Indexing Rich Internet Applications Using Components-Based Crawling

Ali Moosavi[1], Salman Hooshmand[1], Sara Baghbanzadeh[1],
Guy-Vincent Jourdan[1,2], Gregor v. Bochmann[1,2], and Iosif Viorel Onut[3,4]

[1] EECS - University of Ottawa
[2] Fellow of IBM Canada CAS Research, Canada
[3] Research and Development, IBM® Security AppScan® Enterprise
[4] IBM Canada Software Lab, Canada
{smousav2,shooshma,sbaghban}@uottawa.ca
{gvj,bochmann}@eecs.uottawa.ca
vioonut@ca.ibm.com

Abstract. Automatic crawling of Rich Internet Applications (RIAs) is a challenge because client-side code modifies the client dynamically, fetching server-side data asynchronously. Most existing solutions model RIAs as state machines with DOMs as states and JavaScript events execution as transitions. This approach fails when used with "real-life", complex RIAs, because the size of the produced model is much too large to be practical. In this paper, we propose a new method to crawl AJAX-based RIAs in an efficient manner by detecting "components", which are areas of the DOM that are independent from each other, and by crawling each component separately. This leads to a dramatic reduction of the required state space for the model, without loss of content coverage. Our method does not require prior knowledge of the RIA nor predefined definition of components. Instead, we infer the components by observing the behavior of the RIA during crawling. Our experimental results show that our method can index quickly and completely industrial RIAs that are simply out of reach for traditional methods.

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

1 Introduction

In the past decade, modern web technologies such as AJAX, Flash, Silverlight, etc. have given emergence to a new class of more responsive and interactive web applications commonly referred to as Rich Internet Applications (RIAs). RIAs make Web-applications more interactive and efficient by introducing client-side computation and updates, and asynchronous communications with the server [1]. Crawling RIAs is more challenging than crawling traditional web applications because some core characteristics of traditional web applications are violated by RIAs. Client states no longer correspond to unique URLs as modern web technologies enable the ability to change the client state and even populate it with

new data without changing the URL, up to the point that it is possible to have complete complex web applications with a single URL. Moreover, JavaScript events (from here on, called *"events"*) can take the place of hyperlinks; and unlike hyperlinks, the crawler cannot predict the outcome of an event before executing it. In that sense, the behavior and user interface of a RIA is more similar to an event-driven software like a desktop software GUI than to a traditional web application.

The problem of crawling AJAX-based RIAs has been a focus of research in the past few years. In such RIAs, executing events can change the Document Object Model (DOM), hence leading the RIA to a new client state, and can possibly leaned to message exchanges with the server, changing the server state as well. A common approach is to model a RIA as a finite state machine (FSM). In the FSM, DOMs are represented as states and event executions are represented as transitions. Events can lead from one DOM-state[1] to another.

One simplifying assumption that is usually made is that server states are in sync with client states. Therefore by covering all client states (i.e. DOM-states), the crawler has also covered all server states. Based on this assumption, by executing each event from each DOM-state once and building a complete FSM model from the RIA, the crawler can assume that the RIA has been entirely covered and modeled. In order to stay in sync with server, however, the crawler cannot jump to arbitrary DOM-states at will (e.g. by saving DOM-state in advance and restoring it when desired); instead, it must follow a sequence of events. If the desired DOM-state is not reachable from the current DOM-state using a chain of transitions (called a "transfer sequence"), the crawler needs to issue a "reset" (reloading the URL of the initial page) to go to the initial DOM-state and follow a valid transfer sequence from there. Resets are usually modelled as special transitions from all DOM-states to the initial DOM-state. In the beginning, the only known DOM-state is the initial DOM-state and all its events are unexecuted yet. By executing an unexecuted event, the crawler discovers its destination, which might be a known DOM-state or a new one. The event execution can then be modelled as a transition between its source and destination DOM-states. A state machine can be represented as a directed graph. The problem of crawling a RIA is therefore that of exploring an unknown graph. At any given time, the crawler needs to execute an unexecuted event, or use the known portion of the graph to traverse to another DOM-state to execute one, until all events in the graph have been executed, at which point the graph is fully uncovered and crawling is done. Based on this model, different exploration strategies (such as depth-first search, Greedy and Model-Based strategies, see Section 2) have been suggested. Comparing different exploration strategies can be done by comparing the number of events and resets executed during the crawl. We call this the *"exploration cost"*.

[1] Related works in the literature commonly refer to DOM-states simply as "states". In this work we differentiate between "DOM-states" and what we will call "component-states".

One major challenge in this field is a state space explosion: the model being built grows exponentially in the size of the RIAs being crawled. This state space explosion not only leads to production of a large model that is difficult to analyze, but also makes the crawlers unable to finish in a reasonable time. Because of this excessive running times, all DOM-based methods that have been published so far are essentially unsuitable for real-life scenarios. These methods cannot be used in the industrial world. Current studies use different notions of DOM equivalence to map several DOMs to one state. These approaches usually involve applying reduction and normalization functions on the DOM. While these approaches have been used and tested successfully on experimental RIAs, they fail to provide satisfactory equivalency criteria when faced with real-world large-scale RIAs.

