain $pref_C(s_i)+(s_i$ - e_i - s'). For this reason, newly generated hypercube is actually an extension to the existing one and belongs to the same projection based on the state s_1. Hence each chain in Chains(s') is added into Chains(s_1) using $pref_C(s_i)+(s_i$ - e_i - s') as a transfer chain to reach s'.

5 Experimental Results

To assess the efficiency of our crawling algorithm, we have implemented a prototype tool (see [4] for detatils).We present results showing the performance of our prototype compared with a prototype of major commercial software for testing web applications, an open source crawling tool for Ajax applications (Crawljax) [17] as well as pure breadth-first and depth-first crawling.

To evalute a method, we first check if it achieves a complete crawl. For the ones that can crawl all the test applications completely, we record the total number of event executions and the total number of reloads required before discovering all the states and all the transitions separately. The collected data gives us the total number of event executions and reloads required to complete the crawl and an idea about how quickly a method is able to discover new states and transitions.

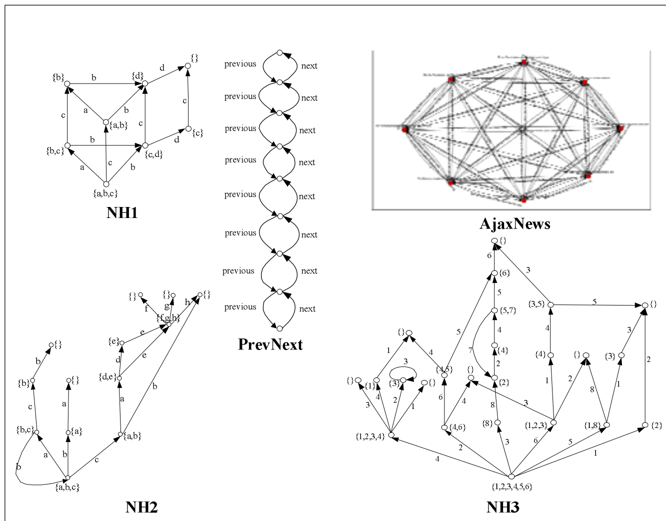

Fig. 3. Models of some test applications

The results were obtained using six Ajax test applications. (See Figure 1 and 3 for the models). Five of them were developed by us and one is a publicly available test application. We designed the applications to reflect different characteristics of the web applications as well as some extreme cases like a perfect hypercube or a complete graph. The first application, called H4D, has the model of a 4 dimensional hypercube (16 states, 32 transitions). Since hypercube is the expected model, H4D is a best case for our strategy. The second application NH1(8 states, 12 transitions) slightly deviates from the hypercube structure. The third and the fourth applications, NH2 (13 states, 15 transitions) and NH3 (24 states, 32 transitions), also deviate from the hypercube structure. The fifth application, PrevNext (9 states, 16 transitions) represents a model that is encountered in many Ajax applications. States of PrevNext are linked by previous and next events. The last application AjaxNews[2] (8 states, 80 transitions) is a publicly available test site [11]. It is in the form of a news reader with 8 different articles. It allows users to browse articles by previous and next buttons as well as to go directly to an article via clicking titles listed on a menu.

In an effort to minimize any influence that may be caused by considering events in a specific order, the events at each state are randomly ordered for each crawl. Also, each application is crawled 10 times with each method and the average of these 10 crawls are used for comparison.

Only depth-first, breadth-first and our algorithm successfully discovered all the states and all the transitions for all the applications. The commercial product protoype could not achieve a complete crawl for any of the applications. That is because it did not apply a specific strategy for crawling RIAs but blindly executed the events on a page once (in the sequence that they appear) without handling DOM changes which may add or remove events to the DOM. This is an example of a very naïve approach. Crawljax could crawl some completely but not most of them. (This is because Crawljax decides which events to execute at the current state using the difference between the current state and the previous state. That is, if a state transition leaves an HTML element with an attached event unchanged, then that event is not executed at the current state although it is still likely that this event could have led to a different state when executed in the current state). Table 1 shows the number of states discovered by each tool.

