Component-Based Crawling of Complex Rich Internet Applications

Ali Moosavi

Thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of the requirements

For the degree of

Master of Computer Science

School of Electrical Engineering and Computer Science

Faculty of Engineering

University of Ottawa

Abstract

During the past decade, web applications have evolved substantially. Taking advantage of new technologies, Rich Internet Applications (RIAs) make heavy use of client side code to present content. Web crawlers, however, face new challenges in crawling RIAs. The problem of crawling RIAs has been a focus for researchers during recent years, and solutions have been proposed based on constructing a state-transition model with DOMs as states and JavaScript events as transitions. When faced with real-life RIAs, however, a major problem prevalent in current solutions is state space explosion caused by the complexity of the RIAs. This problem prevents the automated crawlers from being usable on complex RIAs as they fail to produce useful results in a timely fashion. This research addresses the challenge of efficiently crawling complex RIAs with two main ideas: component-based crawling and similarity detection. Our experimental results show that these ideas lead to a drastic reduction of the time required to produce results, enabling the crawler to explore RIAs previously too complex for automated crawl.

Acknowledgement

I would like express my gratitude to my supervisor, Dr. Guy-Vincent Jourdan and my co-supervisor Dr. Iosif Viorel Onut for their constant support and motivation throughout my graduate studies. Their patience and guidance made this work of research possible. I would like to thank Dr. Gregor von Bochmann for his engagement in the project and providing insightful inputs.

I cannot be more thankful to my parents, Seyed Khalil Moosavi and Fakhrolsadat Moosavi for all their love and support. Without them, living abroad and studying at postgraduate level would be impossible for me.

I am very grateful to my colleagues in Software Security Research Group at University of Ottawa Emre Dincturk, Salman Hooshmand, Suryakant Choudhary, Seyed M. Mirtaheri, Di Zou, and Khaled ben Hafaiedh for accompanying and helping me during this whole time, and staying by my side as friends.

Finally, I would like to acknowledge financial support of the National Sciences and Engineering Research Council (NSERC), and IBM Center of Advanced Studies (CAS).

Table of Contents

Table of Figures

List of Tables

1. Introduction

In today's world, crawlers are in charge of various tasks. They need to adapt to the ever-changing web technologies and trends. Traditionally, web applications consisted of a set of pages accessible through unique URLs. The server side carried out any computations and the client side was only responsible for rendering the results. Later, newer web technologies such as AJAX [1], Flash [2] and HTML5 [3] transformed this architecture and enabled a new breed of web applications called Rich Internet Applications (***RIA***s). RIAs provide more sophisticated client side functionality, improving user experience and reducing communication between the client side and the server side. Today, these technologies are in widely used. Crawlers, however, have problems in exploring RIAs automatically. Several studies have been conducted around crawling and testing RIAs in general, and AJAX-based RIAs in particular. Current solutions in this recent research area present limited capabilities and fail to operate effectively on complex examples. To our knowledge, currently, no major industrial players use real RIA crawling techniques. Instead, they ask RIA owners to provide their content in a crawler-friendly manner [4]. This research work focuses on developing crawling solutions with acceptable performance on large-scale and complex AJAX-based RIAs, with the aim of creating technology that suits industrial needs.

In this chapter we discuss crawlers and why they face challenges when dealing with RIAs in section 1.1. We then describe an existing shortcoming in the current solution that is in use by research prototypes, and the importance of resolving this issue in section 1.2. Finally, in section 1.3 we summarize the contributions of this research work in addressing this challenge.

1.1. **Crawling RIAs**

Crawling is the process of exploring a web application automatically. Crawlers are essential tools in today's world of web-oriented applications and services. Crawlers are used for a variety of purposes, such as content indexing (for example for use by a search engine) [5] [6], automated regression testing (as part of software development process) [7], black-box security and accessibility assessment [8], [9].

With the advent of new web technologies such as AJAX and Flash, there has been a shift in web applications design towards putting more complexity on the client side in the form of executable code. Increasingly, more and more modern web applications rely heavily on client-side code to fetch and present their content. By using these technologies RIAs can make incremental updates to the client state of the application, rather than loading complete pages from the server. In the case AJAX-based RIAs, for example, the application can use JavaScript code to manipulate the DOM on the client side, optionally contacting sever and adding new data to client state without changing the URL.

As these technologies are already in widespread use, it is more important than ever before for crawlers to support them. While using these technologies has provided benefits for users such as increased interactivity and responsiveness, they introduce challenges for crawlers.

1.1.1. **Challenges**

Traditional methods of crawling are not sufficient to cover complete content of a RIA, since these methods are built on assumptions that are no longer valid in RIAs. Traditionally, crawlers use Unified Resource Locators (URLs) to navigate through the web. A web crawler is fed with a list of seed URLs that it starts from. For each URL, it loads the page and adds any URLs linked from that page to its working queue [10]. If the link is already visited, there is no need to visit it again. Once all discovered URLs are visited, the crawling job is finished, as it has covered all the content that is reachable from the seed URLs via hyperlinks. This method is based on the assumption that URLs correspond to client states. While this method is sufficient for crawling traditional web applications, RIAs break the functionality of this method in two ways.

Firstly, as stated earlier, a RIA can update its client state without making a change in the URL. Therefore, client states in RIAs no longer have a one-to-one correspondence to URLs. Executable objects on the client side can alter the client state and present new data that is important for the crawler, with or without contacting the server. Therefore, the crawler should have a clear distinction between client states and URLs, as now many client states are possible within the same URL. In the case of AJAX-based RIAs, it is executing JavaScript Events (simply called ***'events'*** in this thesis) that can alter the client state, and replace the use of URLs in traditional web applications as a means to reach different states. It is possible to build a complete RIA with a single URL using AJAX. As a result, only visiting URLs is not sufficient to cover the content of a RIA anymore. To ensure content coverage, the crawler should have a method to explore all client states under the same URL.

Secondly, events have a more complicated behaviour than URLs. It is usually safe to assume that navigating to the same URL, from anywhere in the website, will always result in the same webpage. This is not the case for events, though. The result of execution of an event has more determinant factors than that of navigating to a URL. Since events can read data from the client state of the application to determine what to do, they can be "***state-dependent***". The crawler needs to examine the same state-dependent event from different client states in order to ensure proper coverage.

Due to the aforementioned challenges, crawling RIAs needs different techniques and methods than crawling traditional web applications. Currently, industrial search engines provide no better solution other than asking RIA owners to manually provide "html snapshots" of their content to make them searchable [4]. This approach puts the burden of providing information on the shoulders of the programmers instead of crawlers, and enforces a big maintenance cost since html snapshots are to be manually kept up-to-date whenever there is an update to the RIA. This contradicts the goal of crawlers whose purpose is to aid in maintenance of a web application by automatic scanning and reporting issues.

The problem of crawling AJAX-based RIAs has been a focus of research studies during the past few years. These research works commonly use a state-transition model to represent a RIA, which is introduced briefly in the following section.

1.1.2. **The State of the Art Solution: The State-Transition Model**

Common approach for RIA crawling in the studies is to define client states based on the Document Object Model (***DOM***) on the client. The RIA is then modelled as a finite state machine, where DOMs are represented as states and event executions are represented as transitions. By executing events, the crawler can navigate the RIA and reach different DOM-states. The problem of crawling a RIA is then modelled as walking in an unexplored directed graph. This model and its assumptions and limitations will be elaborated in Chapter 2, together with the approaches in the literature for crawling RIAs using this model.

The research topic of crawling RIAs is relatively new and the amount of research work in this area is limited. Research works mostly focus on other aspects of RIA crawling, such as parallelizing the crawl or performing security tests using the extracted model. While these works have been successfully applied to sample test cases, their applicability is subject to the limitations of the crawling method they use. To our knowledge, the state-transition model is the only major model presented for crawling complete content of RIAs, and several research projects use it in their crawling method. This model, however, quickly loses scalability as the RIA complexity grows.

Many RIAs today are feature-rich applications, rather than merely a set of pages. These "***complex RIAs***" have several functionalities, each acting independently of the others. Examples include social networking sites, widget-based RIAs, Content Management Systems and more. The user is free to choose among many actions at all times, and each different combination of these actions shapes the DOM differently. The state-transition model faces a state space explosion problem when applied to complex RIAs. This problem is the main motivation behind this research to develop methods of crawling that can run on complex RIAs.

1.2. Motivations

A major challenge affecting current research works is state space explosion. This problem has been reported several times in publications from various research teams working in this area [11], [12], [13]. Real-life RIAs tend to produce a large number of states in the state-transition model. Even a RIA with a limited set of functionalities can easily present a large number of different DOMs, the majority of which do not contain interesting information for the crawler. For example, many new DOMs can be generated simply by presenting a different combination of already-presented data. As a result, not only does the crawl take an excessive amount of time, and the user might have to terminate the crawl prematurely, but this also leads to the production of extremely large models, which in turn makes analyzing or testing the model expensive and impractical [14]. Moreover, in the presence of time limits, the crawler might spend its valuable time on exhaustive crawling of irrelevant regions of the RIA, leading to a model that has poor functionality coverage despite its large size.

Without a proper crawling method that can grasp complex RIAs and deduce a reasonable-sized model from them, tools that rely on crawling will be unusable for real-life scenarios. Tools and techniques developed in research studies need to be able to handle industrial use cases in a timely manner in order to be applicable in industrial needs. This research is partly funded by IBM, with the aim of developing a crawling method suitable for industrial use.

We aim to address the challenge of crawling complex RIAs by introducing a novel method for crawling, called "***Component-Based Crawling***". Component-based Crawling breaks down the state space by capturing independent portions of the DOM tree and assigning separate states to them. Component-based crawling is able to cover complete content of a RIA in a substantially more efficient manner than the current methods, without running into state space explosion where current methods do. We also present a useful technique "***Similarity Detection***", which helps covering as much functionality of the RIA as possible in a limited time by detecting similar structures and events and avoiding them in order to diversify the crawl. Both the methods are filed by IBM as patents [15], [16] and implemented in prototype versions of IBM AppScan Enterprise (ASE) [17].

1.3. List of Contributions

The major contributions of this work are summarized in the following list:

– A meta-model for expressing a RIA as a set of independent components and their interactions
– An algorithm for crawling complex RIAs using the abovementioned meta-model
– An algorithm and criterion for predicting similar portions of a RIA and diversifying the crawl

Moreover, in order to achieve and validate the above-mentioned goals, we also provide:

- Implementation of the abovementioned techniques as a working RIA crawler
- Experimental studies on the performance of the abovementioned techniques and comparison against state of the art techniques
- Experimental studies on the scalability of component-based crawling as data in a RIA grows

1.4. Organization of the Thesis

This thesis is organized as follows:

Chapter 2 provides a detailed description of the state-transition model, its assumptions and limitations. It discusses how state of the art research works define the model, and attempt to avoid state space explosion. Chapter 3 describes the Component-Based Crawling method by first describing the meta-model and then the algorithm that uses the meta-model for crawling. Chapter 4 provides our experimental results on the effectiveness of this method. Chapter 5 describes the Similarity Detection technique, and Chapter 6 provides conclusion marks possible future directions for this work of research.

2. The State of the Art

In this chapter, we present an overview of current research for crawling AJAX-Based RIAs. After a brief introduction to some concepts in this field in section 2.1, in section 2.2 we provide a detailed description of the State-Transition model, the method commonly used for crawling RIAs. We then follow with a discussion on how different studies use various versions of this model. Next, in section 2.3 we discuss the state space explosion problem, various techniques proposed in the literature for tackling this problem and their effectiveness, before introducing our proposed techniques in latter chapters.

2.1. Introduction

There are a few terms and abbreviations used in this document that might need description for a reader unfamiliar with this topic. When a browser loads an HTML document, it builds a "***DOM***" (Document Object Model), which is a structural representation of the document [18]. The DOM provides a language independent interface for scripting languages such as JavaScript [19] to access the structure of the document. The browser allows these scripting languages to modify the DOM, and renders the modifications for the user. When the client reloads the URL of the document (i.e. issues a "***reset"***), the document is fetched from the server again and the DOM is rebuilt to its initial un-modified state. "***AJAX***"(Asynchronous JavaScript and XML) is a technology that allows the client side JavaScript code to communicate with a server asynchronously (in the background) without interfering with the display and behavior of the existing page [20].

In order to locate nodes in a DOM (or an XML document), a standard query language called ***XPath*** (XML Path Language) is used [21]. A node's path representation using XPath is not unique, as XPath syntax allows several ways to query nodes. Likewise, an XPath query may return multiple nodes. The way we utilize XPath queries in this work is discussed in section 3.3.2.

2.1.1. Architecture of an AJAX-based RIA

Web applications use a client-server architecture. The state of the client side of the application consists of the DOM tree, the URL, possible cookies, etc. In RIAs, the client must also provide an execution environment to encompass and run the client side code of the RIA, therefore there are additional elements that determine the client state (e.g. the value of JavaScript variables in an AJAX-Based RIA). In an AJAX-Based RIA, triggering execution of a JavaScript event can result in changes to the client state, and possible message exchanges with the server, making a change in the server state as well. Crawlers, just like users, do not have access to the server state. They typically make certain assumptions about the server states to assume completeness of their crawl. For example, the simplest form of assumption would be inexistence of server states, in which case the crawler is allowed to cache client states and restore them at will, without informing the server.

AJAX, as its full name suggests, provides the possibility of asynchronous communication between the client and the server. Asynchronous communication means that when the browser sends a request to the server, it does not block the user and allows her to continue interacting with the web application. Therefore, the user can generate more requests to the server at the same time she is waiting for the response of the previous requests. An asynchronous interaction scenario with AJAX is depicted in Figure 1.

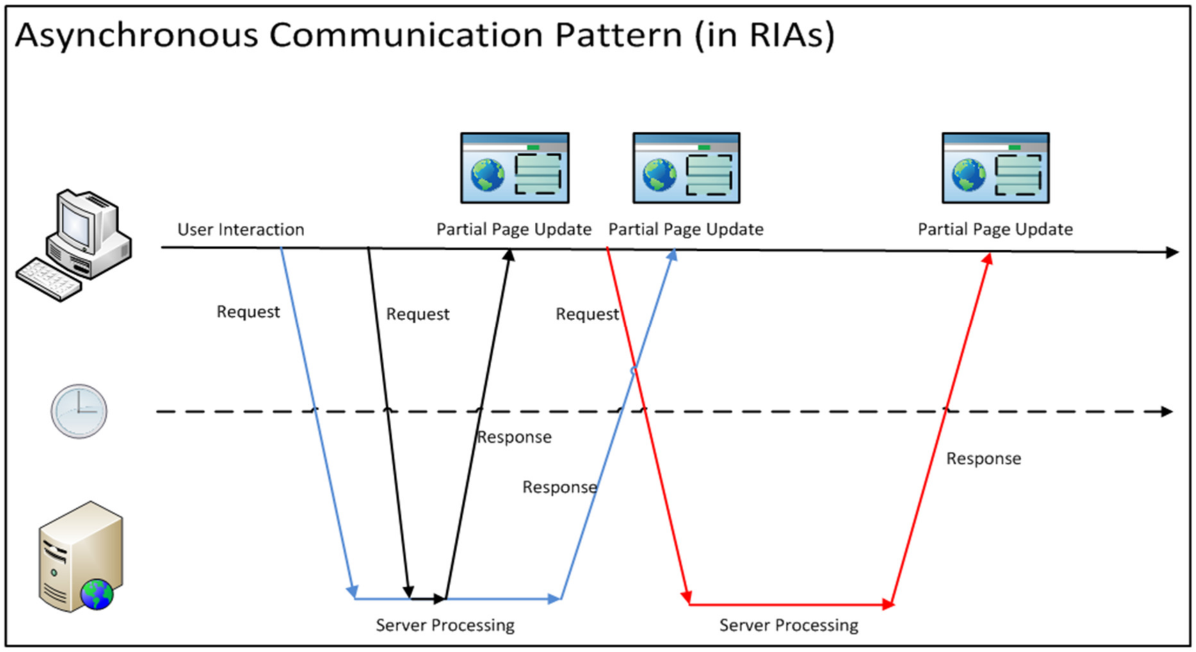


Figure 1: Example of asynchronous communication using AJAX calls (appeared in **[42]**)

Upon sending an AJAX request to server, one must also specify its callback method. The callback method is the code that is to be run when the response arrives, to handle the response data. The callback method can modify the client state using the data received, for example updating part of the DOM.

2.2. The State-Transition Model

Common approach in studies is to model a RIA as a finite state machine (FSM), and use DOM as an identifier for client states. In the FSM, states represent DOMs and transitions

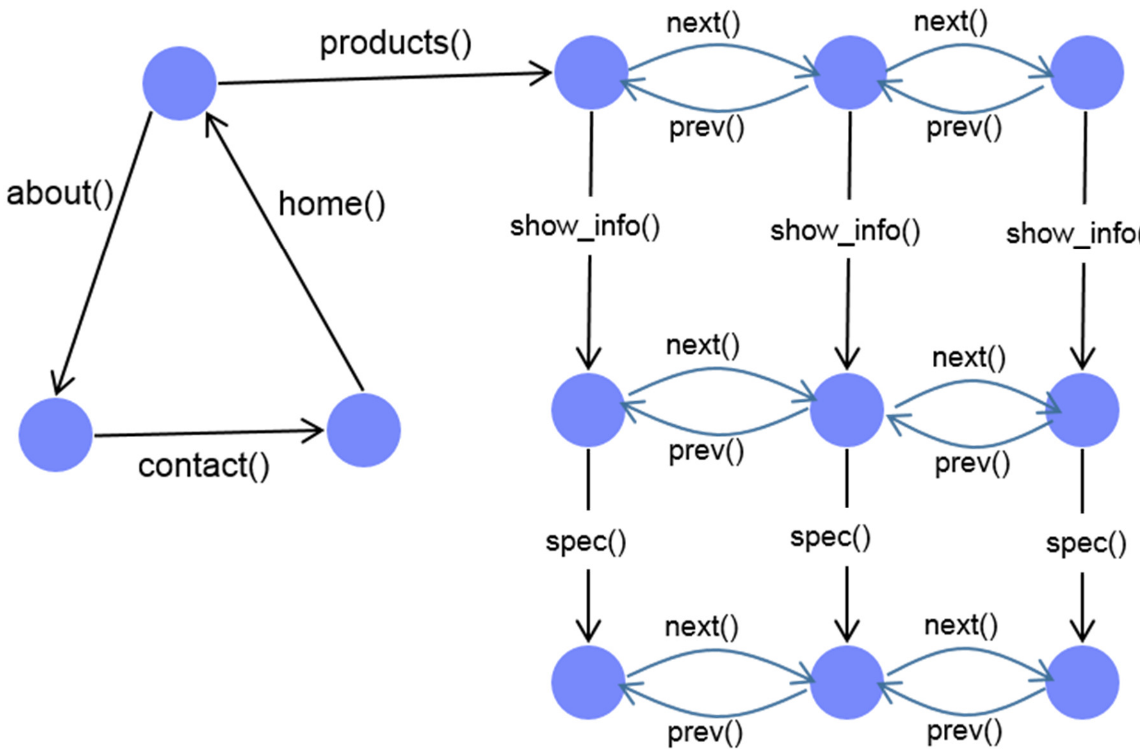


Figure 2. An example of a simple state-transition model

represent event executions. Events can lead from one DOM-state[1] to another.