Most of the time, this state space explosion is caused by having the same data being displayed in different combinations, leading to large sets of new DOMs and large state space for a small set of functionalities. In a typical RIA, it is common to encounter a new DOM-state which is simply a different combination of already-known data (Figure 1). We call this situation *new DOM-state without new data*. Such DOM-states should be ideally regarded as already known. Today's complex RIA interfaces consist of many interactive parts that are independent from one another, and the Cartesian product of different content that each part can show easily leads to an exponential blow-up of the number of DOM-states. In the following, we call these independent parts *components*, and each of their values *component-state*. A fairly intuitive example is widget-based RIAs, in which various combination of contents that each widget can show creates a very large number of different DOM-states. Not all these DOM-states are of interest to the crawler. A content indexing crawler, for instance, needs to retrieve the content once and finish in a timely fashion. These "rehashed" DOM-states only prolong crawling while providing no new data. Figure 1 provides an example. This issue is not just limited to widgets, but is present in any independent part in RIAs down to every single popup or list item. Typical everyday websites such as Facebook, Gmail and Yahoo, and any typical RIA mail client, enterprise portal or CMS contain dozens if not hundreds of independent parts. Different combinations of these independent parts lead crawlers through a seemingly endless string of new DOM-states with no new data. A human user, on the other hand, is not confused by this issue since she views these components as separate entities, and in fact would be surprised if the behavior of one of these parts turns out to be dependent on another.

We observe that one major drawback inherent to all these methods is that they model client states of RIA at the DOM level. We propose a novel method to crawl RIAs efficiently by modeling in terms of states of individual sub-trees of the DOM that are deemed independent, which we call *components*. Our method detects independent components of RIA automatically, using the result of diffs between DOMs. By modeling at the component level rather than at the DOM level, the crawler is able to crawl complex RIAs completely (and in fact quickly) while covering all the content. The resulting end-model is smaller and therefore easier for humans to understand and for machines to analyze, while providing

Fig. 1. Example of a new DOM-state with no new data. The DOM in **(c)** is only a combination of data already present in **(b)** and **(a)**, but will have a new DOM-state in the existing methods

more information about the RIAs being modeled. As we will show in our experimental results, the method presented here is suitable to crawl and index real-life, complex RIAs, without loss of content in our experiments, where previously published methods failed to do the same thing even on much simplified version of the same RIAs.

The remainder of this paper is organized as follows. In Section 2, we provide a review of related work. In Section 3 we present the general overview of our solution. We first describe the model that the crawler builds, and we then describe how the crawler builds this model and makes use of it during the crawl. Experimental results and comparisons are presented in Section 4, and we conclude in Section 5.

2 Related Works

Crawling RIAs using a state transition model has been extensively studied. Duda et al. use a breadth-first search approach to explore RIA, assuming the ability to cache and restore client states at will [2, 3]. In [2], they point to the state space explosion problem caused by independent parts as an unresolved challenge. Amalfitano et al. use manual user-sessions to build a state machine [4]. In a follow-up work, they automate their tool by using depth-first exploration [5]. Peng et al. propose using a greedy algorithm as exploration strategy that outperforms depth-first and breadth-first search exploration significantly [6]. We use here the same greedy approach as exploration strategy. A different approach, called "model-based crawling" [7], focuses on finding the clients states as soon as possible by assuming some particular behavior from the RIAs [8,9]. The model

used in [9] accumulates statical information about the result of previous event executions to infer what event to execute next. All the these method suffer from the problem of accumulating new DOM states that do no contain new data. In [10], an approach similar to model-based crawling is used, but for sites that have a known structure.

DynaRIA [11] provides a tool for tracing AJAX-Based application executions. It generates abstract views on the structure and run-time behavior of the application. The generated crawling model has been used for accessibility testing [12] or for generating test sequences [13]. It also also been used for modelling native android apps [14, 15] and native iOS apps [16].

All the above mentioned works use DOM-level state machines and use different DOM equivalence criteria to guide crawling: in [17], an edit distance between DOM-states is used. Methods based on DOM manipulations are used in [7–9, 18]. These various DOM equivalence criteria do help but ultimately fail to address completely the state space explosion problem. To alleviate this problem, in [17] it is proposed to explore only new events that appear on a DOM after an event execution.This limits the crawler's ability to reach complete coverage, and does not prevent exploring redundant data when different event execution paths lead to the same structure (e.g. a widget frame) but in different DOMs. *FeedEx* [19] extends [6] by selecting states and events to be explored based on probability to discover new states and increase coverage. They include four factors to prioritize the events : code coverage, navigational diversity, page structural diversity and test model size. Surveys of RIAs crawling can be found in [20, 21].

In the context of detecting independent parts, static widget detection methods such as [22] and [23], and detection of underlying source dataset [24] have been developed. In [23], the use of patterns for detecting widgets based on static JavaScript code analysis and interaction between widget parts is proposed. However, these methods designed only to detect widgets or source datasets, which are a small subset of independent entities in RIAs.

3 Component-Based Crawling

3.1 Overview of our Solution

Our solution is to model RIAs at a "finer" level, using subtrees of the DOM (called "components") instead of modeling in terms of DOMs. By building a state-machine at the component level, we get a better understanding of how the RIA behaves, which helps addressing the aforementioned state explosion problem [25]. The crawler can use this model along with its exploration strategy. Our prototype implementation uses the greedy algorithm presented in [26] as the exploration strategy. In this section we present a brief general overview of the concept of components, before providing more details in the following Sections.