Table 1. Number of states discovered by Crawljax, Commercial software and our tool

Application	Total States	States Discovered by		
		Commercial	Crawljax	Our Tool
H4D	16	5	11	16
NH1	8	4	8	8
NH2	13	3	6	13
NH3	24	3	14	24
PrevNext	9	3	3	9

For each test application and for each method, we present the number of transitions and reloads required to visit all the states (and traverse all the transitions) of the application. Figures 5a and 5b shows the total transitions and total reloads required

[2] http://www.giannifrey.com/ajax/news.html

for discovering states, respectively. Figure 6a and 6b shows the total transitions and total reloads required for traversing all transitions. For compactness we use boxplots that should be interpreted as follows. The top of vertical lines show the maximum number required to discover all the states (or all transitions).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 3 quarters of all the states (or transitions) in the application, respectively. We will use the position of this box and the middle line to assess whether a method is able to discover new states (or transitions) faster than others.

As Figure 5a shows for H4D, NH1, NH2 and NH3 our method discovered all the states using the least number of transitions. For PrevNext, depth-first required slightly fewer transitions than ours and for AjaxNews breadth-first required significantly fewer transitions. Since the underlying model of AjaxNews is a complete graph, it is an optimal case for breadth-first. Hence it was not surprising to see breadth-first discovering all states using minimal amount of transitions. Looking at the figure to assess the rate of state discovery measured by the number of transitions, we conclude that our method discovers states at a faster rate, except for AjaxNews. Even for AjaxNews, our algorithm discovered more than half of all the states at a rate comparable to breadth-first.

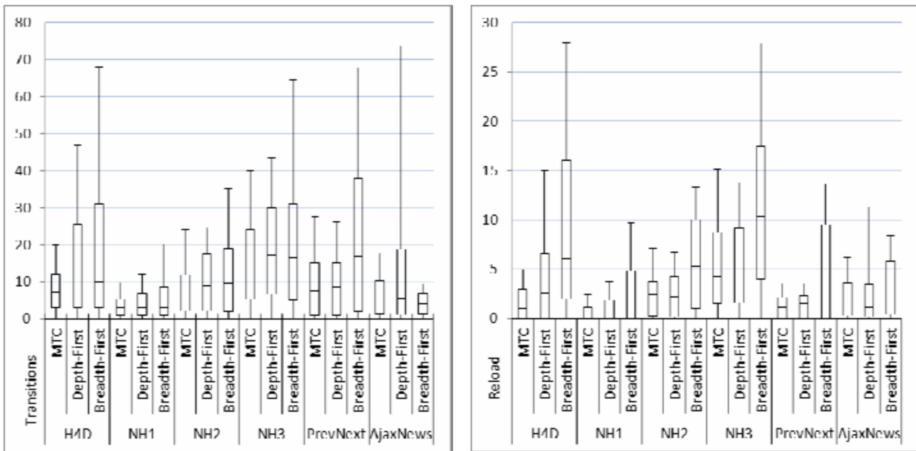

Fig. 5. a. Transitions for state discovery b. Reloads for state discovery

Figure 5b contains the boxplots of the number of reloads required before discovering all states. It is clear that, our method uses, if not fewer, a comparable number of reloads to depth-first. For H4D and AjaxNews, our method uses significantly less reloads than the others. Looking at the figure to assess the rate of state discovery measured by the number of reloads, conclusion is that our method is able to discover states at higher rates than other mehtods except AjaxNews for which the rate is slightly lower than depth-first.

Figure 6a contains the boxplots showing for each application and for each method the number transitions executed before having traversed all transitions. Comparing the number of transitions used to be able to traverse all the transitions (note that this is the number of transitions executed to complete the crawl), we see that for H4D and NH1 our method is the best, for PrevNext depth-first and our method are even and for NH2 and NH3 depth-first uses less transitions. For AjaxNews breadth-first uses the fewest transitions as expected. Comparing the rate of executing unexecuted transitions, for all the applications except PrevNext, our method uses fewer transitions to execute the majority of transitions. For PrevNext, depth-first has a slightly better rate but the rate is better for ours for more than half of the transitions before depth-first slightly takes over. Note also that, for AjaxNews our method has a significanty higher rate.

Lastly, Figure 6b shows the number of reloads required before executing all transitions. Looking at the total number of reloads required to complete the crawl, for H4D, NH1 and AjaxNews our method was the best, for PrevNext depth-first and our method are even and for NH2 and NH3 depth-first used slightly fewer reloads. Comparing the rates, in order to execute the majority of the transitions, our method required fewer reloads for all the applications except PrevNext, for which depth-first and our algorithm have similar results.