The FSM can be defined as a tuple $M = (S, s_1, \Sigma, \delta)$ where S is the set of DOM-states, s_1 is the initial DOM-state (when the URL of the RIA is loaded), Σ is the set of events, and δ is a function $S \times \Sigma \rightarrow S$ that defines the set of valid transitions. At any given time, the application is in one DOM-state, referred to as the current DOM-state. When $\delta(s_i, e) = s_j$, it means that we can reach to DOM-state s_j by executing event e from DOM-state s_i. Not all events are available in all DOM-states; therefore, δ is a partial function.

One simplifying assumption that is usually made is that the behaviour of the RIA is deterministic from the point of view of the crawler. This means that if we go back to a visited DOM-state using valid transitions and execute an event that we had explored before, the resulting DOM-state will be the same as before. Hence we are allowed to model the RIA as a deterministic FSM. Based on this assumption, by executing each event form each DOM-state once and building a complete FSM model, the crawler can assume the crawling is done, and that the resulting FSM is a representative model of the system.

However, the crawler is not allowed jump to arbitrary DOM-states at will (e.g. by saving DOM-state in advance and restoring it when desired). Instead, it has to take available transitions in order to transfer between DOM-states. This is done to ensure the RIA is being explored as it was intended to be explored by a user. Jumping between arbitrary DOM-states would bypass any server communication that would take place along the way,

[1] Related works in the literature commonly refer to DOM-states simply as "states". In this work, in order to emphasize the difference between "component-states" which we use in our method and DOM-states, we refer to them explicitly as DOM-states.

possibly breaking the functionality of the RIA. 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 URL of the initial page) to go to the initial DOM-state of the RIA and take a transfer sequence from there. Resets are usually modelled as special transitions from all DOM-states to the initial DOM-state.

At 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.

An FSM $M = (S, s_1, \Sigma, \delta)$ can also be represented as a directed graph $G = (V, E)$, where V is a set of vertices, and E is a set of directed and labelled edges, where $(v_i, v_j; e)$ is an edge from vertex v_i to vertex v_j with label e. We can define a bijective function $R: S \to V$ between states in the FSM and vertices in the graph. The transitions in the FSM will correspond to edges in the graph:

$$\delta(s_i, e) = s_j \Leftrightarrow \big(R(s_i), R(s_j); e\big) \in E$$

The graph can optionally be a weighted graph to reflect time cost of each transition.

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 the crawling is finished. Resets do not need to be ‘uncovered’ since their behavior is known, but they can be used as auxiliary edges when using a path to reach from one node to another.

Based on this model, different exploration strategies (such as Depth-First-Search (***DFS***), Greedy and Model-Based strategies) have been suggested and evaluated by related works on sample experimental RIAs. Comparing different exploration strategies is usually done based on the sum of all events and resets executed during the crawl (possibly considering the time cost of each of them) until finished, which we refer to as "***exploration cost***".

2.2.1. **Assumptions**

In order to function properly, the state-transition model makes certain assumptions about the RIA. We list these assumptions here for clarity:

– **Server states:** As stated before, the crawler has no access to server states. It only crawls based on observing client states and assumes that server states have no impact on the determinism of the model. Issuing a ‘reset’ is assumed to always take the client to its initial state, and executing an event from a client state is assumed to always lead to the same client state. If there is a change in the server state that is not directly observable on the client side, this assumption is violated. This can potentially affect the behaviour of events, and result in non-deterministic behaviour of those events from the crawler’s point of view.

– **Serializing AJAX calls:** As explained before, AJAX calls work asynchronously, and browsers do not prevent users from triggering additional events while other events are still pending. One simplifying assumption that the state-transition model makes, however, is that events are executed one after another. Once an event is triggered, the crawler waits for the response of the any AJAX calls made to the server to be received and processed fully before declaring the new client state as the destination of the event.

- **User inputs:** User inputs are also modelled as events. However, the number of values that can be entered (for example, in a text field), is very high. It is usually infeasible to try all possible values during the crawl. Instead, we assume that the crawler is provided with a set of user inputs to be used. The completeness of the model is then subject to the values provided. The problem of generating a comprehensive set of user inputs is not specific to RIAs. Any general methods used for this purpose can be applied in the context of RIA as well. Example research studies in this field can be found in [22], [23], [24], [25], [26].

2.2.2. **Use in the literature**

Several works study crawling RIAs using the state-transition model. In one of the earliest works, Duda et al. assume the ability to cache and restore DOM-states at will [11]. They use a Breadth-First-Search (***BFS***) algorithm to explore a RIA in [11], [27], [28], [29]. However, as stated beforehand, this assumption limits the crawling capability in the existence of server states.

Amalfitano et al. also focus on obtaining a state-transition model from a RIA, and using the model for generating test suits for the RIA. In their initial work [30], they use manual user-sessions to extract execution traces and build a model. In their follow-up paper [31] they automate their tool by using a DFS exploration strategy. Mesbah et al. [32] introduce their tool 'CrawlJax' for crawling and extracting a model from a RIA. It is able to take a static html snapshot of each DOM-state and build a non-AJAX version of the website in the end. CrawlJax also uses DFS algorithm to explore the RIA. By default, CrawlJax assumes all events in the RIA to be "***state-independent***" events, and explores each event only from the DOM-state where the event was first encountered. The event is not explored on the subsequently discovered DOM-states. This results in a partial coverage the RIA in the existence of state-dependent events, which is commonly the case. CrawlJax can also be configured to explore all events in each DOM-state. The authors use CrawlJax in many subsequent papers and focus on multi-threading [33], security testing on the obtained model [34] [7], etc.

Some research works focus on improving the efficiency of crawling a RIA by focusing on the exploration strategy used. Peng et al. propose using a greedy algorithm as exploration strategy in [35] that out-performs DFS and BFS exploration significantly. Our work uses the same greedy approach for its exploration strategy. The authors of [36], [37], [38] introduce various model-based crawling strategies to be used as exploration strategy, and sum up the performance evaluation of all their exploration strategies in [12].

The crawler needs to have a DOM equivalency function to compare the current DOM-state against the previous ones, and determine if it is equal to any previous DOM-states or not. It is a common approach consider DOMs with minor differences as the same DOM-state. The DOM equivalence function varies among different research works. [27] Uses strict equality to compare DOMs. CrawlJax uses a distance function to compute the edit distance between different DOMs [39], and considers them as the same DOM-state if the distance is below a certain threshold. The works presented in [40], [37], [38] apply some reduction and normalization functions presented in [41] on the DOM to exclude the irrelevant data before comparing DOMs

2.3. The Problem: State Space Explosion

One major challenge in this field is state space explosion. Most RIAs tend to present a very large number of DOM-states that cannot be crawled in a reasonable time. Usually in RIAs, numerous events exist in each DOM-state, and each makes a slight change to the DOM. It is possible for different combinations of these events to result in many different DOM-states that have no new data, but are merely a new combination of already-seen data. This phenomenon can cause the model of a complex RIA to grow extremely fast, sometimes exponentially, in proportion to the number of events in the RIA. Because of this problem, a RIA with a small set of functionalities can produce a very large state space. As a result, the crawler will not be able finish the crawl and waste its time on exploring many events in DOM-states that exhibit no new data. This is the main problem causing the current methods to fail as effective crawlers on complex RIAs.

Previously mentioned DOM-equivalence methods do not provide a complete solution for this problem. Various research works that use different DOM-equivalency methods report inability to cover the content of RIAs comprehensively without falling into state space explosion. Duda et al point to this exact problem of Cartesian state space explosion caused by independent parts in [11], using figures and examples, as an unresolved challenge. The authors of [12], which sums up model-based crawling methods also mention the problem of visiting new DOM-states with no new data. CrawlJax authors point to the problem in one of their newest papers [13]. The main contribution of this thesis is to introduce a novel method called “Component-based Crawling” to solve this problem. Component-based crawling, introduced in chapter 3, aims to cover complete content of a RIA with a model and exploration cost that can be exponentially smaller than other methods, by identifying independent parts of a RIA and taking them into account separately. Using Component-based Crawling, crawlers will be able to explore a new set of RIAs that were previously deemed too complex for complete coverage.

There are also other techniques used in the literature to improve efficiency. CrawlJax by default focuses on exploring only new events that appear on a DOM after an event execution. This approach of limiting the crawler’s attention to only a portion of the state space helps avoiding irrelevant DOM-states and finishing the crawl with a reasonable amount of data gathered. However, it does not solve the problem of covering complete content of a RIA. Moreover, often in complex RIAs a structure such as a widget frame can appear through many different event execution paths. In such case, the aforementioned approach explores all occurrences of the structure in the RIA, which can again lead to state space explosion.

Due to excessive times in crawling large-scale RIAs in these methods, it was suggested that finishing the crawl might be unreasonable in many cases and the crawler should aim to cover a reasonable amount of content when crawling is terminated midway [12], [42], [13]. The authors of CrawlJax acknowledge inability to finish crawl in a later paper [13] and focus on diversifying the crawl to obtain more results in a limited time. We present our own technique for diversifying the crawl in chapter 4. It is implemented and tested on top of the component-based crawling algorithm in our prototype crawler.

3. Component-based Crawling

In this chapter we introduce component-based crawling, our proposed method of crawling that overcomes some common problems in RIA crawling. Component-based crawling models the RIA in a different way than the state-transition model introduced in section 2.2, and achieves a significantly better efficiency.

This chapter is organized as follows: Section 3.1 describes the problem that we are going to solve, and section 3.2 presents the general overview of our solution. We first describe in detail the model of the website that the crawler builds, and then move on to describe how the crawler builds this model and makes use of it during the crawl. Section 3.3 contains detailed description of the model, and section 3.4 contains detailed discussion of the algorithm. A discussion of some challenges in our proposed method is provided in section 3.5. Experimental results and comparisons are presented separately in chapter 4.

3.1. Problem Statement

The main challenge a crawler faces is state space explosion in the RIA model that causes the crawler to take excessive time to finish the crawl. Most of the time state space explosion is caused by different mixtures of the same data, leading to new DOMs and producing a large state space for a small functionality of the RIA. In a typical RIA, it happens very often that a new DOM-state is encountered that contains no new data and is only a different combination of already-known data. The usefulness of these combinations depends on the aim of the crawl. Usually exploring these combinations is not desirable for the crawler.

Today’s complex RIA interfaces consist of many interactive parts that act independently, 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. A fairly intuitive example is widget-based RIAs, in which various combinations of contents that each widget can show creates a large volume of different DOM-states. Not all these DOMs are of interest to the crawler. A content indexing crawler, for instance, needs to visit every piece of content once and finish in a timely fashion. These rehash DOM-states only lengthen crawling while providing no new data. Figure 3 provides an example.

This issue is not just limited to widgets, but is present in any independent part in RIAs down

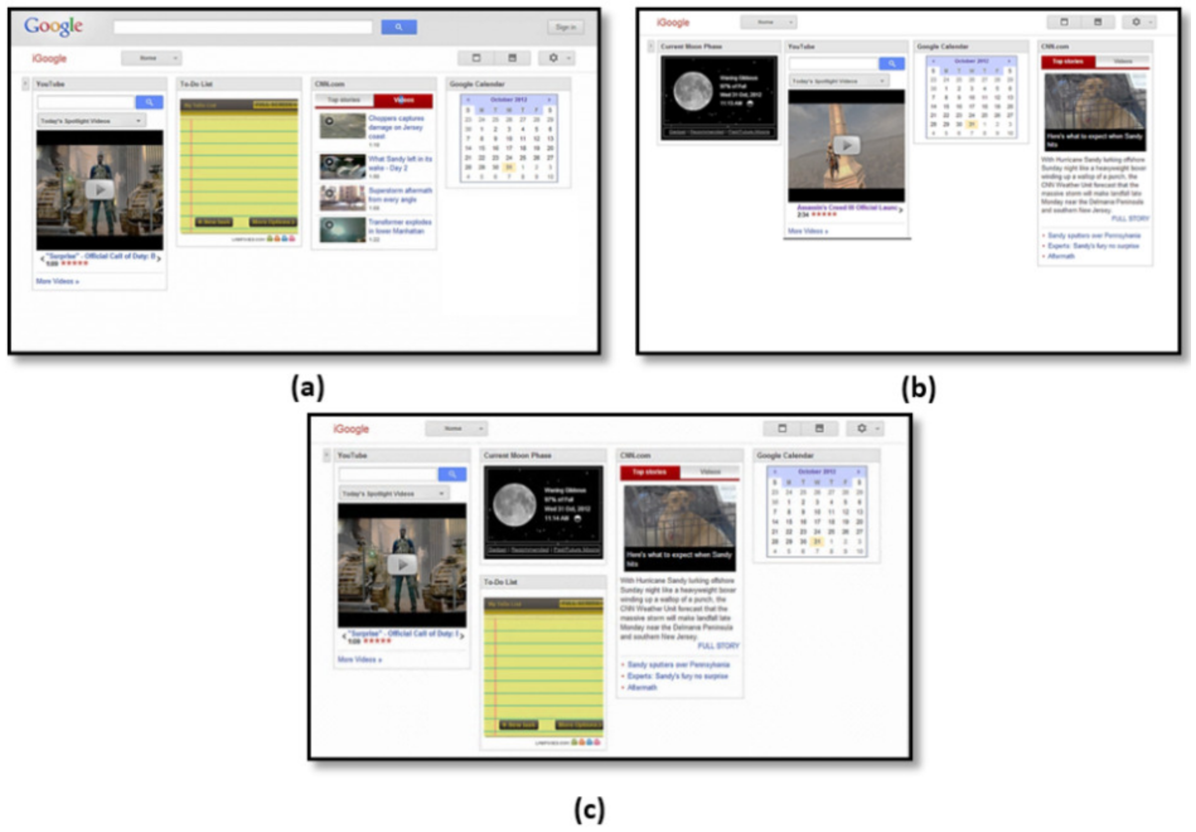


Figure 3. Example of a new DOM-state with no new data. The DOM in **(c)** is only a combination of data already present in **(b)** and **(a)**, but will have a new DOM-state in the existing methods

to every single popup or list item. Typical everyday websites such as Facebook, Gmail and Yahoo contain tens of independent parts in every snapshot. The situation is similar with any typical RIA mail client, enterprise portal or CMS. Different combinations of these independent parts lead the crawlers into crawling a lot of new DOM-states with no new data. A human user, on the other hand, is not confused by this issue since he views them as separate entities with independent behaviour, and assumes that the behaviour of one is not affected by another. In fact, the user would be surprised if the behavior of one of these parts turns out to be dependent on another.

Based on this observation, unlike the existing solutions, we decide to avoid modelling client states of RIAs considering the whole DOM of the page. We propose a novel method to crawl RIAs efficiently by modelling in terms of states of individual subtrees of the DOM that are deemed independent, which we call '***components'***. Our method detects independent components of a RIA based on difference between DOMs. By modelling at component level rather than at the entire DOM level, the crawler will be able to crawl complex RIAs exponentially faster while still covering all the content. The resulting end-model is smaller and therefore easier for humans to understand and for machines to analyze, while providing some more detailed information about the RIA that is absent from DOM level models.

In the context of detecting independent parts, static widget detection methods such as [43] have been developed. However, they are designed only to detect widgets, which are a small subset of independent entities in RIAs. Moreover, unlike our method these methods are based on a set of predefined rules, and do not adapt to individual RIAs by observing behavior of the RIA. We are not aware of any other research that handles independent parts of a RIA.

3.2. Solution Overview

Our solution is to model the RIA at a finer level in terms of meaningful subtrees of the DOM (called '***components'***) instead of modelling in terms of entire DOMs. By building a state-machine at the component level, we have a finer knowledge of how the RIA behaves, which helps in addressing the aforementioned problems and letting the crawler crawl more efficiently. The crawler can use this model regardless of its exploration strategy. Our prototype implementation uses the greedy algorithm presented in [35] as the exploration strategy, aggregated with our method to use component-states instead of DOM-states. In this section we present a brief introduction of the concept of components, how they help, and how the crawler can discover them.

Let us discuss the concept of components from the point of view of a human user, and then from the crawler's point of view, as illustrated in Figure 4. In a typical real-life RIA, each part of the page interacts with the user independently, and so the user normally thinks of these parts as separate entities. Examples of components include menu bars, draggable windows in Twitter, as well as each individual tweet, chat windows in Gmail, the notifications drop-down and mouse-over balloons in Facebook, etc. The user normally expects to be able to interact with each component independently from other components on the page.

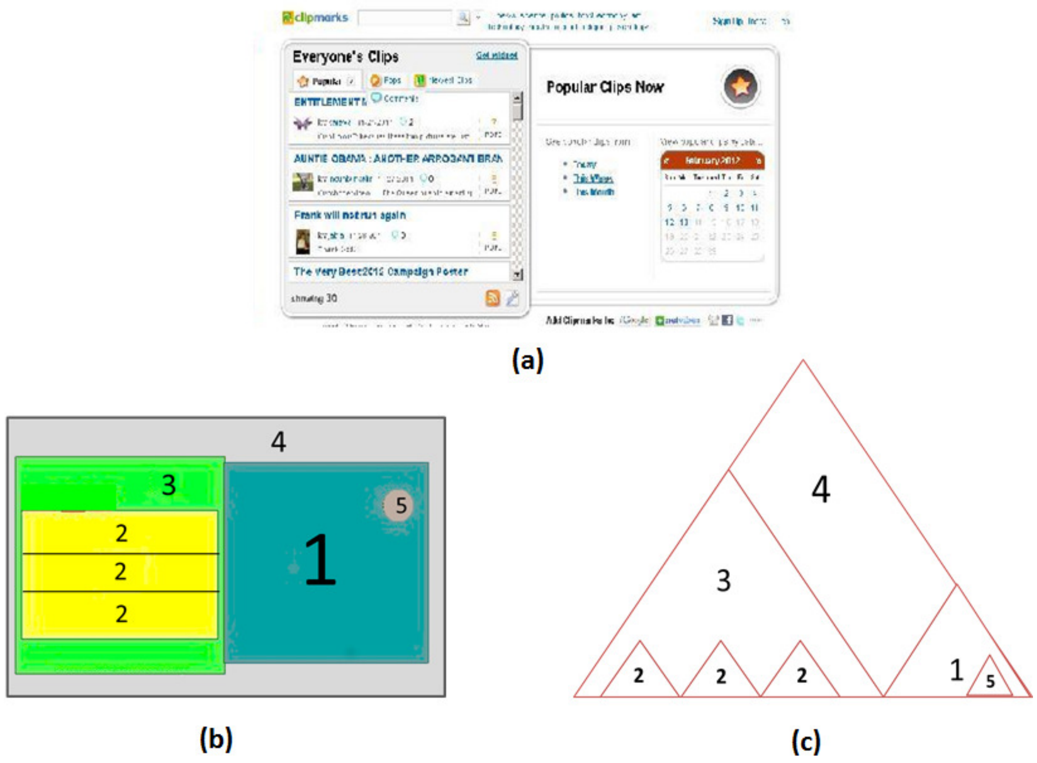


Figure 4. (a) A webpage, **(b)** components on the page the way a human user sees them as entities of the page, and **(c)** the way the crawler sees them as subtrees of the DOM.