In a typical real-life RIA such as the one depicted in Figure 2, a given "page" (DOM-state) contains a collection of independent entities. We call these entities "components". They are subtrees of the current DOM. Examples of components

include menu bars, draggable windows in Twitter, as well as each individual tweet, chat windows in Gmail, as well as the frame around each chat window, the notifications drop-down and mouse-over balloons in Facebook, etc. Users normally expect to be able to interact with each of these components independently, without paying attention to the state of the other components on the page. Classical crawling methods do not consider these components, and consequently generate every possible combination of these components states while building the model. Our aim is to detect these components to crawl each of them separately. The assumption of independency between components enables our method to "divide and conquer" the RIA to overcome state space explosion without loss of coverage. We expect this assumption to hold true in almost all real-life RIAs as it follows human user intuition. We did not encounter any counterexamples in our investigation of real-life RIAs. If, however, there are components on a particular RIA that affect each other, the crawler might lose coverage of some of the content of the RIA since it does not analyze the interactions between the components. In our experiments, this situation did not occur.

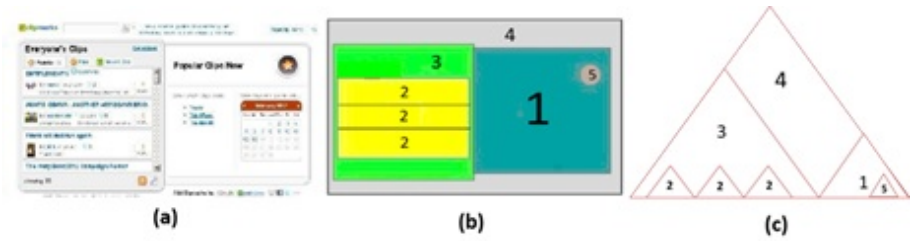

Fig. 2. (a) A webpage, **(b)** components on the page the way a human user sees them as entities of the page, and **(c)** the way the crawler sees them as subtrees of the DOM

The input of the crawler after each event execution is the DOM tree. Since components appear as subtrees in the DOM tree, we partition the DOM into multiple subtrees that are deemed independent of each other. Each of these subtrees correspond to a particular state of a component (a *component-state*). We model the RIAs as a combination of independent component states instead of assigning a DOM-state to the entire DOM. The idea of components and their associated component-states completely replace use of DOM-states in our method. Each component has a set of possible component-states, and a component-state of a particular component is only compared to other component-states of its own.

As explained, in our model, at any given time, the RIA is in a set of component-states, since it consists of different components each in its own component-state. It is worth mentioning that the DOM is partitioned into components in a collectively exhaustive and mutually exclusive manner, meaning that each XML-node

on the DOM tree belongs to one and only one component. Modeling RIAs at the component level as several benefits. The most obvious one is that it reduces the state space by avoiding modeling separately every combination of component states, including the many instances in which the combination is new but each component state has been seen before (as depicted previously in Figure 2). Moreover, this fine-grained view of RIA helps the crawler map the effect of event executions more precisely, resulting in a simpler model of the RIA with fewer states and transitions. As a result, the crawler will traverse the RIA more efficiently by taking fewer steps when aiming to revisit a particular structure (such as a text or event) in the RIA that is not present in the current DOM. The resulting model of the RIA will also be more easily understandable by humans because it has fewer states and transitions and the effect of each event execution on the DOM is defined more clearly.

To illustrate the potential gain of our methods, imagine that the current DOM is made of k independent components $C_1, C_2, \ldots, C_k$. Assume that each component C_i has $\bar{C_i}$ components states. Using the traditional, DOM-state based method, this will lead to $\prod_{i=1}^{k} \bar{C_i}$ DOM-states. If in addition the components can be moved freely on the page, this number will be repeated $k!$ times, leading to $k! \prod_{i=1}^{k} \bar{C_i}$ DOM-states. This already intractable number will increase even more of some components are repeated or if some components can be removed from the DOM. Using our method yields only $\sum_{i=1}^{k} \bar{C_i}$ component-states for the same RIAs, even when the components are repeated or removed from the page.

3.2 Model Description

In our model, we partition each DOM into independent components. Each component has its own component-state so the current DOM corresponds to a set of component-states in the state machine. Because JavaScript events are attached to XML nodes, each event resides in one of the component-states present in the DOM. We call it the *"owner component-state"* of the event[2].

3.2.1 Multistate Machine An event is represented as a transition that starts from its owner component-state. Since the execution of the event can affect multiple components, the corresponding transition can end in multiple component-states. Therefore, our model is a multi-state-machine. Figure 3 illustrates how an event execution is modeled in our method versus other methods. The destination component-states of a transition correspond to component-states that were not present in the DOM, and appeared as a result of the execution of the event.

Our model is a multistate-machine, defined as a tuple $M = (A, I, \Sigma, \partial)$ where A is the set of component-states, I is the set of initial component-states (those that are present in the DOM when the URL is loaded), Σ is the set of events, and $\partial : A \times \Sigma \rightarrow 2^A$ is a function that defines the set of valid transitions. Similarly to

[2]For events that are not attached to an XML-node on the DOM, such as timer events, a special global always-present component is defined as their owner component.