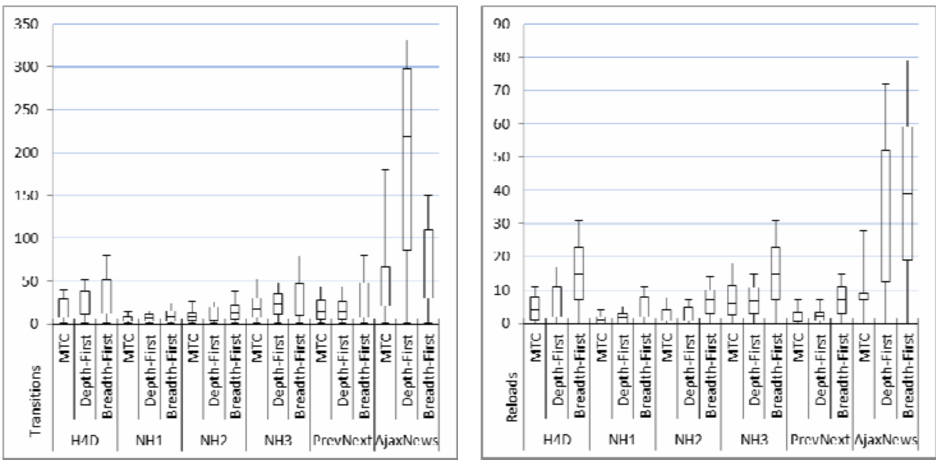

Fig. 6. a. Transitions for new transitions b. Reloads for new transitions

As an overall evaluation based on these results, we make following remarks. Our approach can extract a correct model of an application regardless of its model. In terms of the total number of transitions and reloads, our approach was the most consistent. That is, for all the cases it was either the best or very close to the best. It was never the worst. This is because, our approach tries to balance the good characteristics of the depth-first and breadth-first approaches. It tries, like depth-first, to maximize the chain length and thus reduce the number of reloads. Also it tries, like breadth-first, to cover the breadth of the application quickly. As we aimed, our strategy has a better rate of discovering new states and transitions.

6 Related Work

We survey here only research focusing on crawling RIAs, for general crawling see [2,7,13]. Although limited, several papers have been published in the area of crawling RIAs, mostly focusing on Ajax applications. For example [10,11,15] focus on crawling for the purpose of indexing and search. In [16], the aim is to make RIAs accessible to search engines that are not Ajax-friendly. In [18] focus is on regression testing of AJAX applications whereas [6] is concerned with security testing.

The works that contain the modeling aspect seem to agree on representing models using finite state machines [14,15,17]. Usually states are defined using the DOM structure and the transitions are based on the events that lead to changes in the DOM [15,17]. In [14], however the states, as well as the events to consider, are abstracted by observing DOM changes in the given execution traces of the application.

As a crawling strategy, [15] uses breadth-first crawl. In order to reduce the number of Ajax requests, whenever a specific Ajax request is made for the first time, the response from the server is cached. Later, if there is a need to call the same function with the same parameters for execution of some event, the cashed response is used instead of making an actual request. This assumes calling the same function with the same parameters always produces the same response regardless of the current state. Therefore it provides no guarantee to produce a correct model. In [17], a depth-first-based method is described and implemented in Crawljax. As we have explained in Section 5, this method, too, is incapable of producing a complete model.

7 Conclusion and Future Work

In this paper, we have presented a general model-based strategy for crawling RIAs. Our solution aims at finding all the client states as soon as possible during the crawl but still eventually finds all the states. Experimental results show that our solution is correct and provides good results on a set of experimental sites: although our solution is not always the best one, it is the only one that works always and seems to always provide one of the best results in terms of efficiency. We believe that these are very encouraging results, but more must be done. We are currently working on the following enhancements:

First, we must extend our experimental results to include large number of real RIAs. We believe that the results will in fact be even more favorable when crawling sites such as Facebook or Google Calendar, because these sites are not going to have the extreme patterns of our test sites, that sometimes favor one crawling technique over the other.