Based on this observation, our aim is to detect these components and have the crawler to reverse-engineer the RIA by analyzing the behavior of each component independently, thus avoiding the complexity of analyzing the mixture. This assumption of independency between components is important in our method for providing full coverage. We expect this assumption to hold true in almost all real-life RIAs as it follows human user intuition. If, however, there are components on a particular RIA that affect each other, the crawler might lose coverage of some content since it does not try out all different combinations of the components. As current experimental results show, this situation rarely happens when the components are well defined.

Since components appear as subtrees in the DOM tree, we partition the DOM into multiple subtrees that are deemed independent of each other. We assign component-states to each subtree, instead of assigning a DOM-state to the entire DOM as a whole. 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 the same component. In our model, at any given time, the page that the user sees is not modelled by one DOM-state. Instead, the page is **in a set of component-states**, since it consists of different components each of which has 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.

Successfully modelling a RIA at the component level provides numerous benefits. The most obvious one is that it can avoid state space explosion caused by rehash DOMs, as depicted previously in Figure 3; since only newly seen component-states on a DOM contribute to the state space. 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 can traverse the RIA more efficiently when taking a transfer sequence, by taking fewer steps. The simpler model of the RIA will also be more easily understandable by humans and analyzable by machines.

To be able to partition a DOM into well-defined components, the crawler needs to have an algorithm for detecting components (called '***component discovery algorithm'***). Various algorithms can be suggested for component discovery. Static DOM analysis methods such as the widget-detection heuristics can be used. However, they cannot serve this purpose well since the concept of components goes well beyond only widgets or menus, making the assumptions made in such algorithms too limiting.

In order to devise a method that can be used more broadly, we propose an algorithm that builds its knowledge during the crawl through learning by observing RIA's behavior as the crawler interacts with it. The algorithm is based on DOM changes before and after execution of each event. In this approach, the crawler starts crawling with no knowledge of components. Every time an event is executed, the subtree of the DOM that has appeared/disappeared/changed is considered as a component. Our knowledge of components increases with every event execution. This method comes from the observation that if part of the webpage reacts while other parts remain still, the reacting part is probably a distinct entity by itself. For example, in a RIA webmail interface, clicking on the title of an email opens up the body of the email while other portions of the UI such as menus and chat boxes remain intact. This makes the algorithm consider the subtree of the DOM that is the container of the email body as a component, and not to mix its states with other portions of the UI thereafter. We are not aware of any other diff-based approaches for discovering independent parts.

The remainder of this chapter describes the RIA component-based model and the crawling algorithm in detail.

3.3. Model Elaboration

In this section we present the way the RIA is modelled as a multi-state-machine and how the crawler keeps track of components in its data structure, followed by a discussion on how independency of components is captured in our model in section 3.3.1. Then in section 3.3.2 we proceed with detailed description of component identifiers.

The usual way of modelling a RIA is to represent it with the state-transition model described previously in section 2.2. In the state-transition model, each DOM in the RIA is represented as a state and each event is represented as a transition. In contrast, in our model we partition each DOM into components, each of which has its own component-state. Therefore, in our model a DOM corresponds to a set of component-states..

Since events are attached to XML nodes, each event resides in one of the component-states present in the DOM (its '***owner component-state***')[2]. 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 5 illustrates how an event execution is modelled in the other methods (a) versus our method (b). 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.

[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.

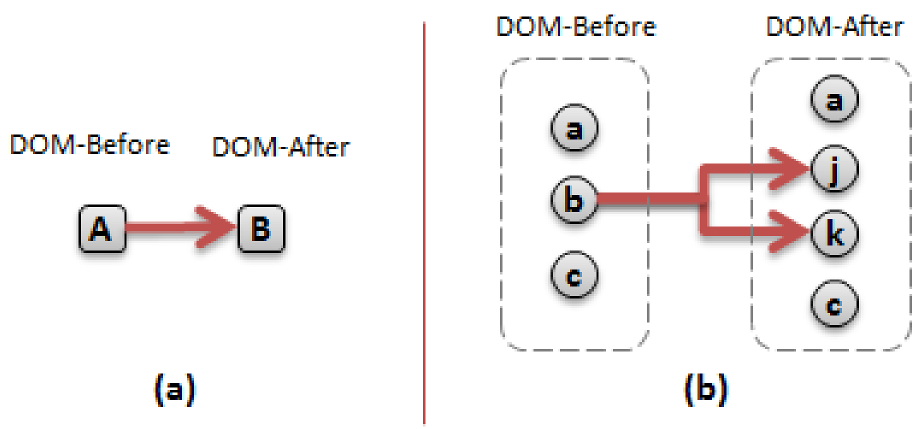


Figure 5. An event execution modelled with **(a)** DOM-states, and **(b)** component-states. Rectangles in **(b)** represent DOM-states and are not used in the actual model

The multi-state-machine can be represented as a tuple $M = (A, I, \Sigma, \delta)$ 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 δ is a function $A \times \Sigma \rightarrow 2^A$ that defines the set of valid transitions. Similar to the state-transition model (introduced in section 2.2), δ 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 lead to any number of component-states.

We can represent the multi-state-machine M as a graph $G = (V, E)$. Every state a_i in M is represented by a vertex v_i in G. We can define a bijection $R: A \rightarrow V$ between component-states in the multi-state-machine and vertices in the graph. And we model each of the multi-transitions as multiple edges with the same label:

$$\delta(a_i, e) = D \Leftrightarrow \forall a_j \in D,\ \big(R(a_i), R(a_j); e\big) \in E$$

Where $D \subseteq 2^A$ is the set of component-states that the multi-transition $\delta(a_i, e)$ leads to.

The multi-state-machine is resilient to shuffling components around in a DOM, and does not store information about exact position of the component-states in a DOM. All the multi-state-machine knows about the position of a component-state is the XPath described in section 3.3.2. Therefore, while our model is able to break a DOM into component-states (the procedure described in section 3.3.2), 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 generate an execution trace to any content in the RIA, it cannot generate an execution trace to lead to an exact DOM.

The crawler keeps information on each component-state of each component in a data structure. A simplified version of the data structure (called '***stateDictionary***') is depicted in Figure 6. It is noteworthy that the definition of a component is not bound to a specific DOM. The same component can appear as subtree in different DOMs. Components are defined based on their location. Therefore, what makes two subtrees in different DOMs to be considered as the same component is their location (not their content). In order to represent the location of a component, we use XPath with some degree of freedom (more on this in section 3.3.2). Different content can appear at that location at different times. They are regarded as various component-states of that component, and are uniquely identified with IDs, as depicted in Figure 6.

#	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='dvClipList']/ div[@class='ListItem']	<VMLCVCPQ!#$!_(~)__IKEF)_I)	...
		{:$%@)(@#*GRJPGFD{#@)(	...
		?"$#%*%@$)(!#HI!_D}{|||#R!#!	...
4	/html/body/div[@id='dvClipList']/ div[@class='ListItemSelected']	+_@#@D|?"FF>)_@#$!!S!R)$%~	...
		+{_#(!|FT$GFKJODJQ@)#URE	...
5	/html/body/ div[@id='dvContent'][@class='cnt']	><EF>F<S|!@|#%$($#%*$^RGE|	...
		|}"?<{_)O!!~~~~MLD:VCD_AS	...

Figure 6. The StateDictionary

In general, the RIA consists of a set of components, and each component has a set of possible component-states. On any given DOM, some components are present in the DOM (each in a given component-state) and some components might be absent. Using the 'Component Location' column, the crawler can look for the components present in the DOM, then it can look into any component's contents and compute an ID to match with the 'Component-State ID' column in order to look up additional info on that component-state (transitions, unexecuted events, etc.), or discover that it is a new component-state. This procedure will be elaborated in section 3.3.2 once using component locations is discussed in detail. As for the component-state ID, we use the hash of the contents of the subtree, but depending on the crawler's needs any state identifiers introduced in the related works can be used.

3.3.1. **Constraints on Component Definitions**

So far we have mentioned that components should be defined in a way that they act 'independent'. Now we can define this constraint more precisely. By 'independent' we mean "the outcome of execution of an event only depends on the component-state of its owner component". This means that the behaviour of the events in a component are independent of other present 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 execution's outcome can affect any number of components and this does not violate the constraint of independency[3]. The logic behind this definition of independency is that by examining an event only in the context of its owner component, the crawler learns the event's execution outcome, and does not need to examine it regarding other components, which is the key to our state space reduction.

It is the responsibility of the component discovery algorithm to define components in a way that satisfies this constraint in all of their component-states; otherwise the model will not

[3] This is because dependency is about factors that affect the behaviour of an event. As long as the event's behaviour only depends on its owner component-state, it can be modelled in a deterministic multi-state-machine, regardless of what changes it makes to the DOM.

represent the RIA correctly and results in loss of coverage of the RIA. The component discovery algorithm must define components properly in such a way that they are coarse enough to satisfy our assumption of independency, yet they are fine-grained enough to reduce the state space effectively.

Failure to define components coarse enough leads to the components not being independent. In such case, examining an event only in the context of its owner component is not enough to model the event's behavior accurately, since the behavior depends on other component-states as well. This can lead to incomplete coverage of the RIA contents.

On the other hand, failure to define components fine-grained enough leads to state space explosion. In the worst case, the whole DOM would be considered as one component, identified with XPath "/", and the set of component-states *A* would be the same as the set of DOM-states *S*. In such case the model essentially becomes equivalent to the DOM-state model in related works, resulting in the crawler behave as in the related works. Our proposed component discovery algorithm is described in detail in section 3.4.

3.3.2. **Component Locations**

Component locations are identified by the XPath of the subtree's root element. 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. It is notable 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 (likely to change across DOMs) to be consistent throughout the RIA and might fail to be useful in locating components. Hence, we only use '`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: to build an XPath for an element *e*

- Take the path *p* from the root of the document to *e*.
- For each HTML element in *p*, include the tag name of the element, the *id* attribute if it has one, and the *class* attribute if it has one.

Figure 6 provides some examples in the 'XPath' column.

There are two noteworthy properties that we would like to point out. First, there can be multiple instances of a component present in the DOM at the same time, each of which might or might not be in a different component-state than another. Figure 7 is an example of a shopping website. Individual list items in the product list are instances of a component

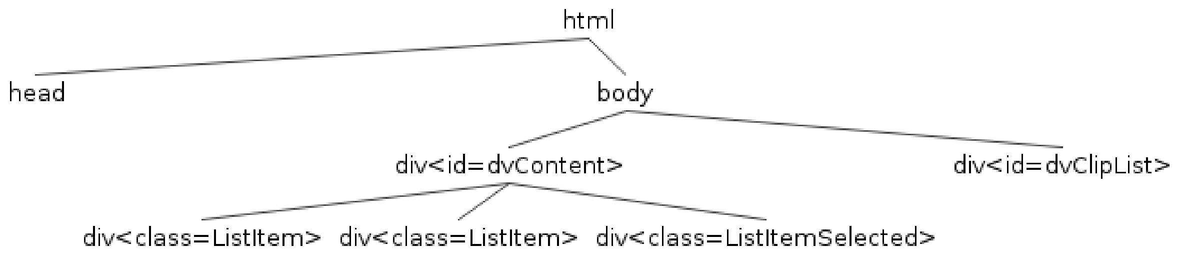


Figure 7. Part of a shopping website's DOM

‘product list item’. XPath also supports several instances of a component in a DOM at the same time, since querying an XPath can result multiple elements in the DOM. Back to the example in Figure 7, the product list items have the same XPath `/html/body/div[@id=‘dvContent’]/div[@class=‘ListItem’]` (mainly because we are excluding position predicate in XPath which is their main point of difference). But the selected item in the list yields a different XPath since it is usually assigned a different class or id attribute (`/html/body/div[@id=‘dvContent’]/div[@class=‘ListItemSelected’]`).

Another noteworthy property is that components can be nested (e.g. a widget and its window frame that has minimize/close buttons can be considered different components), just as an XPath can point to a subtree under another XPath’s subtree. More examples can be found in Figure 4 that exhibits these two properties.

Using component locations as guidelines, the crawler can partition the DOM in order to obtain set of current component-states; as described in the following pseudo-code. This procedure is used by the greedy strategy for finding our current position in the multi-state-machine, and also by the pseudo-code in section 3.4 for determining destinations of a transition.

```
Procedure determine_set_of_current_states
    For each xpath in stateDictionary
        instances_of_component ← go through the xpath and give the subtree
        For each _instance in instances_of_component
            For each known sub-path under the current xpath
                go through the sub-path and prune the subtrees
            stateID ← read_contents_and_compute_stateID (instance)
            Add the stateID to set_of_current_states
Return set_of_current_states
```

Based on our discussion in this section, we can summarize the definition of components and component-states as follows:

A component is identified by its XPath. We define a function $xpath$ such that for each node *n* in the DOM tree, $xpath(n)$ returns the XPath of *n* as defined above. If *x* is the XPath of a component, any node *n* such that $xpath(n) = x$ is the root of a subtree that holds an instance of that component. Since traversing an XPath in a DOM tree can lead to multiple nodes, there can be multiple instances of a component present in a DOM. A "component-state" is the subtree *T* under node *n*, with all other component-states inside *T* pruned. We can formalize the definition of a component-state as follows:

We define function '*subtree*' such that $subtree(r)$ returns the subtree rooted by *r*, where *r* is a node in the DOM tree. Subtrees, just like graphs, have a set of nodes and a set of edges. We define a pruning operator – on subtrees as:

$$T_1 - T_2 = T_3$$

Such that T_3 is a subtree with the same root as T_1, but with T_2 pruned from it. Therefore:

$$V_{T3} = V_{T1} \backslash V_{T2}\ , E_{T3} = E_{T1} \backslash E_{T2}$$

Where V_{Tn} is the set of nodes of the subtree T_n, and E_{Tn} is the set of edges of the subtree T_n. We can then define pruning a set of subtrees from a subtree:

$$T - \{T_1, T_2, \cdots, T_n\} = T - T_1 - T_2 - \cdots - T_n$$

We say a node *n* is inside a subtree *T* when:

$$n\ in\ T \Leftrightarrow n \in V_T$$

Suppose *X* is the set of all XPaths in the stateDictionary (the 'component-location' column in Figure 6). We want to obtain the component-state *b* such that its root is node *r*. First we find all the nodes inside $subtree(r)$ that are roots of other components:

$$G = \{n | xpath(n) \in X \land n\ in\ subtree(r) \land n \neq r\}$$

And then prune their subtrees from subtree of *r*.

$$b = subtree(r) - \{subtree(n) | n \in G\}$$

3.4. **Algorithm Elaboration**

In order to automatically explore a web application, a crawler needs to have an exploration strategy that tells it which events to execute, how to analyze the event execution outcomes, and when to stop. The method proposed in this paper only relates to analyzing event execution outcome, in order to build the model that was described in detail in section 3.3. So theoretically it can be used by any exploration strategy. The exploration strategy can then benefit from the model that is being built by our method. As mentioned earlier, we used the greedy strategy as the exploration strategy in our experimental implementation.

We now proceed to describe how the crawler populates the stateDictionary during the crawl (the component discovery algorithm). 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' list itself incrementally during the crawl as it observes the behavior of the RIA (as well as the 'Component-State ID' lists). If a pre-loaded 'Component Location' list is given, the crawler can leverage that as a fixed component locations list. However, we do not assume such a list exists at the beginning of the crawl, and the algorithm has the ability to discover Component Locations itself.

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 below:

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 (or any other traversal algorithm). For each node *x* in T_{before}, we compute the path from root to *x*, and find the node in T_{after} that has the same path. If *x* and its corresponding node in T_{after} are different, or have different number of children, *x* is considered as root of a component, its XPath is added to the stateDictionary if not already existing, and the search is discontinoued in subtree of *x*. If several such nodes exist in T_{after}, their deepest common ancestor is used as the root of the component.

From that point on, whenever the crawler encounters a new DOM, it detaches the contents of the component and considers it as a component-state; One of many component-states present in the DOM. Initially, the stateDictionary contains only one component with the XPath of “/”. More components are discovered and added to the stateDictionary as the crawling proceeds. The algorithm can be summarized as the pseudo-code below:

```
1.    Procedure ComponentBasedCrawl
2.    For (as long as crawling goes)
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 ← get_xpath (delta)
7.          If (stateDictionary does not contain xpath)
8.                add xpath to stateDictionary
9.          resulting_states ← delta.determine_set_of_current_states()
10.         For each state in resulting_states
11.               If (stateDictionary does not contain state)
12.                     add state to stateDictionary
13.         event.destinations ← resulting_states
14.   Return stateDictionary
```

In the pseudo-code above, in each iteration the crawler executes an event based on its exploration strategy (in our case, greedy) in lines 2-3. Then in line 4 it finds the changed subtree of the DOM and stores it in variable `delta`. Then in lines 5-8 it gets XPath of `delta` and adds it to the stateDictionary if not already there. This is to discover new component definitions during the crawl (Here we are updating the 'Component Location' column). Then in line 9 it runs the `determine_set_of_current_states` procedure that we introduced in section 3.3.2 on the delta. The procedure returns a set of component-states, which are added to the list of their corresponding component's states in stateDictionary, if not already there (lines 10-12). This is to populate information in stateDictionary on what component-states can each component have (Here we are updating the ‘Component-State ID’ column). Finally, in line 13 the set of resulting component-states is associated with the last executed event. This means that when modelling the RIA as a multi-state-machine, we model this event execution as a transition that ends in the resulting component-states (see Figure 5). The crawler then proceeds to pick another event based on its exploration strategy and execute it.

3.5. **Violations**

Our method makes the assumption that components are independent, and we discover new components based on diff between DOMs. Note that this component discovery algorithm has no direct correspondence to the assumption that components must be independent. Therefore there is no guarantee that the components defined by this algorithm indeed satisfy the assumption of independence. As a result, this assumption might be violated, in which case the behaviour of an event in a component may not be totally independent from other components in the DOM. Whenever the outcome of an event execution does not adhere to our deterministic model of the RIA, we say that the crawler has encountered a “***violation***”.

Occurrence of violations may or may not negatively affect the coverage. It can be the case that some components are wrongly assumed independent, and thus a certain combination of their events that could lead to new content is never explored. In this case, the crawler as missed some content.

Ideally, dependent components should be detected and merged. Merging two components into one causes the crawling method to explore all component-states of the new component (which consists of all combinations of component-states of the merged components), therefore reaching the missed content. However, this idea requires a way to detect dependent components. One solution would be that whenever the crawler encounters a violation, it should merge the component with another component based on heuristics. The heuristics guess which components might have been dependent that caused the violation.

However, dependent components are not the only source of violations. Violations can also occur if any of the general assumptions in section 2.2.1 do not hold. In such cases, merging components in the abovementioned method will not fix the problem, and violations continue to happen after merge. As a result, the method wrongly keeps merging components as it encounters violations, until all components are merged into one, in which case component-based crawling reverts to normal DOM-based crawling. Therefore, not only the abovementioned solution may not help in some cases, but also it may defeat the purpose of component-based crawling.

We acknowledge that this issue requires further investigation. Detecting and merging dependent components are costly operations and impose high overhead on the crawler. In our current implementation, occurrences of violations are simply ignored. This implementation has achieved full coverage on all of our experimental test cases, and in none of them we encountered a situation where violations cause loss of coverage. In a future work, we may address the problem of encountering violations.