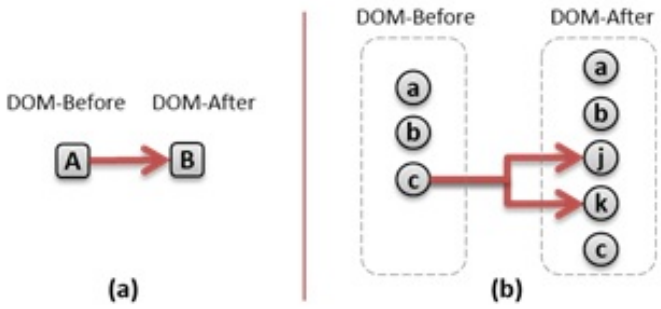

Fig. 3. An event execution modeled with **(a)** DOM-states, and **(b)** component-states. Rectangles in **(b)** represent DOM-states and are not used in the actual model

the classical state-transition model, ∂ is a partial function, since not all events are available on all component states. Unlike the state-transition model, we have a set of initial states, and executing an event can modify any number of component-states. Our multistate-machine is resilient to shuffling components around in a DOM, and does not store information about exact location of the component-states in a DOM.

3.2.2 Components Definition We have mentioned that components must be "independent" from one another. By this, we mean that the outcome of execution of an event only depends on the component-state of its owner component. In other words, the behavior of an events in a component is independent from the other components in the DOM and their individual component states. As an example, the border around a widget that has minimize/close buttons is independent of the widget itself, since it minimizes or closes regardless of the widget that it is displaying. Therefore, the widget border and the widget itself can be considered separate independent components. On the other hand, the next/previous buttons around a picture frame are dependent on that picture frame, since their outcome depends on the picture currently being shown. So the next/previous buttons should be put in the same component as the picture frame. Note that event executions outcome can affect any number of components and this does not violate the constraint of independency.

Two notions are important in our definition: first, we need to specify how we define a component, then how we capture the various component states that component might have.

Component are identified by an XPath, which specifies the root of the subtree that contains this component. In order to find a particular component in the DOM, one should start from the document root and follow the component's associated XPath. The element reached is the root of the component i.e. the component is the subtree under that element. Note that an XPath can potentially map to several nodes, therefore several instances of a component can be

present in a DOM at the same time. Since the XPath serves as an identifier for a component, we need the XPath to be consistent throughout the RIA, i.e. it should be able to point to the intended subtree across different DOMs of the RIA. However, some attributes commonly used in XPath are too volatile to be consistent throughout the RIA. Hence, we only use the "id" and "class" attributes for each node in the XPath, and omit other predicates such as the position predicate.

Here is how we build an XPath for an element e: we consider the path p from the root of the DOM to e, and for each HTML element in p, we include the tag name of the element, the *id* attribute if it has one, and the *class* attribute if it has one.

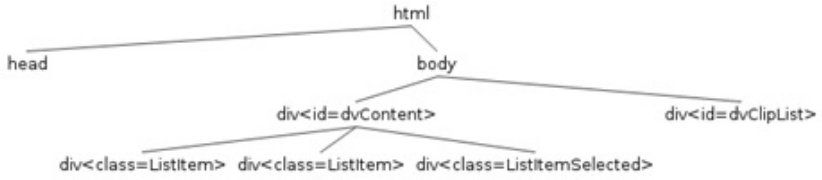

Fig. 4. Part of a shopping website's DOM

Figure 4 is an example of DOM for a shopping website. Individual list items in the product list are instances of a component "product list item". In this example, there are two instances of this component, which is identified by the XPath */html/body/div[@id="dvContent"]/div[@class="ListItem"]*. But the selected item in the list yields a different XPath since it is assigned a different class attribute, *[@class="ListItemSelected"]*.

Each component, identified by its XPath which points at the root of the component, can have a series of component states that are going to be uncovered throughout the crawl. These states are simply the content of the subtree found under the XPath. For efficiency reasons, we are not recording the entire subtree for each component state. Instead, we derive a unique identifier for the component state by hashing the content of the corresponding subtree[3]. In practice, the crawler keeps information on each component-state of each component in a data structure for use by the greedy algorithm. A simplified version of the data structure, called *stateDictionary*, is depicted in Table 1. In general, on any given DOM, some components are present in the DOM and some are not. Using the "Component Location" column, the crawler can find out which components are present on the DOM, then use the component's content to compute its ID. Using the "Component-State ID" column it can look up additional info on that

[3]In reality, we first prune nested sub-components as explained later, and we also perform some transformations on the subtree to detect equivalents sub-components. See [27] for more details.

component-state (event/transition destinations, unexecuted events, etc.), or discover that it is a new component-state.

#	Component Location (XPATH)	Component State-ID (Hash)	Info
1	/	@J#F@)J#403rn0f29r3m19	....
2	/html/body/div[@id="dvClipList"]	*&^$@̂J$$P@@$#$#_!$_*$_*	...
		GPDFJD}{PL"!{#R$$)%$*!_$#!!	...
3	/html/body/div[@id="dvContent"]	VMLCVCPQ!#$!_()̃__IKEF)_I)	...
		{:$%@)(@#*GRJPGFD{#@)(	...
		?"$#%*%@$)(!#HI!_D}{|||#R!#!	...