Second, a possible criticism of our solution may be the computation of the strategy based on the hypercube hypothesis ahead of time. Because of the explosion of the size of a hypercube model, this approach may not be practical. Although we have already addressed this issue by generating chains one at a time, as required during the crawl, for simplicity we prefered to explain our solution by using pre-computed chains. We plan to explain the details of the on-the-fly chain generation technique in a separate contribution.

A third enhancement would be to look at other models, beyond the hypercube. Our model-based crawling methodology can be used with any models, and we are currently investigating which other models could be efficiently used.

Finally, a fourth direction would be to enhance the model with the notion of "important" states and events, that is, some states would have priority over others and some events would be more important to execute than others. Again, we believe that our model-based crawling strategy can be adjusted to deal with this situation.

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

Disclaimer. The views expressed in this article are the sole responsibility of the authors and do not necessarily reflect those of IBM.

Trademarks. IBM, Rational 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.

References

1. Anderson, I.: Combinatorics of Finite Sets. Oxford Univ. Press, London (1987)
2. Arasu, A., Cho, J., Garcia-Molina, A., Paepcke, A., Raghavan, S.: Searching the web. ACM Transactions on Internet Technology 1(1), 2–43 (2001)
3. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.C.: State of the Art: Automated Black-Box Web Application Vulnerability Testing. In: Proc. IEEE Symposium on Security and Privacy (2010)
4. Benjamin, K.: A Strategy for Efficient Crawling of Rich Internet Applications. Master's Thesis. SITE-University of Ottawa (2010), `http://ssrg.site.uottawa.ca/docs/Benjamin-Thesis.pdf`
5. Benjamin, K., Bochmann, G.v., Jourdan, G.V., Onut, I.V.: Some Modeling Challenges when Testing Rich Internet Applications for Security. In: First International Workshop on Modeling and Detection of Vulnerabilities, Paris, France (2010)
6. Bezemer, B., Mesbah, A., Deursen, A.v: Automated Security Testing of Web Widget Interactions. In: Foundations of Software Engineering Symposium (FSE), pp. 81–90. ACM, New York (2009)
7. Brin, S., Page, L.: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Computer Networks and ISDN Systems 30(1-7), 107–117 (1998)
8. Bruijn, N.d.G., Tengbergen, C., Kruyswijk, D.: On the set of divisors of a number. Nieuw Arch. Wisk. 23, 191–194 (1951)
9. Dilworth, R.P.: A Decomposition Theorem for Partially Ordered Sets. Annals of Mathematics 51, 161–166 (1950)
10. Duda, C., Frey, G., Kossmann, D., Zhou, C.: AJAXSearch: Crawling, Indexing and Searching Web 2.0 Applications.VLDB (2008)
11. Frey, G.: Indexing Ajax Web Applications, Master's Thesis, ETH Zurich (2007)
12. Hsu, T., Logan, M., Shahriari, S., Towse, C.: Partitioning the Boolean Lattice into Chains of Large Minimum Size. Journal of Combinatorial Theory 97(1), 62–84 (2002)

13. Manku, G.S., Jain, A., Das Sarma, A.: Detecting near-duplicates for web crawling. In: Proceedings of the 16th International Conference on World Wide Web. WWW 2007, pp. 141–150. ACM, New York (2007)
14. Marchetto, A., Tonella, P., Ricca, F.: State-based testing of Ajax web applications. In: Proc. 1st IEEE Intl. Conf. on Software Testing Verification and Validation (ICST 2008). IEEE Computer Society, Los Alamitos (2008)
15. Matter, R.: Ajax Crawl: making Ajax applications searchable. Master's Thesis. ETH, Zurich (2008)
16. Mesbah, A., Deursen, A.v.: Exposing the Hidden Web Induced by AJAX.TUD-SERG Technical Report Series. TUD-SERG-2008-001 (2008)
17. Mesbah, A., Bozdag, E., Deursen, A.v: Crawling Ajax by Inferring User Interface State Changes. In: Proceedings of the 8th International Conference on Web Engineering, pp. 122–134. IEEE Computer Society, Los Alamitos (2008)
18. Roest, D., Mesbah, A., Deursen, A.v: Regression Testing Ajax Applications: Coping with Dynamism. In: Third International Conference on Software Testing, Verification and Validation, pp. 127–136 (2010)