3.6. **Conclusion**

In this chapter we described the method of component-based crawling in detail. Using a multi-state-machine, it is possible to model a RIA as a set of components with individual component-states, rather than DOM-states. Modelling a RIA with components captures interactions of events at a finer level, and prevents the crawler from exploring unnecessary combinations. We proposed an algorithms for efficient crawling of RIAs based on this model, starting with no knowledge of components in a RIA, discovering components during the crawl and applying the knowledge as more components are discovered.

4. Experimental Results

In this section, we compare the performance of our component-based crawling method to other methods known to be the most efficient algorithms for DOM-based crawling of RIAs with complete coverage.

The following sections are organized as follows: In section 4.1 we describe in detail what experiments are conducted, what methods are compared and on what basis they are compared, and how the results are produced and verified. In section 4.2 we present the subject RIAs that are used as test cases in the experiments. Then in section 4.3 we provide the experimental results on comparing the performance of the methods on all our test cases. In addition, on some of the test cases we are able to increase and decrease the size of the RIA by controlling the items shown. On these test cases, we perform experiments in section 4.4 to compare how different methods scale as the size of the RIA enlarges. Finally, we summarize our findings from the experimental results in section 4.5.

4.1. Experimental Setup

4.1.1. Candidate Methods

Based on previous studies on the performance of AJAX crawling algorithms [35], [12], the greedy exploration [35] method and the model-based crawling methods consistently outperform standard DFS and BFS methods. Experimental studies performed in [12] and [42] show that model-based strategies tend to show a better performance than the greedy exploration strategy, and that the probability strategy [42] tends to be the most efficient model-based crawling strategy. Therefore, we compare the performance of our component-based method against greedy exploration and probability model as two of the most efficient DOM-based crawling algorithms with complete coverage. As suggested by [42], the probability strategy is configured with initial probability set to 0.75.

As hinted in section 3.4, our implementation of component-based crawling uses the greedy algorithm as its exploration strategy. Therefore, comparing the experimental results of our component-based greedy method and the standard DOM-based greedy method enables the reader to observe the direct impact of component-based crawling on performance, without any change in the exploration strategy.

4.1.2. **Variables to Measure**

We compare the performance of the candidate methods form various different aspects. These aspects are discussed in this section:

Cost of Finishing Crawl

The first and most obvious performance determinant is the cost of finishing the crawl. We compare the time and exploration cost that each method takes for performing the full crawling procedure. (The cost metrics are elaborated in the next section). The less amount of time/cost it takes a method to finish crawling, the more usable it is; considering the fact that all methods provide the same content coverage (see section 4.1.5).

Model Size

As explained in [14], [13], the size of the generated model has a great impact on usability of the crawling results. A larger model increases the cost of analyzing or testing the model, and is harder to maintain. Therefore, the smaller the model, the more usable and maintainable it is. For this reason, we also use the size of the model generated by the crawler as another performance representative for comparing different methods.

Authors of [13] use the number of transitions in the model as an indicator of its size. We provide both the number of states and the number of transitions in our results, and use the number of transitions as the indicator for size.

4.1.3. **Cost Metrics**

To ensure credibility, the different crawling methods are compared using 2 different cost metrics. These metrics are explained below:

1- **Exploration Cost:** Firstly, we use exploration cost (weighted sum of events and resets executed, introduced in section 2.2) as a performance metric. In order to calculate the exploration cost, for each of the test cases we have measured the average event execution time $t(e)_{avg}$ and average time to perform a reset $t(r)_{avg}$. For each RIA, $t(e)_{avg}$ is calculated by executing randomly selected events, and $t(r)_{avg}$ is calculated by loading the RIAs URL multiple times, and measuring the average times for each respectively. Then, the "***reset weight***" is then defined as

$$w_r = \frac{t(r)_{avg}}{t(e)_{avg}}$$

. Then, with the simplifying assumption that all events have a weight of 1, we calculate the exploration cost as

$$n_e + n_r \times w_r$$

where n_e is the number of events executed and n_r is the number of resets performed.

2- **Time:** Exploration cost provides a better metric than simply measuring the time that it takes to perform crawling, since time measurement is affected by factors external to the crawling method. Examples of these factors are communication delays and external tasks run by the OS. However, in order to ensure that the processing overhead of component-based crawling does not affect its efficiency negatively; we also compare the methods based on time measurements.

4.1.4. **Implementation**

All the presented algorithms are implemented in a prototype of IBM® Security AppScan® Enterprise (ASE) [17], which is the same platform used in [37], [12], [38]. ASE uses web

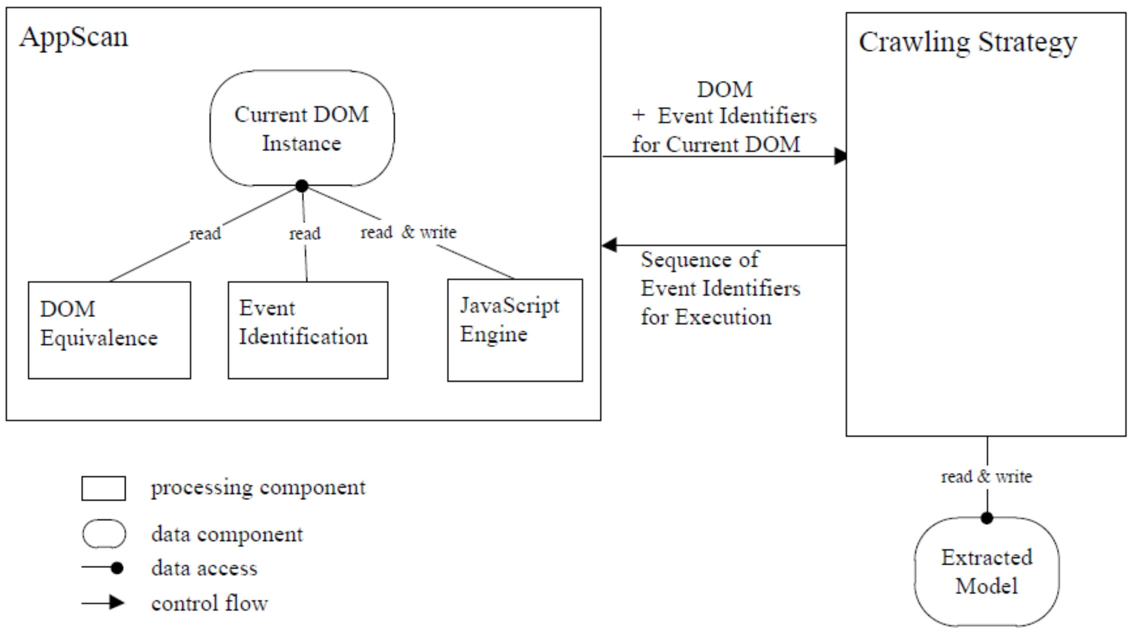


Figure 8. Architecture of our crawler (as appeared in **[42]**, with slight modifications)

crawling for the purpose of vulnerability detection and security testing on web applications. Current versions of ASE do not rely on external web browsers for providing the client side environment of the application, and instead use an embedded implementation of a browser. The embedded browser is capable of navigating to webpages, identifying elements with registered events. Our implementations of RIA crawling algorithms make use of ASE's JavaScript engine and event identification mechanism in order to execute.

For detecting equivalent DOMs in the DOM-based methods, ASE's default DOM equivalency function (outlined in [41]) is used. For detecting equivalent states in component-based crawling, the same function is used, applied to component-states instead of DOMs.

All subjects RIAs are deployed on a local server for the purpose of experiments. Targeting RIAs online on public domains is not suitable for running experiments, since crawlers should practice politeness [44] which is not to overwhelm the server with too many requests that may disrupt the normal operation of the server to serve its users. Otherwise their requests may be identified as a Denial of Service (***DoS***) attack and be dropped. Moreover, publicly available RIAs may change over time and this prevents the reproducibility of the experiments.

4.1.5. **Coverage Verification**

On all test cases, content coverage of the component-based crawling method is compared against the other methods and verified for equality. In order to do so, a database is associated with each crawling session. During the crawling session, after each event execution the crawler adds every line of the HTML representation of the DOM to the database. The database for each session therefore holds the content covered in that session. When we crawl a RIA with different methods, the databases are checked for equality to ensure none of the methods missed any content that the others covered. All methods are verified to have had equal coverage in all our experiments. This ensures that on our test cases, although component-based crawling does not visit all of the possible DOM-states, it covers all the content in the RIA. It only misses DOMs that contain no new data that the crawler had not seen already.

4.2. Test Cases

In this section we introduce the RIAs that we use as test cases in our experimental studies. Two of the websites are created and maintained by our own research group as basic test cases. The rest of examples are instances of real world RIAs deployed on our local sever for the purpose of the experiments.

Since some of these test cases are too complex for complete DOM-based crawling, previous studies that use these test cases [37], [12], [42] use a modified version of some of them. These studies exclude some of the data in the original RIA in order to reduce the state space of the RIA. Limited versions of these RIAs were used since crawling the RIA with the original set of data was impossible due to state space explosion. Component-based crawling, however, is able to crawl the full version of the RIAs. On each of the test cases, we explain the modifications performed, if any. While we use the limited version of the websites in section 4.3 to make comparison with DOM-based methods possible, we also run component-based crawling on the full version of the RIAs as part of the scalability experiments in section 4.4.

TestRIA

TestRIA, shown in Figure 9, is an example RIA maintained by our research group. It mimics a fully Ajax-based single URL E-commerce website with a three-column layout and a top menu. Users can select different menu items on the home page and the page contents fetched via Ajax interactively. Users can navigate with additional menus that appear on the left column, navigate through item catalogs, or see more details about them. Some sections include next/previous style navigation functionality.

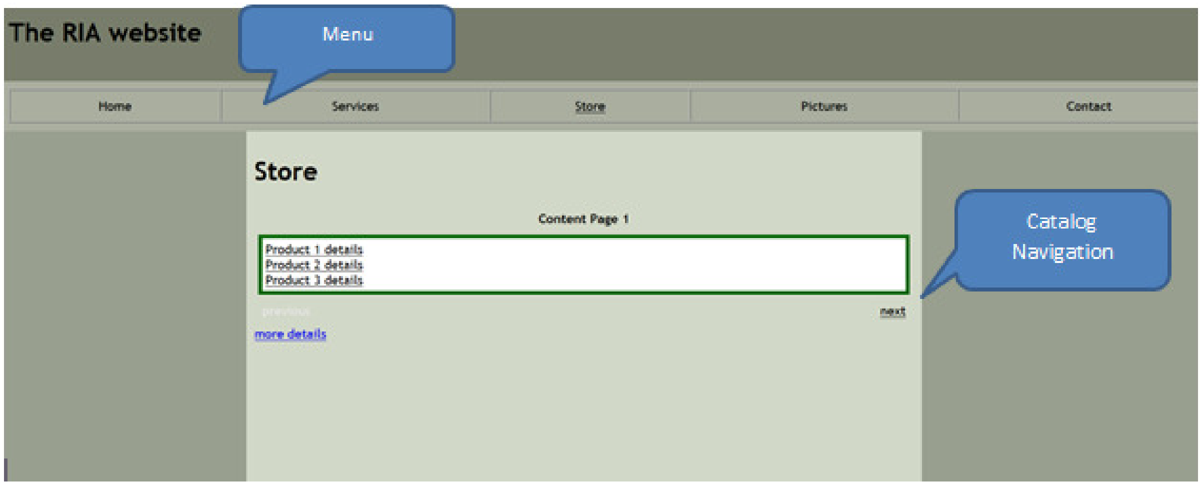
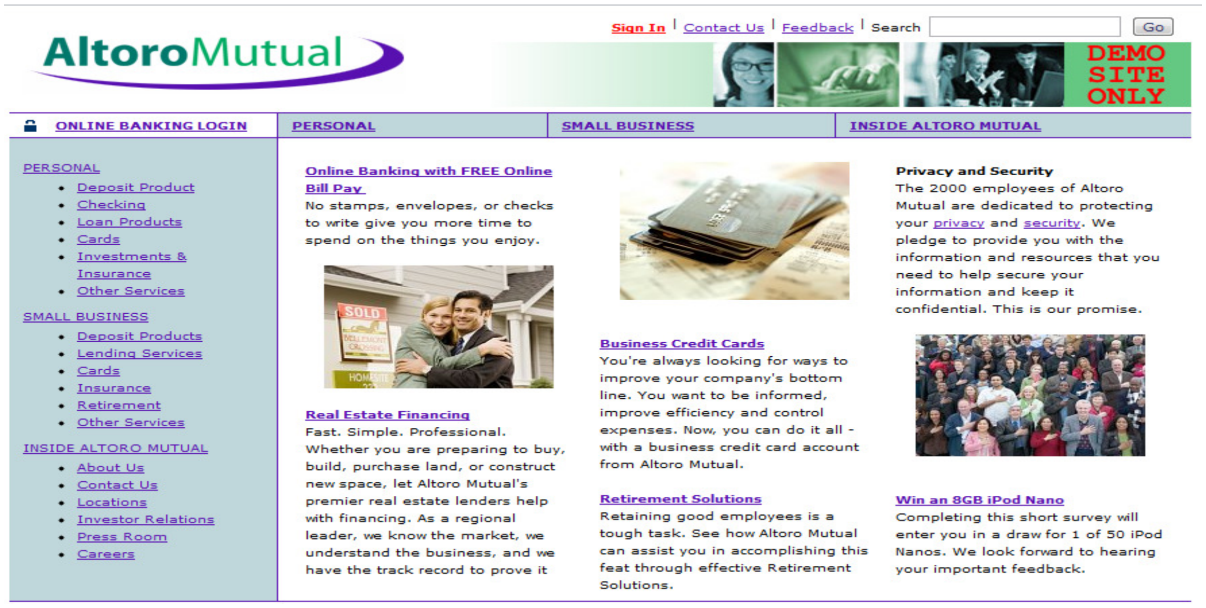


Figure 9. TestRIA

Altoro Mutual

Altoro Mutual is a demo website for a fictional bank, originally maintained by the by the IBM® AppScan® team as a mock website for security testing. The original website [45] is a traditional web application featuring hyperlinks for navigation. We have created an Ajaxified version of the website that uses AJAX calls instead of hyperlinks to fetch content. The website has no complex functionality and provides content via menu items that use AJAX.



Online Banking with FREE Online Bill Pay
No stamps, envelopes, or checks to write give you more time to spend on the things you enjoy.

Real Estate Financing
Fast. Simple. Professional. Whether you are preparing to buy, build, purchase land, or construct new space, let Altoro Mutual's premier real estate lenders help with financing. As a regional leader, we know the market, we understand the business, and we have the track record to prove it

Business Credit Cards
You're always looking for ways to improve your company's bottom line. You want to be informed, improve efficiency and control expenses. Now, you can do it all - with a business credit card account from Altoro Mutual.

Retirement Solutions
Retaining good employees is a tough task. See how Altoro Mutual can assist you in accomplishing this feat through effective Retirement Solutions.

Privacy and Security
The 2000 employees of Altoro Mutual are dedicated to protecting your privacy and security. We pledge to provide you with the information and resources that you need to help secure your information and keep it confidential. This is our promise.

Win an 8GB iPod Nano
Completing this short survey will enter you in a draw for 1 of 50 iPod Nanos. We look forward to hearing your important feedback.

Figure 10. A screenshot of Altoro Mutual

ClipMarks

CilpMarks [46] is a good example of social bookmarking websites. This AJAX-based RIA is for sharing parts of any webpage one likes with other users. The main page lists shared items in a list on the left side. Clicking on each item loads the content into the right hand side pane. The right hand side pane also provides functionality for sharing, voting up, following, etc. Each item on the left side list also has a 'pops' button, clicking on which displays a list of users voted for that item in a popup dialog.

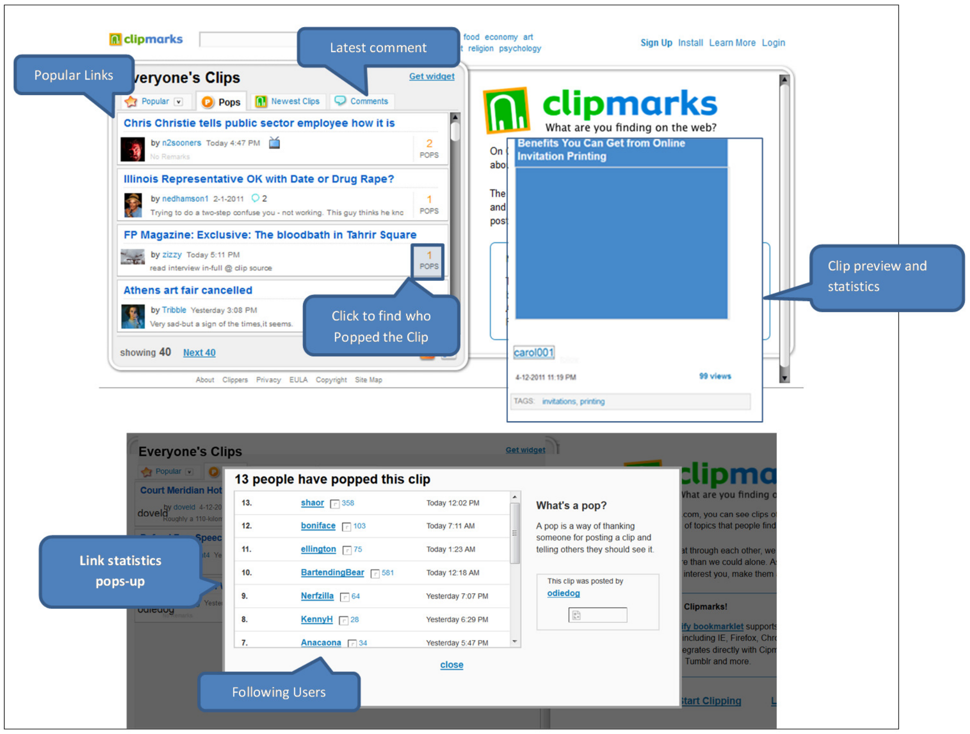
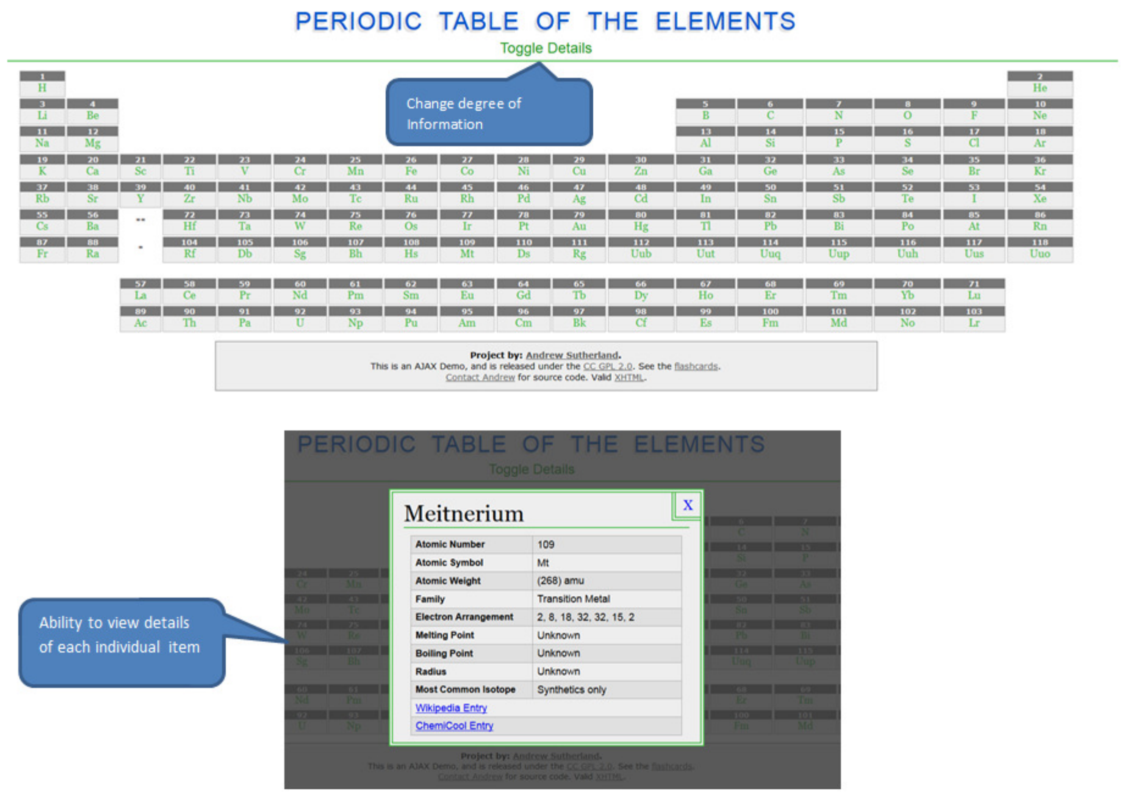


Figure 11. ClipMarks

The instance of the RIA used in the experiments contains 3 items (clips) since including more clips required excessive amount of time for experimenting with DOM-based methods. Experiment with different number of items is also conducted in section 4.4.