Table 1. The StateDictionary.

Since components are identified by their XPath, it is possible to have nested components, with the root of one component (identified by its XPath) failing inside the subtree of another component. If a component contains other components, these components should be removed from the containing component when defining that component's states. The pseudo-algorithm below explains how to list all the known component states that are present in the current DOM. Refer to [27] for more details.

Algorithm 1 Pseudo-code to find current component-states

```
procedure FindCurrentStates
    for all xpath in stateDictionary do
        ComponentInstances := go through the xpath and give the subtree
        for all Instance in ComponentInstances do
            for all sub-path under the current xpath do
                go through the sub-path and prune the subtrees
            end for
            stateID := ReadContentsAndComputeStateID(instance)
            Add the stateID to SetOfCurrentState
        end for
    end for
    return SetOfCurrentStates
end procedure
```

Note that the only location information for component states is an XPath to the root of the component state. Therefore, while our model is able to break a DOM into component-states, it is not possible to reconstruct an exact DOM using the multi-state-machine. While the resulting model of a RIA can be used to infer an execution trace to any given component state (thus any content of the RIA), it cannot be used to infer an execution trace to any given DOM.

3.3 Crawling Algorithm

As explained above, a crawling algorithm relies on an exploration strategy that tells it which events to execute next. Several exploration strategies can be used with our model. We explain our algorithm independently from the chosen strategy (which is "greedy" in our prototype).

Generally, using the "Component Location" list in the stateDictionary, the crawler can discover new component-states during the crawl and populate the "Component-State ID" lists. Our proposed component discovery algorithm populates the Component Location and Component-State ID lists incrementally during the crawl as it observes the behavior of the RIA. The algorithm is based on comparing the DOM tree snapshots before and after each event execution. Every time an event is executed by the crawler, the subtree of the DOM that has changed as a result of the event execution is considered a component.

The way we compare the DOM trees to obtain the changed subtree is defined as follows: suppose the DOM-tree before the event execution is T_{before} and the DOM tree after the event execution is T_{after}. We traverse T_{before} using breadth-first search. For each node x in T_{before}, we compute the path from root to x, and find the node y in T_{after} that has the same path. If x and y are different, or have different number of children, x is considered the root of a component; its XPath is added to the stateDictionary if not already existing, and the search is discontinued in the subtree of x . If several such nodes y exist in T_{after}, their deepest common ancestor is used as the root of the component.

Initially, the stateDictionary contains only one component with XPath "/". Additional components are discovered and added to the stateDictionary as the crawling proceeds. The algorithm is summarized in the pseudo-code below. One important practical point to note is that the discovery of a new component can lead to a modification of previously known component states, if the new component is nested inside these component states. As explained before, the new component must be pruned from the containing component states, so their component state ID must be recomputed. It is not practical to save all component states DOM to be able to recompute their ID when this occurs. Instead, in our prototype we mark these component states as invalid and visit them again later during the crawl.

4 Experimental Results

4.1 Test Cases

In order to evaluate the efficiency of our method, we have run some crawling experiments with a number of experimental and real RIAs. We split these results in two categories. In the first category, we have seven simple RIAs[4]. Two of these are test application that we have built ourselves for testing purpose, while the five other ones are real, but simple (or simplified) RIAs. We have also run our

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

Algorithm 2 Pseudo-code of proposed crawling algorithm

```
1: procedure ComponentBasedCrawl
2:     for as long as crawling goes do
3:         event := select next event to be executed based on the exploration strategy
4:         execute (event)
5:         delta := diff (dom_before, dom_after)
6:         xpath := getXpath (delta)
7:         if stateDictionary does not contain xpath then
8:             add xpath to stateDictionary
9:         end if
10:        resultingStates := FindCurrentStates(delta)
11:        for all state in resultingState do
12:            if stateDictionary does not contain state then
13:                add state to stateDictionary
14:            end if
15:            event.destinations := resultingStates
16:        end for
17:    end for
18:    return stateDictionary
19: end procedure
```

test on two real "complex" RIAs: IBM *Rational Team Concert* (RTC[5]), an agile application life-cycle management web-based application, and *MODX*[6], an open source content management system. For these two test cases, the complexity of the web site made it impossible for us to crawl with classical method for comparison, so we report the results separately in Section 4.3. Note that the number of test cases is not as large as we would like, but we are faced with the limitation of the tools we use to execute the crawl on RIAs[7]. We provide the characteristics of the model built for each of these nine RAIs in table 2. Note that these are the numbers for our component-based model, which is much smaller than the classical DOM-based model (see [27] for more details).

Name	**# States**	**# Trans.**	**Type**	**Name**	**# States**	**# Trans.**	**Type**
Bebop	119	774	Simple	TestRIA	67	191	Test
Elfinder	152	3,239	Simple	Altoro	87	536	Test
Periodic	365	2,019	Simple	RTC	432	3,667	Complex
Clipmarks	31	377	Simple	MODX	1,291	7,868	Complex
DynaTable	24	49	Simple				

Table 2. Applications tested, along with their number of states and transitions in component-based model.

[5] `https://jazz.net/products/rational-team-concert`

[6] `http://modx.com`

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

We have implemented all the mentioned crawling strategies in a prototype of IBM® Security AppScan® Enterprise[8]. Each strategy is implemented as a separate class in the same code base, so they use the same DOM equivalence mechanism [28], the same event identification mechanism [29], and the same embedded browser. For this reason, in Section 4.2 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.

4.2 Results on simple RIAs

This first set of test case were simple enough to allow crawling with the traditional method. We report here comparisons with the greedy exploration [6] and the probability strategy [8], which are known to be to two most efficient strategy for building an exhaustive model [9]. This gives us complete knowledge of the model, allowing us to see whether our optimized strategy provides 100% coverage.

4.2.1 Complete Exploration Cost Our first set of results are about the "total exploration costs", that is, the cost of finishing the crawl, expressed in terms of number of events executed. Most results are detailed in Figure 5. As can be seen, our component-based crawling method consistently outperforms both probability method and the greedy method by a very wide margin. The difference is more dramatic in RIAs that have a complex behavior, though even for the smaller ones, TestRIA and Altoro Mutual (not shown), the cost of component-based crawling is about 30% of the cost of the other methods. The best example among our test cases is Bebop, which contains very few data items shown on the page, but can sort and filter and expand/collapse those items in different manners. Even in an instance of the RIA with only 3 items, component-based crawling is 200 times more efficient than the other methods. This difference in performance quickly gets even bigger in an instance of the RIA with more items, as shown in Section 4.2.4.

4.2.2 Time Measurement Since component-based crawling requires a fair amount of computation at each step, we also measured time in similar experiments to ensure this processing overhead does not degrade the overall performance. As can be seen on Table 3, even in terms of absolute time component-based crawling significantly outperforms the two other methods.

[8]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.

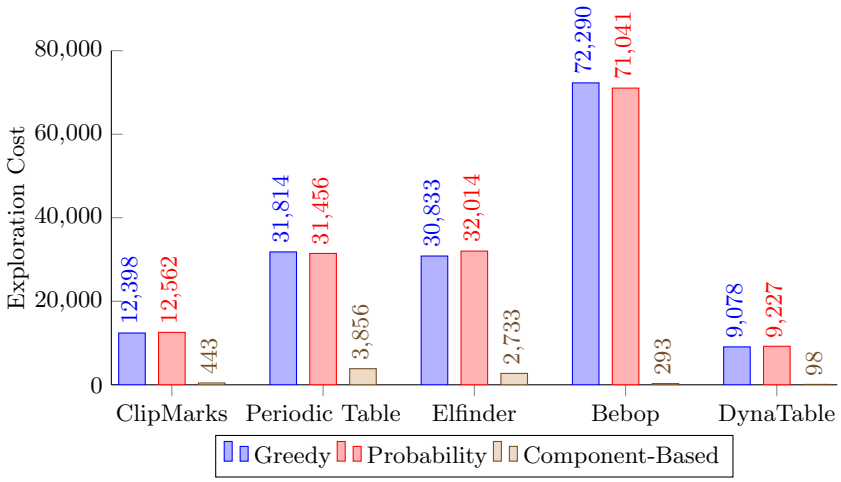

Fig. 5. Comparison of exploration costs of finishing crawl for different methods

	TestRIA	Altor-Mutual	ClipMarks	Periodic Table	Elfinder	Bebop	DynaTable
Greedy	00 : 00 : 18	00 : 00 : 34	00 : 03 : 38	01 : 13 : 08	00 : 51 : 22	01 : 25 : 11	00 : 05 : 35
Probability	00 : 00 : 11	00 : 00 : 20	00 : 02 : 50	01 : 09 : 42	00 : 49 : 00	01 : 17 : 32	00 : 04 : 51
Component-Based	00 : 00 : 06	00 : 00 : 04	00 : 00 : 13	00 : 01 : 21	00 : 08 : 21	00 : 00 : 29	00 : 00 : 06

Table 3. Time of finishing crawl for different methods (hh:mm:ss).

4.2.3 Coverage Unlike previous DOM-based methods, component-based crawling does not guaranty complete coverage. This is because the method is based on discovering automatically independent components, and if the method wrongly identifies as "components" sections of the DOM that are not independent from each other, some coverage might be lost. It is difficult to know in general the amount of coverage that can be lost, but in the case of our seven test cases, our method systematically reached 100% coverage. No information was lost despite the dramatic decrease in crawling time.

4.2.4 Scalability Some of the test RIAs that we used had to be significantly "trimmed" before they could be crawled by the traditional methods. One example is Clipmark, which displays a number of items on the page. Although it had initially 40 items, we had to reduce it to 3 in order to finish the crawl with the traditional methods! When using component-based crawling, on all of our test beds we were easily able to finish the crawl on the original data, and we could increase the number of items beyond that without problem. We show the data for two examples, Clipmarks on Figure 6 and Bebop on Figure 7. As can clearly been seen on both examples, while the crawling time increases linearly with the number of items in the page when using component-based crawling, it grows exponentially with the DOM-based, greedy method and soon becomes intractable. The results are the same with probability, and will *necessarily* be

similar for *any* DOM-based crawling method, since the size of the end-model itself increases exponentially with the number of items in the page. This shows that component-based crawling is able to crawl and index RIAs that are simply out of reach to any DOM-based strategy.