Periodic Table

This RIA provides a good example of a large and dense graph. The RIA [47] exhibits the periodic table that contains all the chemical elements in a table. Clicking on a chemical element displays detailed information about the chemical element in a window, while other chemical elements are still accessible. There is also an anchor at the top of each page (Toggle Details) which switches the style of the current page between two alternative styles.


PERIODIC TABLE OF THE ELEMENTS
Toggle Details
Change degree of Information
Ability to view details of each individual item
X

1 H																	2 He
3 Li	4 Be											5 B	6 C	7 N	8 O	9 F	10 Ne
11 Na	12 Mg											13 Al	14 Si	15 P	16 S	17 Cl	18 Ar
19 K	20 Ca	21 Sc	22 Ti	23 V	24 Cr	25 Mn	26 Fe	27 Co	28 Ni	29 Cu	30 Zn	31 Ga	32 Ge	33 As	34 Se	35 Br	36 Kr
37 Rb	38 Sr	39 Y	40 Zr	41 Nb	42 Mo	43 Tc	44 Ru	45 Rh	46 Pd	47 Ag	48 Cd	49 In	50 Sn	51 Sb	52 Te	53 I	54 Xe
55 Cs	56 Ba	**	72 Hf	73 Ta	74 W	75 Re	76 Os	77 Ir	78 Pt	79 Au	80 Hg	81 Tl	82 Pb	83 Bi	84 Po	85 At	86 Rn
87 Fr	88 Ra	*	104 Rf	105 Db	106 Sg	107 Bh	108 Hs	109 Mt	110 Ds	111 Rg	112 Uub	113 Uut	114 Uuq	115 Uup	116 Uuh	117 Uus	118 Uuo
		57 La	58 Ce	59 Pr	60 Nd	61 Pm	62 Sm	63 Eu	64 Gd	65 Tb	66 Dy	67 Ho	68 Er	69 Tm	70 Yb	71 Lu	
		89 Ac	90 Th	91 Pa	92 U	93 Np	94 Pu	95 Am	96 Cm	97 Bk	98 Cf	99 Es	100 Fm	101 Md	102 No	103 Lr	

Project by: Andrew Sutherland.
This is an AJAX Demo, and is released under the CC GPL 2.0. See the flashcards.
Contact Andrew for source code. Valid XHTML.

Meitnerium

Atomic Number	109
Atomic Symbol	Mt
Atomic Weight	(268) amu
Family	Transition Metal
Electron Arrangement	2, 8, 18, 32, 32, 15, 2
Melting Point	Unknown
Boiling Point	Unknown
Radius	Unknown
Most Common Isotope	Synthetics only
Wikipedia Entry	
ChemiCool Entry	

Figure 12. Periodic Table RIA

ElFinder

ElFinder [48] is an open source AJAX-based RIA for file browsing via a web Interface. The user can browse the folders by using the tree view on the left pane, selecting or double clicking files and folders on in the icon view area, and using the ‘home’ and ‘up’ buttons on the toolbar.

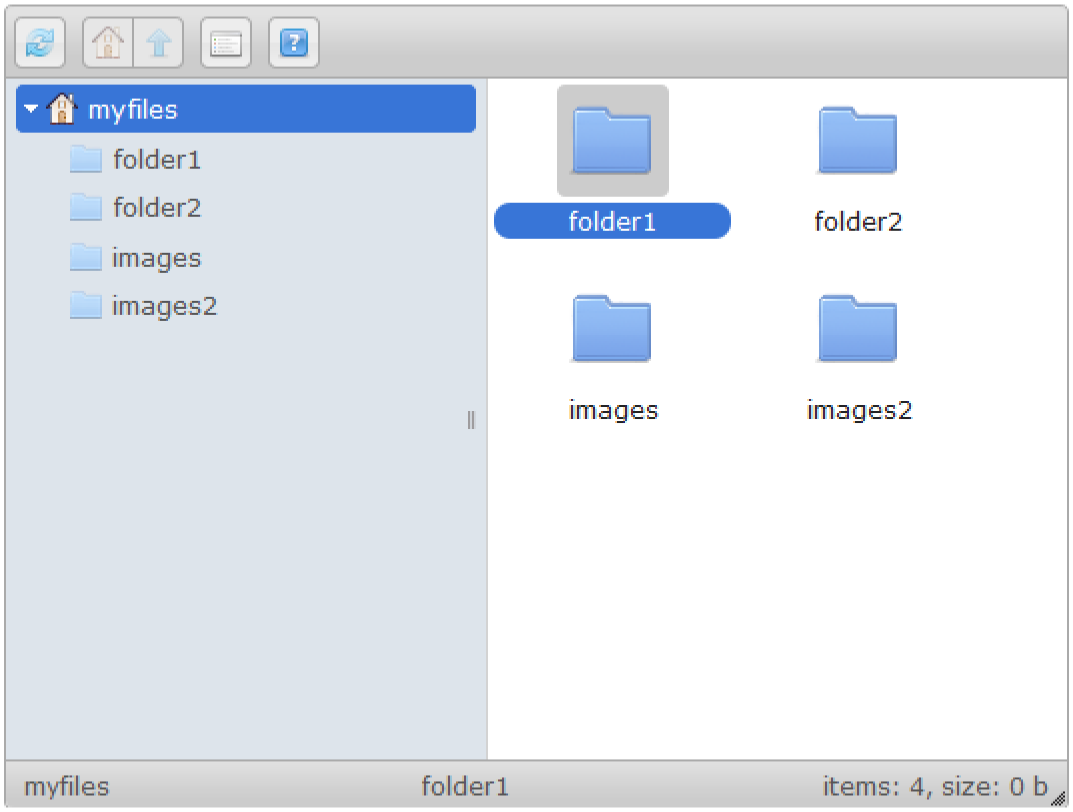
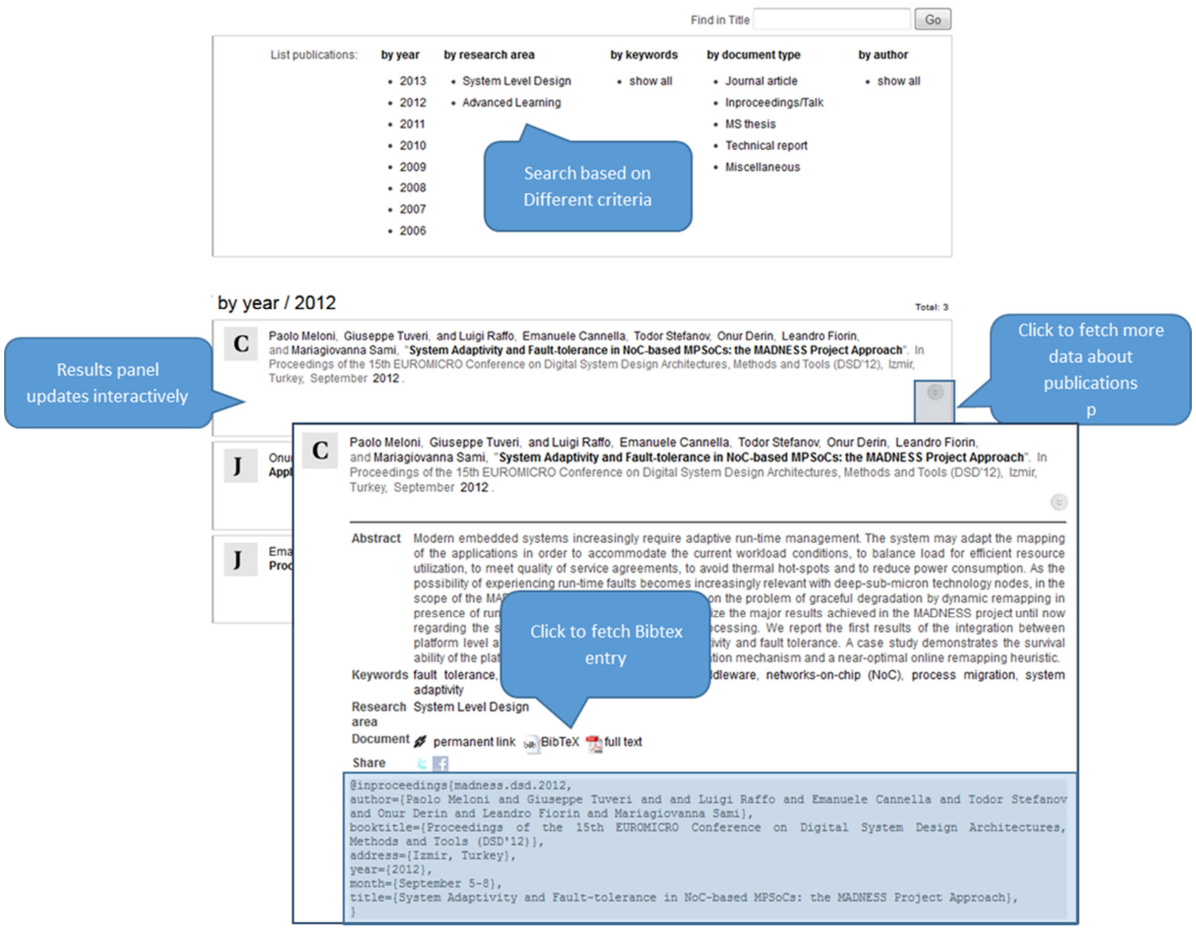


Figure 13. A snapshot of our simplified version of elFinder

In our experiments, we use a simplified version of the RIA, with some of the original functionalities that made changes to the server state of the RIA (such as rename and edit) disabled. The toolbar on the top has ‘refresh’, ‘home’, ‘up’, ‘view’, and ‘help’ buttons. The ‘view’ button toggles between icon view and details view in the main browsing area. The help button renders a floating help window with 3 tabs. In our experiments, we point the browser to a directory structure with 4 folders, 2 of which have a file inside and the other 2 have 3 files inside.

Bebop

Bebop is an open source AJAX-based interface to browse a list of publication references. The top portion of the application contains a set of events that filters the displayed references according to different categories, and at the bottom the references are listed. Each of the listed references can be in toggled between 3 different states on how much information is displayed.



C Paolo Meloni, Giuseppe Tuveri, and Luigi Raffo, Emanuele Cannella, Todor Stefanov, Onur Derin, Leandro Fiorin, and Mariagiovanna Sami, "**System Adaptivity and Fault-tolerance in NoC-based MPSoCs: the MADNESS Project Approach**". In Proceedings of the 15th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD'12), Izmir, Turkey, September **2012**.

C Paolo Meloni, Giuseppe Tuveri, and Luigi Raffo, Emanuele Cannella, Todor Stefanov, Onur Derin, Leandro Fiorin, and Mariagiovanna Sami, "**System Adaptivity and Fault-tolerance in NoC-based MPSoCs: the MADNESS Project Approach**". In Proceedings of the 15th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD'12), Izmir, Turkey, September **2012**.

Abstract Modern embedded systems increasingly require adaptive run-time management. The system may adapt the mapping of the applications in order to accommodate the current workload conditions, to balance load for efficient resource utilization, to meet quality of service agreements, to avoid thermal hot-spots and to reduce power consumption. As the possibility of experiencing run-time faults becomes increasingly relevant with deep-sub-micron technology nodes, in the scope of the MA[illegible] on the problem of graceful degradation by dynamic remapping in presence of run[illegible]ize the major results achieved in the MADNESS project until now regarding the s[illegible]cessing. We report the first results of the integration between platform level a[illegible]ivity and fault tolerance. A case study demonstrates the survival ability of the plat[illegible]tion mechanism and a near-optimal online remapping heuristic.

Keywords fault tolerance, [illegible]dleware, networks-on-chip (NoC), process migration, system adaptivity

Research area System Level Design

Document permanent link BibTeX full text

Share

```
@inproceedings{madness.dsd.2012,
author={Paolo Meloni and Giuseppe Tuveri and and Luigi Raffo and Emanuele Cannella and Todor Stefanov
and Onur Derin and Leandro Fiorin and Mariagiovanna Sami},
booktitle={Proceedings of the 15th EUROMICRO Conference on Digital System Design Architectures,
Methods and Tools (DSD'12)},
address={Izmir, Turkey},
year={2012},
month={September 5-8},
title={System Adaptivity and Fault-tolerance in NoC-based MPSoCs: the MADNESS Project Approach},
}
```

Figure 14. Bebop

Bebop is a good example of a RIA that can show a very large number of different DOMs with a very limited set of data, causing state space explosion. The original version of the RIA has 28 reference items. In our experiments, however, we use instances of Bebop with only 3 reference items loaded, to make experiments practically possible. Experiments with different number of items (including the original 28) is also conducted in section 4.4.

4.3. Comparison on subject RIAs

In this section we present the experimental results on the efficiency in covering complete content of our test cases. We compare the cost that it takes for the crawler to finish crawling of the RIA, measured in both exploration cost and time. Then in section 4.3.3 we compare the resulting models based on size.

4.3.1. Exploration Cost

Figure 15 plots the total crawling cost incurred by each of the candidate methods on each test case.

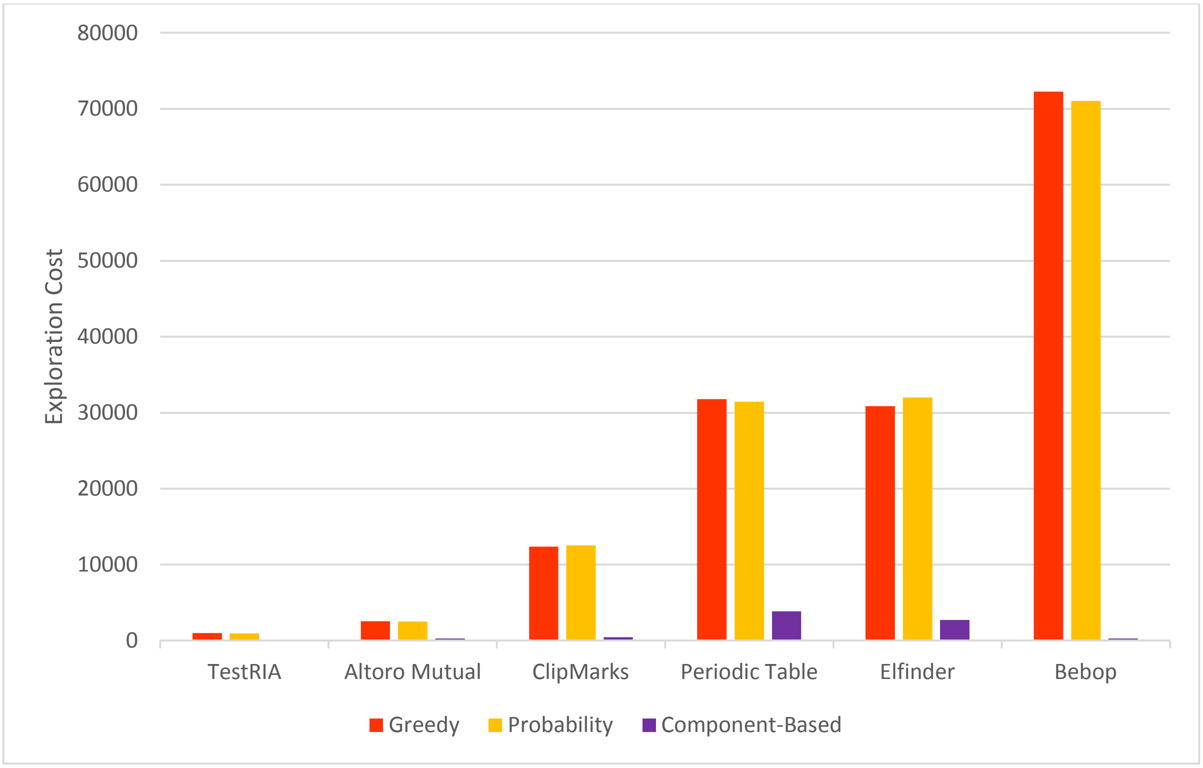


Figure 15. Comparison of exploration costs of finishing crawl for different methods

And the details are presented in the following table:

	TestRIA	Altoro Mutual	ClipMarks	Periodic Table	elFinder	Bebop
Reset Weight	2	2	18	8	10	2
Greedy	1,003	2,576	12,398	31,814	30,833	72,290
Probability	974	2,520	12,562	31,456	32,014	71,041
Component-Based	142	308	443	3,856	2,733	293

Table1. Exploration costs of finishing crawl for different methods

As seen in the figure and the table, the component-based crawling method consistently outperforms probability method and the greedy method by far. The difference between greedy and probability methods is negligible compared the difference of component-based crawling with them. As we move from simpler test cases (TestRIA and Altoro Mutual) to bigger test cases such as Periodic Table, the difference also becomes even larger.

The difference is more dramatic in RIAs that have a complex behaviour. 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 (This is studied further in the scalability tests section). Results on a Bebop instance with more items would not visually fit in the chart, therefore we used an instance with only 3 items in this section.

While component-based crawling still outperforms other methods in crawling elFinder, the performance gain is not as much as the other complex RIAs. The reason is that elFinder is file browser in which the status of the main icon view effects the behaviour of various parts of the UI such as the status bar and the toolbar. Therefore, almost the whole RIA is considered as a component by our method, and only few functionalities of the RIA are considered separate independent components.

4.3.2. Time

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.