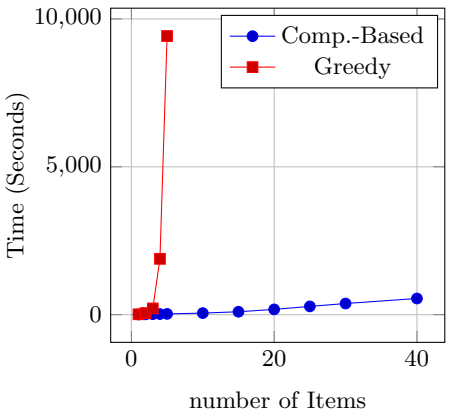

Fig. 6. Scalability with Clipmarks.

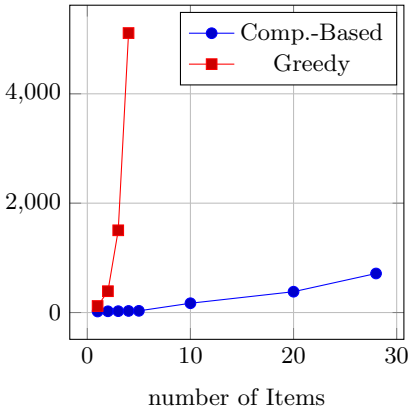

Fig. 7. Scalability with BeBop.

4.3 Results on Complex RIAs

When crawling large and complex RIAs, comparison with DOM-based methods do not tell much, since these methods are essentially unable to crawl them. In addition, for many of these RIAs (e.g. Facebook, Gmail etc.), the amount of data available is very large. "Finishing" the crawl is often not a realistic proposition. Instead, the question becomes how efficient the crawl can be as it progresses overtime. In order to measure this, we focus on the question of crawling for indexing, where we argue that a fair definition of "efficient" is the ability of the crawler to keep finding new content. In our experiment, we have measured how much the textual information accumulated increases overtime. An efficient method will provide a steady increasing amount of information, while an inefficient method might stop providing any new information for long period of times (basically re-fetching known data many times). We have measured how "efficient" component-based crawling is, by counting how many "lines" of text (excluding html tags) are accumulated overtime. As can been seen from the Figures 8 and 9, for both of our examples, the method provides a nice steadily increasing line, showing that the method is efficient at fetching new information overtime. An inefficient method would have plateaued, during which the crawl is not adding any new data.

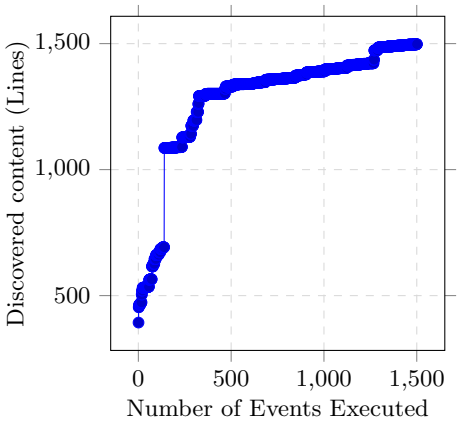

Fig. 8. Progress overtime with RTC.

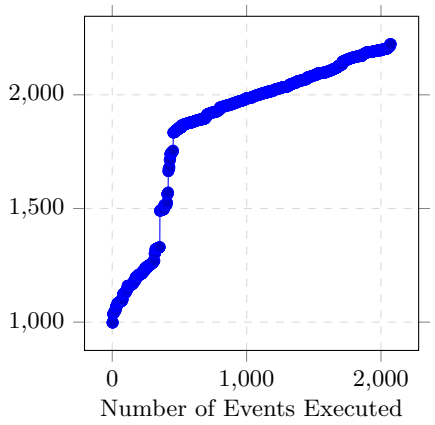

Fig. 9. Progress overtime with MODX.

5 Conclusions and Future Work

This work addresses one of the most prevalent problems in the context of crawling AJAX-based RIAs: state space explosion. The presented method detects independent components and models the RIA at component level rather than DOM level, resulting in exponential reduction of the overall crawling complexity, with minimal or no loss of data coverage. The method captures the effect of event execution more precisely, resulting in a simpler model with fewer states and transitions. The produced model can point to any desired data with an event execution trace from the initial state, but cannot necessarily produce a path to lead to any valid DOM-state. The methods has been implemented using a greedy exploration strategy and DOM diff as automatic component discovery algorithm. Our experimental results verify the significant performance gain of the method while covering equal content as DOM-based methods.

This work can be improved in several areas. In particular, our future work include devising better algorithms for component discovery. One important improvement can be done on detection and handling of violations: currently, we have no effective way of recovering from a situation where components that have been assumed to be independent turn out not to be. We simply ignore theses violation. Although that did not impact negatively our experimental results, we cannot be sure that this won't be the case on every RIAs. A naive approach for detecting violations and adapting the strategy accordingly is not particularly difficult, but it would be too costly to be practical.

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.

References

1. Fraternali, P., Rossi, G., Sánchez-Figueroa, F.: Rich internet applications. Internet Computing, IEEE **14**(3) (2010) 9–12
2. Duda, C., Frey, G., Kossmann, D., Zhou, C.: Ajaxsearch: crawling, indexing and searching web 2.0 applications. Proceedings of the VLDB Endowment **1**(2) (2008) 1440–1443
3. Duda, C., Frey, G., Kossmann, D., Matter, R., Zhou, C.: Ajax crawl: making ajax applications searchable. In: ICDE'09, IEEE (2009) 78–89
4. Amalfitano, D., Fasolino, A.R., Tramontana, P.: Reverse engineering finite state machines from rich internet applications. In: Proceedings of WCRE, IEEE (2008) 69–73
5. Amalfitano, D., Fasolino, A.R., Tramontana, P.: Rich internet application testing using execution trace data. In: Proceedings of ICSTW, IEEE (2010) 274–283
6. Peng, Z., He, N., Jiang, C., Li, Z., Xu, L., Li, Y., Ren, Y.: Graph-based ajax crawl: Mining data from rich internet applications. In: Proceedings of ICCSEE. Volume 3. (march 2012) 590 –594
7. Dincturk, M.E., Jourdan, G.V., Bochmann, G.v., Onut, I.V.: A model-based approach for crawling rich internet applications. ACM Transactions on the WEB (2014) to appear
8. Choudhary, S., Dincturk, M.E., Mirtaheri, S.M., Jourdan, G.V., v. Bochmann, G., Onut, I.V.: Model-based rich internet applications crawling:menu and probability models. Journal of Web Engineering **13**(3) (2014) to appear
9. Choudhary, S., Dincturk, M.E., Mirtaheri, S.M., Jourdan, G.V., v. Bochmann, G., Onut, I.V.: Building rich internet applications models: Example of a better strategy. In: Web Engineering. Volume 7977 of LNCS., Springer (2013) 291–305
10. Faheem, M., Senellart, P.: Intelligent and adaptive crawling of web applications for web archiving. In: Web Engineering. Volume 7977 of LNCS. Springer (2013) 306–322
11. Amalfitano, D., Fasolino, A.R., Polcaro, A., Tramontana, P.: The dynaria tool for the comprehension of ajax web applications by dynamic analysis. Innovations in Systems and Software Engineering (2013) 1–17
12. Abu Doush, I., Alkhateeb, F., Maghayreh, E.A., Al-Betar, M.A.: The design of ria accessibility evaluation tool. Advances in Engineering Software **57** (2013) 1–7
13. Mesbah, A., van Deursen, A.: Invariant-based automatic testing of ajax user interfaces. In: ICSE. (may 2009) 210 –220
14. Amalfitano, D., Fasolino, A.R., Tramontana, P.: A gui crawling-based technique for android mobile application testing. In: Proceedings of ICSTW, Washington, DC, USA, IEEE Computer Society (2011) 252–261
15. Amalfitano, D., Fasolino, A.R., Tramontana, P., De Carmine, S., Memon, A.M.: Using gui ripping for automated testing of android applications. In: Proceedings of ASE, New York, NY, USA, ACM (2012) 258–261

16. Erfani, M., Mesbah, A.: Reverse engineering ios mobile applications. In: Proceedings of WCRE. (2012)
17. Mesbah, A., Bozdag, E., van Deursen, A.: Crawling ajax by inferring user interface state changes. In: Proceedings of ICWE, IEEE (2008) 122–134
18. Ayoub, K., Aly, H., Walsh, J.: Dom based page uniqueness identification, canada patent ca2706743a1 (2010)
19. Milani Fard, A., Mesbah, A.: Feedback-directed exploration of web applications to derive test models. In: Proceedings of ISSRE, IEEE Computer Society (2013) 10 pages
20. Choudhary, S., Dincturk, M.E., Mirtaheri, S.M., Moosavi, A., von Bochmann, G., Jourdan, G.V., Onut, I.V.: Crawling rich internet applications: the state of the art. In: CASCON. (2012) 146–160
21. Mirtaheri, S.M., Dinçtürk, M.E., Hooshmand, S., Bochmann, G.V., Jourdan, G.V., Onut, I.V.: A brief history of web crawlers. In: Proceedings of CASCON, IBM Corp. (2013) 40–54
22. Bezemer, C.P., Mesbah, A., van Deursen, A.: Automated security testing of web widget interactions. In: Proceedings of ESEC/FSE, ACM (2009) 81–90
23. Chen, A.Q.: Widget identification and modification for web 2.0 access technologies (wimwat). ACM SIGACCESS Accessibility and Computing (96) (2010) 11–18
24. Crescenzi, V., Mecca, G., Merialdo, P., et al.: Roadrunner: Towards automatic data extraction from large web sites. In: VLDB. Volume 1. (2001) 109–118
25. Harel, D.: Statecharts: A visual formalism for complex systems. Science of computer programming **8**(3) (1987) 231–274
26. Peng, Z., He, N., Jiang, C., Li, Z., Xu, L., Li, Y., Ren, Y.: Graph-based ajax crawl: Mining data from rich internet applications. In: Proceedings of ICCSEE. Volume 3., IEEE (2012) 590–594
27. Moosavi, A.: 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`.
28. Benjamin, K., Bochmann, G.v., Jourdan, G.V., Onut, I.V.: Some modeling challenges when testing rich internet applications for security. In: Proceedings of ICSTW, Washington, DC, USA, IEEE Computer Society (2010) 403–409
29. Choudhary, S., Dincturk, M.E., Bochmann, G.V., Jourdan, G.V., Onut, I.V., Ionescu, P.: Solving some modeling challenges when testing rich internet applications for security. Proceedings of ICST (2012) 850–857