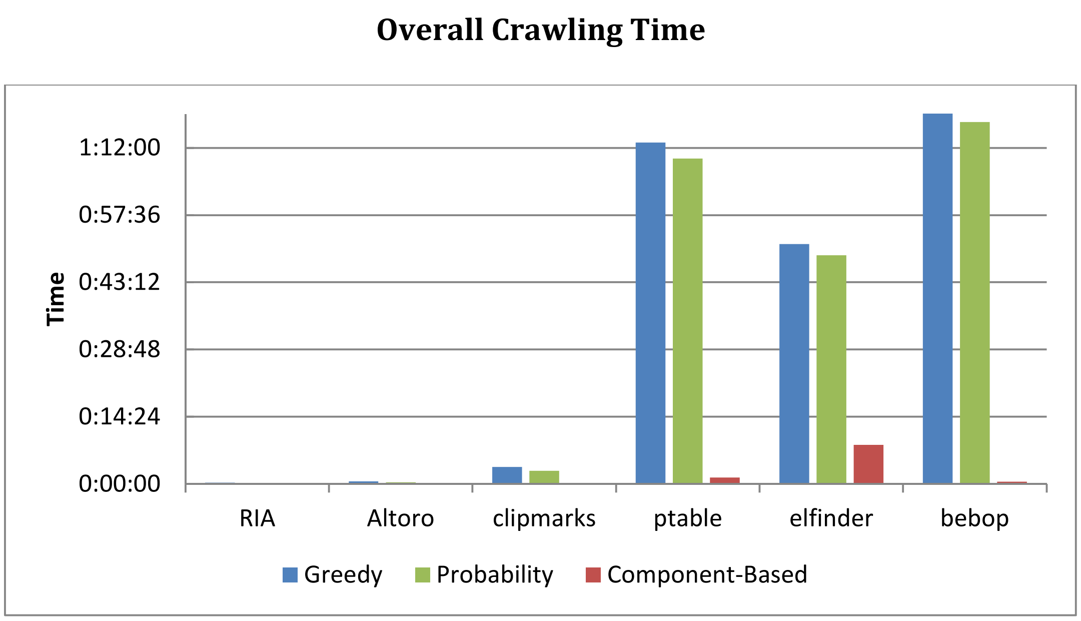


Figure 16. Comparison of time of finishing crawl for different methods

	RIA	Altoro	ClipMarks	Periodic Table	elFinder	Bebop
Greedy	0:00:18	0:00:34	0:03:38	1:13:08	0:51:22	1:25:11
Probability	0:00:11	0:00:20	0:02:50	1:09:42	0:49:00	1:17:32
Component-Based	0:00:06	0:00:04	0:00:13	0:01:21	0:08:21	0:00:29

Table 2. Time of finishing crawl for different methods

As can be seen in the above figures and tables, the performance gain of the component-based crawling method compared to the other methods measured by time is similar to when measured by exploration cost. These results verify the fact that component-based crawling incurs negligible computation overhead.

The most computationally expensive operation in our crawler implementation is the function to calculate state-ids [41], which incurs reducing and normalizing the DOM and computing hash. DOM-based crawling methods invoke this function on the entire DOM once in each step to calculate the state-id, whereas component-based crawling invokes this function many times (once for each present component-state) per step. However, since these invocations do not engage with the entire DOM and only work with small pieces of the DOM, they are performed much faster, hence making component-based crawling's several short invocations comparable to other methods' one lengthy invocation.

4.3.3. **Model Size**

In this section we compare the size of the models resulting from component-based crawling and DOM-based crawling methods. The resulting model does not depend on the exploration strategy used. Therefore, different exploration strategies compared in previous section produce the same model from a RIA when crawling is finished completely. However, when using component-based crawling we produce a different model from the same RIA since it is now modelled at component level rather than DOM level. We previously provided detailed description of the model in section 3.3. It is worth noting that although the models differ, they cover the same functionality and content from the website.

The following tables provide the number of states and number of transitions in the models obtained by DOM-based crawling and component-based crawling on each of the test cases.

	TestRIA	
	States	Transitions
Dom-Based	39	305
Component-Based	67	191

	Altoro Mutual	
	States	Transitions
Dom-Based	45	1,210
Component-Based	87	536

	Periodic Table	
	States	Transitions
Dom-Based	240	29,034
Component-Based	365	2,019

	ClipMarks	
	States	Transitions
Dom-Based	129	10,580
Component-Based	31	377

	elFinder	
	States	Transitions
Dom-Based	640	16,368
Component-Based	152	3,239

	Bebop (3 items)	
	States	Transitions
Dom-Based	285	29,284
Component-Based	119	774

	Bebop (5 items)	
	States	Transitions
Dom-Based	1,800	145,811
Component-Based	141	1,134

Table 3. Size of the obtained models using DOM-based crawling and Component-based crawling

As suggested in [13], we take the number of transitions as a metric for size of the model, since fewer number of transitions means fewer number of execution traces to be tested, which reduces the cost of testing (or any other analysis on) the model.

The results show that although modelling at component level can result in more states in some cases, it consistently provides substantially fewer transitions in all test cases[4]. Like in the previous sections, the difference becomes more significant as we move to larger test cases.

The models are verified manually for correctness with the help of a Model Visualizer tool developed in our team. The Model Visualizer can display the model, show information about any transition or state that is selected in the UI, and playback a desired event execution trace in a browser window for easy verification.

Below we present visual comparison of the model generated by each method on three of the test cases as an example. Visual comparison of more complex test cases are not included as they present large-scale or dense graphs that are not clearly understandable when printed.

[4] Notice that for component-based crawling, in some cases the number of transitions in the model in Table 3 is actually more than the number of events executed by the crawler (Table1). This is due to the fact that in component-based crawling, unlike DOM-based crawling, an event execution may be modelled with several transitions, each pertaining to one of the destination component-states that emerge as a result of the event's execution.

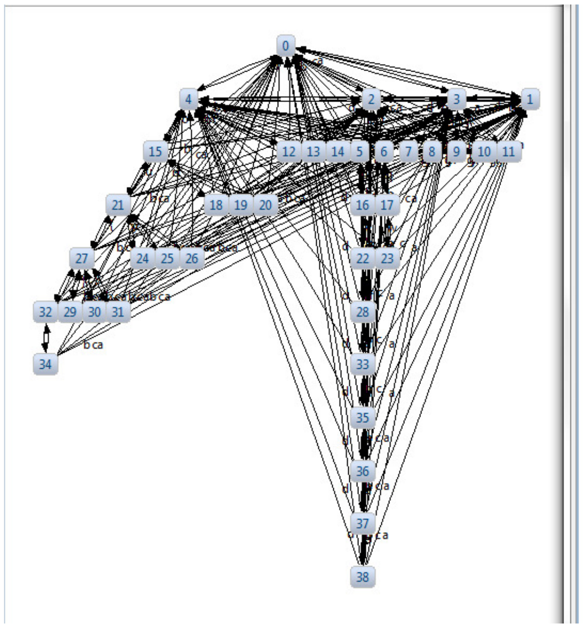


Figure 18. The TestRIA website modelled at DOM level (left) and component level (right). As seen in the figures, comopnent-based crawling results in more number of states, but a cleaner model with fewer transitions. For example, menu items are modelled as transitions from every state in the DOM level model, while they reside in their own component in the component-level model.

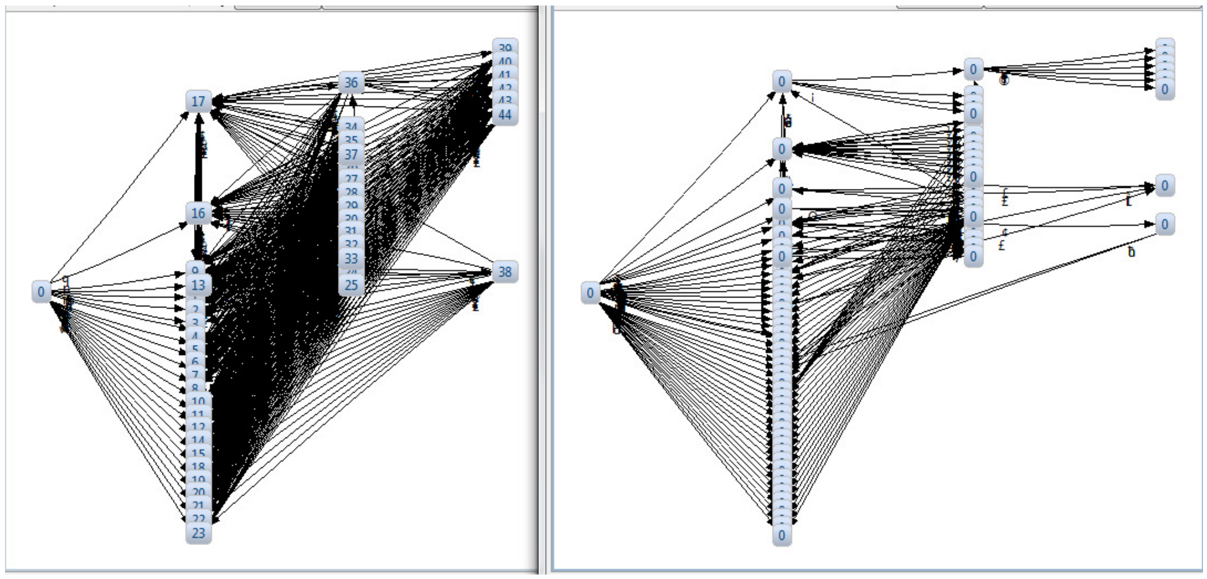


Figure 17. The Altoro Mutual website modelled at DOM level (left) and component level (right)

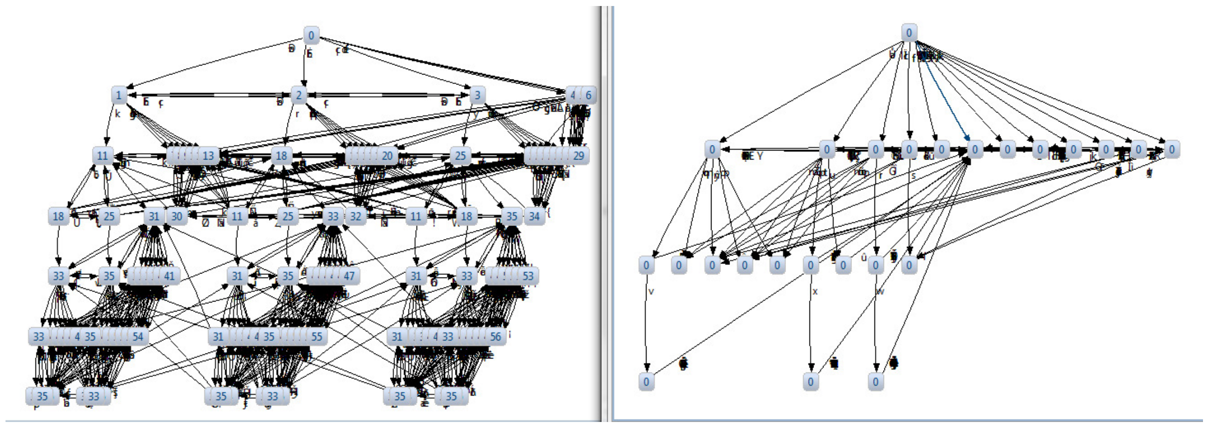


Figure 19. The ClipMarks website modelled at DOM level (left) and at component level (right). As we move to more complex test cases, the component level model looks more differently from the DOM level model. This instance of ClipMarks has 3 items. As seen in the figure, this resulted in 3 identical branches In the DOM level model.

4.4. Scalability Tests

In some of our test cases, we are able to control the size of the RIA by changing the number of data items in the source code of the RIA. In order to test the scalability of the component-based versus DOM-based methods, we conducted additional experiments on these test cases. In this section, we observe the scalability of the crawling methods by experimenting with different instances of the same RIA with different sizes.

For visual clarity of the charts, we only include the results for the greedy method and our component-based method in the charts. The results for the probability method are omitted in these charts since they would be visually indistinguishable from the results of the greedy method, as these two methods showcase near identical scaling behaviour. In the tables in this section, “N/A” refers to “not available”, were obtaining result for DOM-based crawling was impractical due to excessive running times.

ClipMarks

In ClipMarks, the initial page of the RIA shows a list of items that users have shared. Therefore by tempering the number of items in our local copy of the RIA, we can observe how a crawling algorithm scales as we add list items incrementally. Figure 20 shows the time cost of crawling different versions of the website using DOM-based and component-based greedy crawling, from 1 item to the original 40 items.

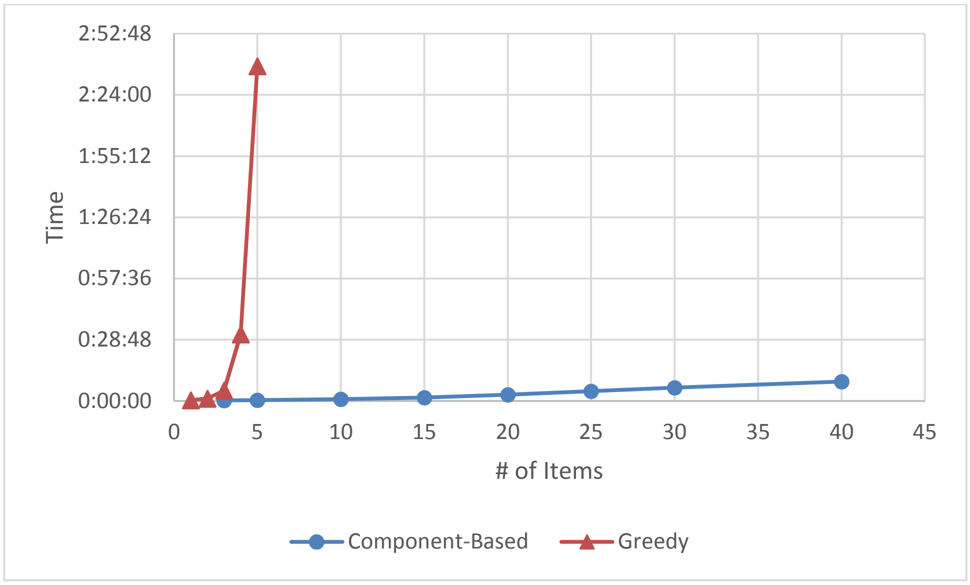


Figure 20. Time of crawling ClipMarks as the number of items in the website increase

The numbers are given below in Table4.

	1	2	3	4	5	10	15	20	25	30	40
Component-Based	0:00:06	0:00:11	0:00:16	0:00:21	0:00:26	0:00:53	0:01:39	0:02:58	0:04:40	0:06:17	0:09:07
Greedy	0:00:13	0:00:54	0:03:32	0:31:28	2:37:02	N/A	N/A	N/A	N/A	N/A	N/A
Probability	0:00:11	0:00:47	0:02:50	0:25:30	2:52:56	N/A	N/A	N/A	N/A	N/A	N/A

Table4. Time of crawling ClipMarks RIA with various numbers of items

As the results show, we can verify that component-based crawling scales nearly linearly where the DOM-based greedy method grows exponentially. This phenomenon is clearly visible in the figure. With only 5 items enabled, the running time for DOM-based methods reaches higher than 2 hours and a half, making further experiments practically infeasible. Component-based crawling, however, finishes crawling of the full version of the RIA with all 40 items enabled in less than 10 minutes.

Bebop

In Bebop also we can change the number of publications that the RIA presents. Results on crawling with different number of publications (including the original version with 28 publications) are shown in the figure below:

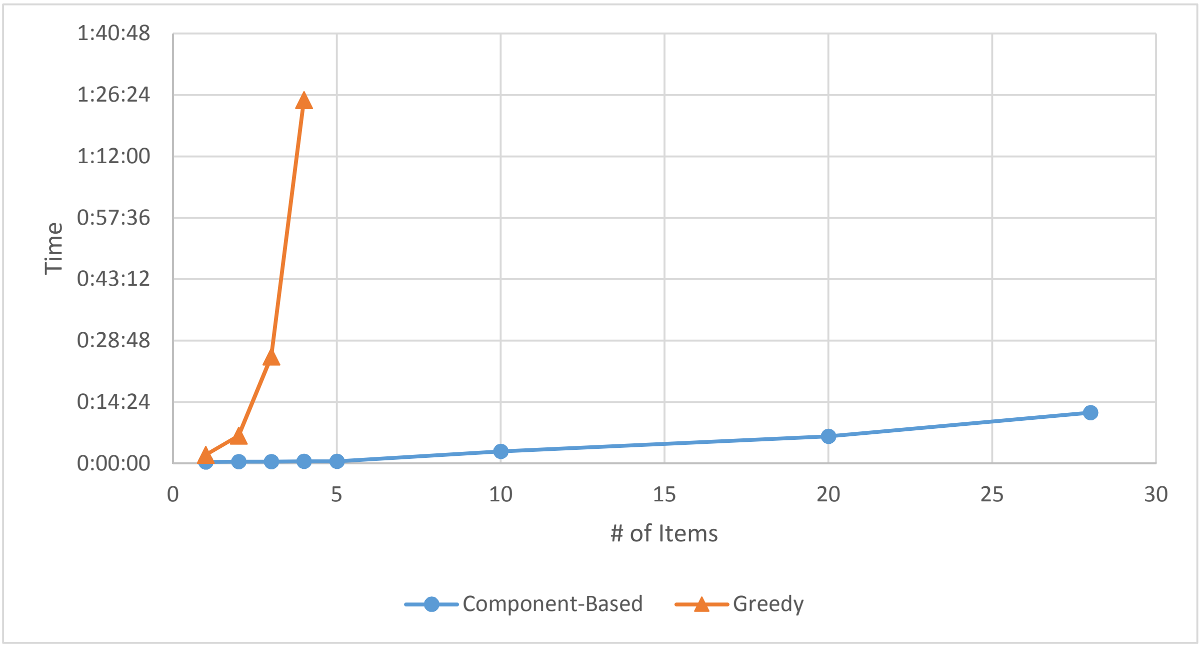


Figure 21. Time of crawling Bebop RIA as the number of items increases

And here are the numbers:

	1	2	3	4	5	10	20	28
Component-Based	0:00:19	0:00:23	0:00:25	0:00:29	0:00:31	0:02:50	0:06:21	0:11:54
Greedy	0:01:59	0:06:30	0:25:04	1:25:11	N/A	N/A	N/A	N/A

Table 5. Time of crawling Bebop RIA with various numbers of items

As with the previous example, in this example also we see that component-based crawling becomes more and more advantageous as the number of items in the RIA increases and the number of DOM-states grows exponentially. Once again, the complete RIA is only crawlable using component-based crawling.

ElFinder

We can perform a similar experiment on elFinder by changing the number of files and folders that exist in the directory structure that elFinder browses. In this set of experiments, we load that directory with a set of folders (no files in the directory root). In each folder (depth 1) there are files. One out of each 3 folders has 3 files inside, the rest of the folders have one file inside. In Figure 22, the x axis shows the total number of files (excluding folders) that exist under the directory.

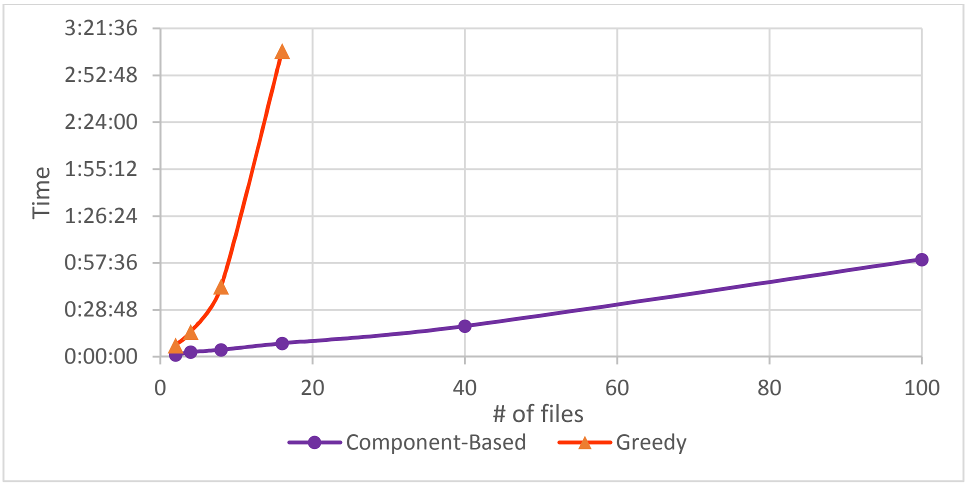


Figure 22. Time of crawling elFinder as the number of files in the RIA browser increases

The numbers are presented in the following table:

	2	**4**	**8**	**16**	**40**	**100**
Component-Based	0:01:11	0:02:57	0:04:21	0:08:18	0:18:52	0:59:51
Greedy	0:07:14	0:15:30	0:43:20	3:08:00	N/A	N/A

Table 6. Time of crawling elFinder RIA with various numbers of files to browse

As mentioned earlier, our algorithm considers most part of the elFinder as one component. As we can see in the results, component-based crawling still scales better than DOM-based greedy, although to a lesser extent compared to the previous 2 test cases.

4.5. Summary

In summary, component-based crawling shows a significantly better efficiency than DOM-based crawling methods, consistently among all the test cases. In fact, larger test cases better exhibit the performance advantage of component-based crawling. Scalability experiments show an almost-linear scalability for component-based crawling where DOM-based crawling becomes exponentially inefficient. Results based on running time are in line with the results based on exploration complexity, which confirms that the processing overhead of our component-based crawling algorithm is negligible.

The complexity of the model derived from the RIA is also significantly lower with component-based crawling, while covering the same functionality and content. This results in better analyzability and maintainability of the generated model compared to other methods.

5. Similarity Detection

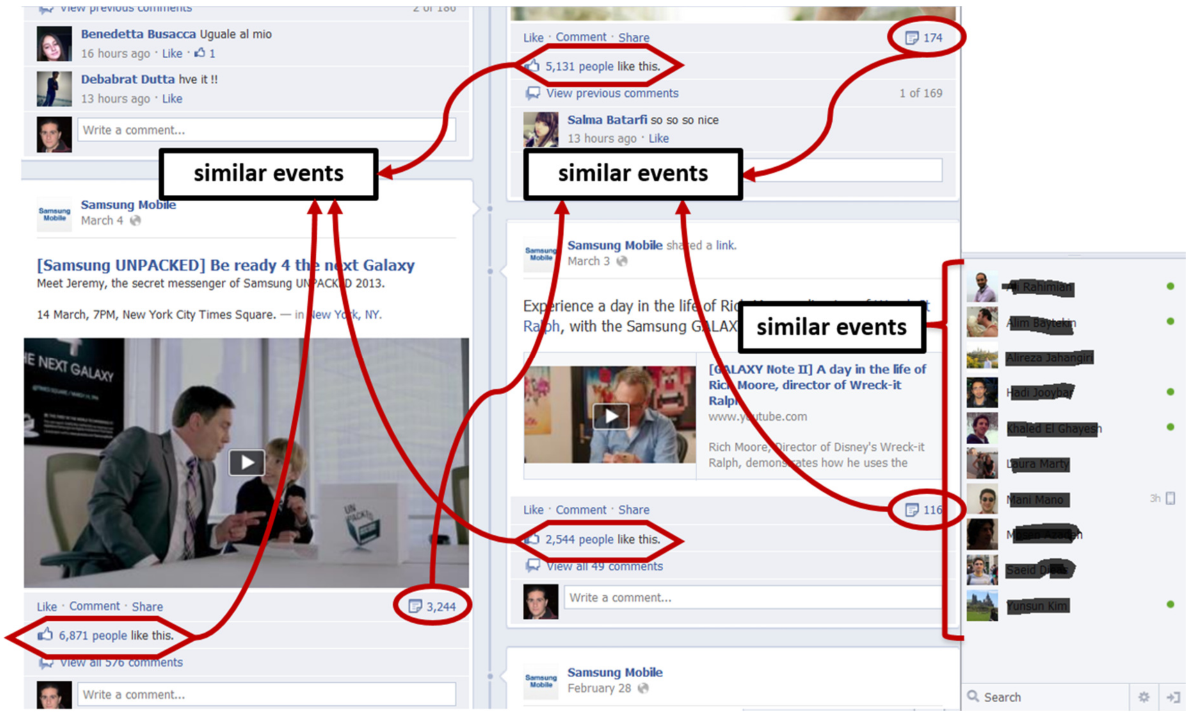
The data form the crawler is usually fed to a '***consuming system***' that analyses the data (e.g. runs security test) and produces end-results. But not all data might have equal value to the consuming system for producing end-results. In RIAs where a large amount of data is present, the crawler may spend valuable time exploring in a pool of unimportant data, while there is valuable data to be discovered elsewhere in the RIA. Therefore, there is a challenge for an automatic crawler to direct the crawl towards finding the more valuable data earlier during the crawl.

In order to address this problem, we aim to detect '***similar events***' (events that tend to produce similar data that do not contribute as much to the end-results), and give them less priority. In this chapter, we first explain the problem in detail and then present our similarity detection solution. Finally, we provide a section for the experimental results to evaluate the effectiveness of the solution.

5.1. Problem Statement

Complex websites present a challenge to automatic crawlers in finding useful results in a timely fashion. Consider websites such as shopping, news, or social websites. These websites contain an enormous amount of data organized in database. They can present very large volume of content through structurally similar UI. Crawling such large RIAs can take a very long time, even with techniques such as component-based crawling. In such cases, we can usually see a pattern of having large arrays of similar content, and it is in the interest of the crawler to limit time spent on each of them, and 'diversify' the crawl.

For example, Facebook is a large RIA, in which a typical page contains numerous posts, and each post has a 'like' button, names of people, etc. Clicking on the 'likes' link of each post in Facebook brings up a popup window with a list of people who liked the post, and hovering the mouse over each person shows a popup balloon with some details about the profile. Going through all like lists and all profile popups is a very time consuming task that may not be useful to the crawler. A crawler that does structure analysis or security scan, for example, is more interested in the structural aspects of the RIA rather than the text content. Therefore visiting one instance of a likes list or a popup balloon is enough and the crawler needs to direct the crawl towards discovering other structures in the RIA. Going



Benedetta Busacca Uguale al mio
16 hours ago · Like · 1

Debabrat Dutta hve it !!
13 hours ago · Like

Write a comment...

Samsung Mobile
March 4

[Samsung UNPACKED] Be ready 4 the next Galaxy
Meet Jeremy, the secret messenger of Samsung UNPACKED 2013.

14 March, 7PM, New York City Times Square. — in New York, NY.

Like · Comment · Share 3,244
6,871 people like this.
View all 576 comments
Write a comment...

Like · Comment · Share 174
5,131 people like this.
View previous comments 1 of 169
Salma Batarfi so so so nice
13 hours ago · Like

Samsung Mobile shared a link.
March 3

Experience a day in the life of Rich Moore, director of Wreck-it Ralph, with the Samsung GALAXY

[GALAXY Note II] A day in the life of Rich Moore, director of Wreck-it Ralph
www.youtube.com
Rich Moore, Director of Disney's Wreck-it Ralph, demonstrates how he uses the

Like · Comment · Share 116
2,544 people like this.
View all 49 comments
Write a comment...

Samsung Mobile
February 28

[redacted]

Search

Figure 23. Examples of some similar events on Facebook.com

through every item in the list only leads to redundant information in this case (if a security hole exists in a profile popup, it probably exists in all profile popups. Instead of reporting instances of the same problem repeatedly, it is better to find other problems in the application first). Other common examples include online stores, news, blogs, and emails to name a few. These websites typically contain thousands of items, each of which displays information on a single product, a single blog, a single news entry, etc.

In such environments, the crawler should be able to find the most diverse set of data as early as possible during the crawl, and leave the rest of uninteresting 'similar' data for later. This can prove helpful for various reasons.

Firstly, it might be unreasonable to assume that the results of crawling are only consumed after the crawl has finished. Crawlers such as security scanners usually pipeline the output to the consuming system as the crawling proceeds, to report the errors on-the-fly. Therefore, finding a broader set of results earlier can be useful for the user. Also, security scanners may aim for finding structure over content, and test for security only structurally different states. Therefore, having similarity will prevent them from clicking on all the 'likes' on Facebook.

Secondly, the user might even cap the crawling time, so the crawl might not proceed to the end. In the case of a content indexing crawler, for example, when the user stops the scan or views the results midway due to excessive crawling time, he/she would expect the set of results to be incomplete, but still representative of different aspects of the RIA. Diversifying the crawl in this case helps obtaining a broader birds-eye view of the RIA earlier, much like a human-guided crawl would do, rather than getting stuck in a corner of the RIA. (Without diversification we might get details on every single entry in the help menu and nothing from the rest of the RIA).

Therefore, the order in which crawler discovers content matters, and diversifying the crawl can prove helpful during a long-running crawling task. Diversifying the crawl can be achieved through detecting similar content, and directing the crawler around them.

We propose to obtain this feature in RIA crawlers by detecting events that perform similar tasks. We call these events "***similar events***". By detecting similar events prior to executing them, a web crawler can decide to skip over them or postpone their execution. Comparing events or predicting their outcome, however, is not a trivial task. Unlike URLs, the destination of an event cannot be known unless executed. The arguments of a JavaScript function call are not the only information passed to the function, and two events with the same function call can yield different results. Therefore, we use heuristics to observe and anticipate the behaviour of events for the purpose of diversifying the crawl.

Note that Similarity Detection is independent from the other concepts in this thesis, and can be used with either component-based or DOM-based crawling. Our proposed technique is also independent from the exploration strategy used (It only filters out similar events). In the next two sections we present our method to detect and deal with similarity.

5.2. **Solution Overview**

Upon visiting a page, events on that page are grouped into "***similarity classes***". If an event does not fit in any of the existing classes, a new similarity class is introduced for that event. Some events discovered later in the crawl might join the class as well. Deciding the similarity class of an event must not require execution of that event. Therefore, similarity class of an event is decided using only factors that can be observed statically from the DOM. Our heuristic method classifies events based on their code and their surrounding context in the DOM. It will be described in section 5.3. Note that the concept of a similarity class is not related to the concept of components or DOMs, and is not restricted to either. A similarity class can span different states of the RIA, so two events from different DOMs/components can be in the same similarity class.

After events on a DOM are categorized into similarity classes, the crawler proceeds to execute a few events from each class. This "***trial***" is done to ensure that events in the same class indeed yield similar outcomes, and avoid any faulty categorization by the heuristic. A good example of such a case would be menu items on many RIAs that are implemented using similar or even identical function calls. Since their code and also their surrounding context is similar, the heuristic might put them in the same similarity class. Executing two of them, however, reveals that they point to significantly different portions of the RIA.

Every class of events has a label of '***similar***', '***dissimilar***', or '***unknown***'. All classes are labelled 'unknown' upon creation. While performing the trials, a class is labelled as 'dissimilar' if at least two events inside that class have dissimilar behaviour. A class is labelled ‘similar’ if its events show similar behaviour after certain number of trials. The number of trials performed for each class (i.e. number of events executed for each class prior to labelling it ‘similar’) is an adjustable parameter. We use the minimum number ‘2’ in our experiments as the number of adequate trials. Trials are only needed for ‘unknown’ classes and need not to continue on a class it is labelled ‘similar’ or ‘dissimilar’.

It is worth noting that performing the trials can be postponed in order to not interfere with the normal exploration of the site. In fact, the crawler proceeds to execute events normally based on its exploration strategy. The event classification mechanism then observes the outcome of each event as they are executed, and adds the knowledge to its trial knowledge base. Therefore, classifying events, performing trials and labelling the classes are all transparent from the exploration strategy and have no impact on the exploration cost.

We formalize similarity classes as follows:

In this chapter, when we refer to an event e, it refers to an instance of the event in a particular context. Therefore, the set of event instances is different from Σ. We refer to the set of event instances as Σ'. For simplicity of notations in this chapter, we refer to the event instance simply as ‘event’, and refer to the context of the event as $c(e)$.

Function f is a heuristic function that, given an event, computes an ID for the event based on their code and their surrounding context in the DOM. If the computed ID of two events match, they belong to the same similarity class:

$$e_1, e_2 \in C \Leftrightarrow f(e_1) = f(e_2)$$

Where e_1 and e_2 are events, and C is a similarity class. In other words, equality between the computed ID of events is used as an equivalency function to partition the set of all events Σ' into several similarity classes.

We define another function g that after a trial is made on an event, computes a separate ID based on the execution outcome of the event. The result of the function on an event is initially undefined until a trial is made. Therefore, initially

$$\forall_e \in \Sigma'.\ g(e) = N/A$$

And all the similarity classes are labelled 'unknown'. As we perform trials, the result of function g is discovered and more event classes are labelled as 'similar' or 'dissimilar'. At any given time, the following rules hold:

$$label(C) = similar \iff \left(\forall_{e_1,e_2} \in C.\ g(e_1) = g(e_2)\right) \wedge (|\{e \in C | g(c)\ is\ defined\}| > b)$$

$$label(C) = dissimilar \iff \exists_{e_1,e_2} \in C.\ g(e_1) \neq g(e_2)$$

$$label(C) = unknown \iff label(C) \neq similar \wedge label(C) \neq dissimilar$$

Where b is the number of adequate trials. Functions f and g and their return values will be elaborated in section 5.3. As stated before, a similarity class C is independent from component-states $a \in A$ and DOM-states $s \in S$. Events in a similarity class can span multiple component-states/DOM-states.

The overall algorithm is summarized in the following pseudo-code:

```
1.   number_of_adequate_trials = 2 // user-adjustable variable
2.   for (as long as crawling goes)
3.           event ← select next event to be executed based on the exploration strategy
4.           execute (event)
5.           similarity_class ← get corresponding similarity class of (event)
6.           if (similarity_class does not exist)
7.                   create similarity_class
8.                   similarity_class.label ← 'unknown'
9.           similarity_class.trials.add(event execution outcomes)
10.          if (similarity_class.trials do not show similar outcomes)
11.                  similarity_class.label ← 'dissimilar'
12.          else if (similarity_class.trials.count = number_of_adequate_trials)
13.                  similarity_class.label ← 'similar'
14.                  for each event in similarity_class
15.                          if (event is unexecuted)
16.                                  mask(event)
17.          if (all unexecuted events in RIA are masked) // crawler finished all dissimilar events
18.                  switch ( user_setting_on_similar_events )
19.                  case (skip)
20.                          end the crawling.
21.                  case (postpone)
22.                          unmask all events in RIA
23.                          turn off similarity detection mechanism
```

For simplicity, the pseudo-code describes a rather inefficient but simplistic implementation of the method.

If a class is labelled 'unknown' or 'dissimilar', its events are executed as usual according to the exploration strategy. If a class is labelled 'similar', however, it masks its events (except those already executed) from the exploration strategy so they are not executed (lines 14-16). This is how the strategy is directed to diversify the crawl and find broader data as soon as possible. A user-adjustable variable determines whether to skip or simply postpone executing similar events. In line 17, after the crawler finishes exploring all other events in the RIA, if the variable is set to ‘skip’, the crawling session terminates (line 20). Otherwise, the masked similar events are now unmasked (lines 21-23) and the crawler proceeds to execute them based on its exploration strategy. In the former case, the crawling cam finish in substantially less time with maximum results. In the latter case, there is no positive impact on the overall running time, but postponing similar events has resulted in the crawler finding more diverse results earlier.

We believe that this method of categorizing events into classes is more effective than the methods that simply detect lists (e.g. the one used in [41]) for our intended use case, since it provides a more flexible framework for detecting, testing and handling similar events; Most importantly because it detects similar events across the entire RIA, as opposed to a single DOM. Moreover, it can deal with similar events that might be in different portions of a DOM, and doesn’t require them to be necessarily in the form of lists. The next section discusses classifying events into similarity classes in more detail.

5.3. Solution Elaboration

In order to classify events in classes, a procedure is needed that given an event, returns the similarity class it belongs to. (`get corresponding similarity class of()` procedure in the pseudo-code above). Internally, the procedure uses information about the event that can be obtained statically form the DOM, and computes a ‘***similarity ID***’ (function *f* in section 5.2). Similarity ID is then matched between events to group them into similarity classes. Events that have identical similarity IDs belong to the same similarity class. Because similarity IDs have a one-to-one correspondence to similarity classes, they are used as identifiers for similarity classes.

As hinted in the previous section, part of the similarity ID takes into consideration the event's characteristics, and part of it considers the event's surrounding context. Appending these two string forms the final similarity ID.

$$f(e) = concat(f_1(e) , f_2(c(e)))$$

Where f is the function that computes similarity ID of an event, e is the event, $c(e)$ is the context of event e, and f_1 and f_2 are functions that return strings. $f1$ is a function that takes into account the event itself and f_2 is a function that takes into account the event's context.

Moreover, a similarity criteria is needed for event execution outcomes as well, in order to compare them during the trials. As a result, event outcomes also have their own similarity IDs (function g in section 5.2). Finally, in order to evaluate the effectiveness of this whole crawl diversification mechanism, a criteria is needed for comparing crawling end-results. Each of these parts are elaborated in the following sub-sections accordingly.

Event Similarity Part 1: Event String

We observe that events that have similar behaviour tend to call the same JavaScript function, though maybe with different arguments. Moreover, they are usually attached to HTML nodes of the same HTML element type. Based on these observations, our procedure of producing a similarity ID is as follows: Write the event's owner HTML node type as an empty closed element, with all its attributes that contain a JavaScript call, replacing the arguments passed to each function call by a single integer that shows the number of arguments. Below is an example of part of a DOM that has two similar events:

```
<tr>
   <td>Chelmsford</td>
   <td>Accusation</td>
   <td>
            <a onclick="javascript:ajaxFunction('myevent',726)">Joan Waterho…</a>
   </td>
</tr>
<tr>
   <td>Spittal</td>
   <td>Accusation</td>
   <td>
            <a onclick="javascript:ajaxFunction('myevent',1521)">John Hutto…</a>
   </td>
</tr>
```

With the method described above, the event's part on similarity ID (return value of $f1(e)$) would be:

```
          <a onclick="javascript:ajaxFunction(2)" />
```

Static analysis of the function body could also be performed to provide additional data, which would make this algorithm more accurate but more computationally costly. In our implementation, however, we did not use JavaScript static analysis due to its complexity.

Classifying events solely based on the event itself is sometimes insufficient. Sometimes the context of the event also plays an important role in determining the event's outcome. Therefore, the context of events should also be taken into consideration when grouping them based on their expected similar behaviour. The following section elaborates on this matter.

Event Similarity Part 2: Context Similarity

Consider a RIA that has a list of products, and for each product it has a set of photos. Both the product list and the photo album provide pagination using next/previous buttons. Since the first set of buttons load a list of products while the other set load some picture, they have dissimilar outcomes and therefore should be put in different similarity classes. However, it can happen normally that all next/previous buttons in the RIA are implemented using the same framework and therefore exhibit similar code. If only event code is used in similarity ID, both sets of buttons would be placed in the same similarity class. Not only this is unintended, but worse, it causes inconsistent behaviour, as described by the following two scenarios:

Scenario 1: Based on the exploration strategy, the crawler executes a few trials on the 'next' button on the product list before it gets to the photo section. Because the trials show similar outcomes, the similarity class is labelled 'similar' and therefore the crawler skips executing the 'next' button on the photo section when it gets to it.

Scenario 2: Under a different exploration strategy, the crawler executes the 'next' button on the product list once, and sometime later it executes it on the photo section before getting to executing it on the product list twice. Citing the difference in outcomes, the similarity class is labelled 'dissimilar' and therefore all next/previous buttons on all paginations are executed during the crawl.

To avoid this problem, the surrounding context of an event must also be considered in generating a similarity ID, so the next/previous buttons on a product list and on a photo album are regarded as separate classes of events. In our implementation, we use an event's owner component-state as its context. A similarity ID is therefore defined on component-states. (Denoted as $f2(c(e))$)

Various methods used by different research studies for defining a state equivalency criterion can be used as similarity ID for component-states. Examples are found in [41], [32], [49], [50]. Our implementation uses a custom configuration on top of the method introduced in [41] and applies it to the XML representation of the component-state. The algorithm works as follows:

1. Any text content is disregarded
2. Algorithm finds a node whose children are all leaves in the tree.
3. Algorithm traverses the leaves and while traversing, it checks for patters of consecutive repeating elements such as `<A><B><C><A><B><C>` (A sequence like `<A><B><C><D><A><B>` in not considered such a pattern, since the repetitions of `<A><B>` are not consecutive).
4. If pattern is detected, all the repetitions are eliminated.
5. The reduced sequence is sorted and is inserted into the parent node as text content, turning the parent node into a leaf. In the abovementioned example of `<B><A><C><B><A><C>` the result would be a new leaf node `<Parent>` with text "`<A><B><C>`" (i.e. `<Parent><A><B><C></Parent>`).
6. Steps 2-5 are repeated until the XML is reduced to a single node.

7. Finally, a hash function such as MD5 is applied to the resulting XML to obtain a fixed-length string. This string is then used by our method as the similarity ID for the component-state.

By appending this string to an event's computed string from previous section, we form the event's complete similarity ID, used to classify events.

Outcome Similarity

As stated earlier, similarity classes are labelled according to comparing similarity of trial outcomes. Therefore, it is necessary to define a similarity criterion for event execution outcomes as well. Defining outcome similarity depends on how event outcomes are modelled. Since our implementation uses component-based crawling introduced in chapter 3, event executions are modelled as multi-destination transitions, in which destination states correspond to the component-states that appear as a result of the event execution. Therefore, our outcome similarity ID (function g) is "the set of component-state similarity IDs of the destination component-states".

$$g(e) = \{f_2(s) | (c(e), s, e) \in \delta\}$$

The component-state similarity ID of each individual destination is obtained using the same function f_2 introduced before. If future versions of the component-based model gather more information about event execution outcomes, that information might as well be used in the outcome similarity ID.

Result Comparison

The similarity criteria introduced so far are enough for the functionality of the mechanism. However, in order to study the effectiveness of the whole similarity detection mechanism introduced in this chapter, we need to be able to compare different crawling end-results, to see if the crawler indeed finds dissimilar results earlier by using this mechanism. Depending on the crawler's goal, crawling results are in different forms and thus different comparison criteria need to be defined accordingly. A content scanner, for example, should have a way to define similar content, whereas a crawler that scans for security entities needs to define a specification for duplicate security entities.

5.4. Experimental Results

In this section we perform experiments to study the effectiveness of Similarity Detection. The goal is to observe the rate of finding dissimilar content during the crawl, and the impact of enabling Similarity Detection mechanism on that.

All the experiments in this section are run using component-based crawling method. In these experiments, as the crawler discovers new component-states, it examines the similarity of the newly found component-state to those already found. At each step, the number of dissimilar component-states found so far is logged. We can then use the log to plot data gathering during the crawl. In the plots presented in this section, the x axis is the number of events executed thus far and the y axis is the number of dissimilar component-states found thus far. Therefore, the plots show how soon dissimilar content is found during the crawling procedure. For comparing component-states for similarity, we use the component-state similarity ID introduced in section 5.3.

Each experiment is run twice. In one, Similarity Detection mechanism is turned off and in the other, the crawler is set to postpone similar events. The plots are presented below. In all the charts, the red dotted line corresponds to execution without similarity detection, and the blue solid line corresponds to execution with skipping similar events.

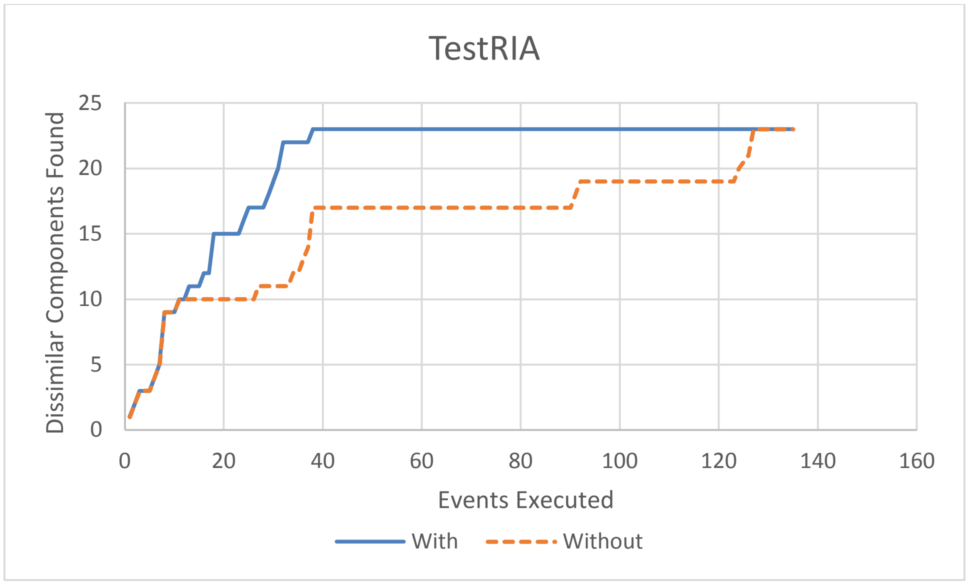


Figure 24. Finding dissimilar content during the crawl procedure in TestRIA, with and without Similarity Detection mechanism

As we can see in Figure 24, on TestRIA enabling Similarity Detection mechanism successfully helps us find dissimilar content much sooner. On TestRIA, there are paginated catalogs (with next/previous buttons) of products, pictures, and services. The content shown for each item in these categories has similar structure to the other items in the same category. Therefore, after examining 2 pages of each section, Similarity Detection postpones exploration of further paginated content to the end of crawl, resulting in us finding more diverse content first.

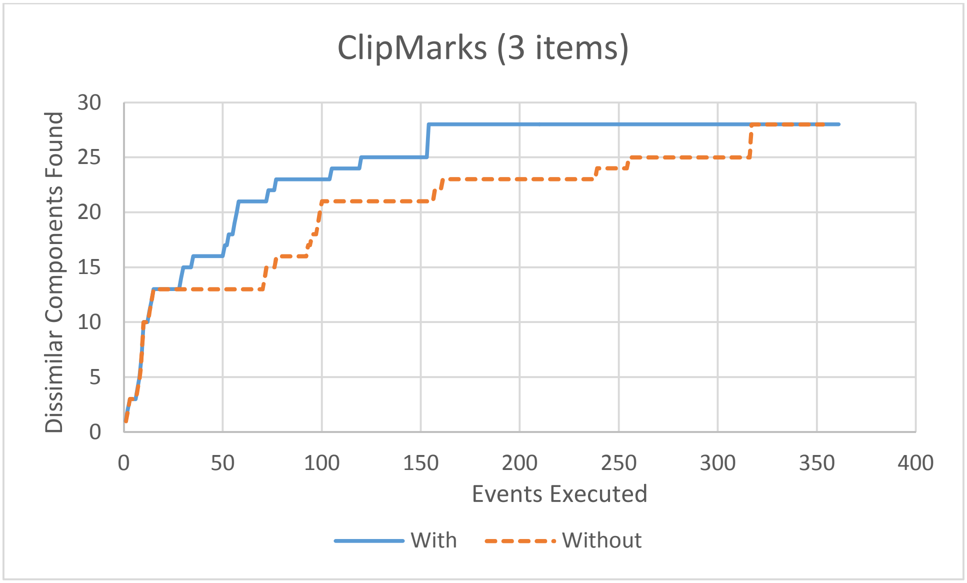


Figure 25. Finding dissimilar content during the crawl procedure in ClipMarks with 3 list items, with and without Similarity Detection mechanism

Figure 25 shows the same phenomenon in ClipMarks. ClipMarks has a list of items, and each item in the list has similar behaviour. Enabling Similarity Detection in this case also successfully results in finding more dissimilar content sooner, mainly due to the existence of the list. Based on these results, we speculated that using the full RIA (which has 40 list items in our snapshot) must showcase the effectiveness of Similarity Detection more evidently. The results for running on the full version of ClipMarks are presented in Figure 26.

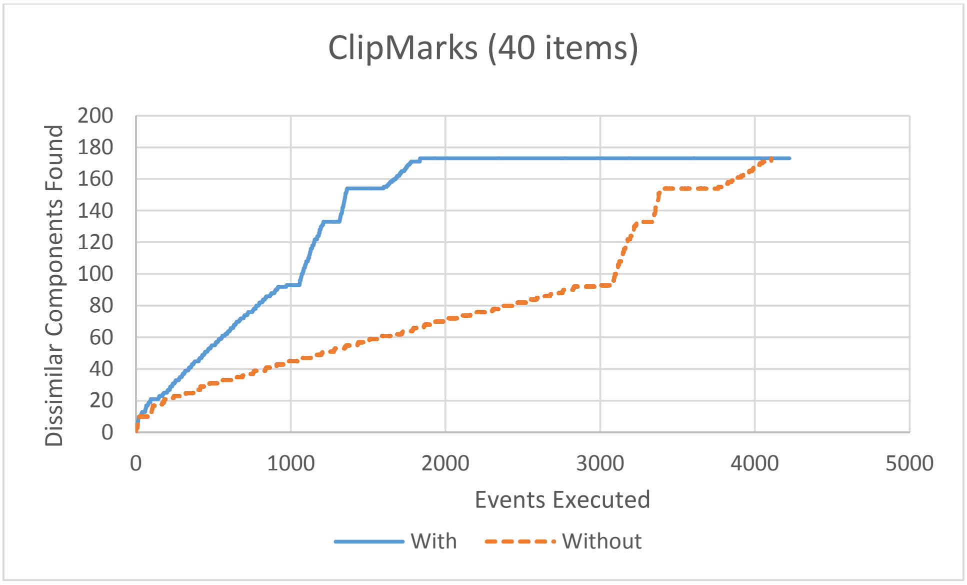


Figure 26. Finding dissimilar content during the crawl procedure in ClipMarks with 40 list items, with and without Similarity Detection mechanism

As seen in Figure 26, Similarity Detection on the full version of the RIA provides a significant benefit. The gap between the two lines in the chart increases as we increase the number of list items in the RIA.

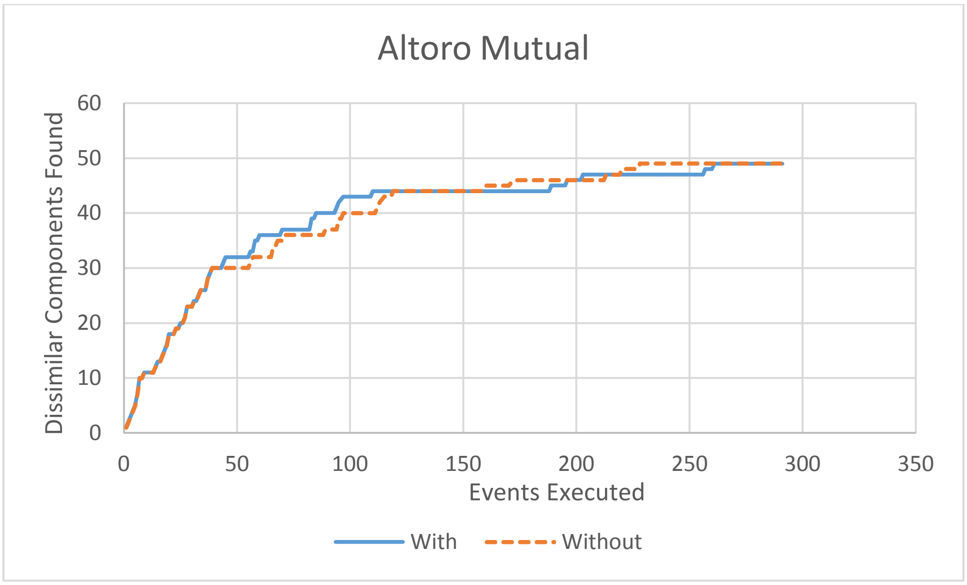


Figure 27. Finding dissimilar content during the crawl procedure in Altoro Mutual, with and without Similarity Detection mechanism

In Figure 27 we see result of experiments on Altoro Mutual. Altoro Mutual is an example of a RIA in which all DOMs look different, and there is almost no similar parts in the RIA. As a result, Similarity Detection cannot help in faster obtaining results in this RIA.

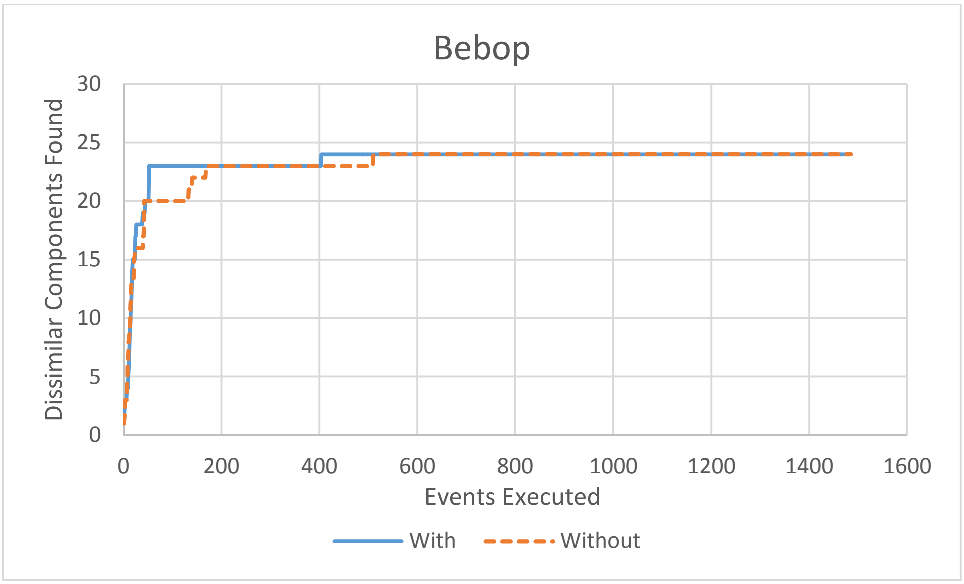


Figure 28. Finding dissimilar content during the crawl procedure in Bebop, with and without Similarity Detection mechanism

Figure 28 displays results on Bebop. In this RIA, most of the contents of the entire RIA are reachable within a few clicks from the initial DOM, and the main functionality of the RIA is to sort and filter the same data in different manners. As a result, as seen in Figure 28, most of the dissimilar contents are found soon even without the Similarity Detection mechanism. However, turning on the Similarity Detection mechanism still helps in discovering the contents even faster, although the difference may not seem as remarkable as in test cases such as ClipMarks.

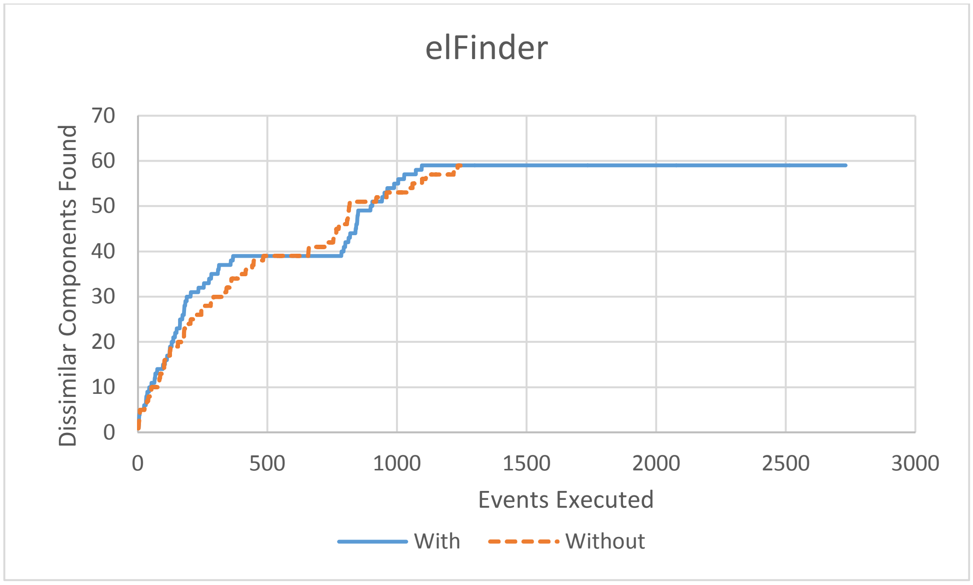


Figure 29. Finding dissimilar content during the crawl procedure in elFinder, with and without Similarity Detection mechanism

Finally, we study the effectiveness of Similarity Detection on elFinder. As we can see, our Similarity Detection in not effective on all RIAs. In this case for example, the website code is too complicated for our heuristic methods to find out similar events and classify them together. As a result, most events are considered dissimilar and no useful change is made to the order of executing events. More powerful heuristic functions for calculating similarity ID may solve this problem in future.

To sum up, Similarity Detection can prove effective in some cases, allowing for diversifying the crawl and finding dissimilar content sooner. On other test cases, however, our current scheme provides no useful change in the direction of the crawler.

5.5. Conclusion

By detecting similarity between events, a crawler can cover dissimilar portions of the RIA and produce the most diverse and comprehensive set of results in less amount of time. In this chapter, we discussed the importance of similarity detection and provided a solution. Our solution groups events into similarity classes based on heuristics, performs trials to ensure similarity of members of a class, and helps the crawler skip similar events. Skipped events can optionally be executed later.

Experimental results show that using this method can improve the speed of finding diverse contents in some cases, and make no significant difference in other cases where the RIA does not contain similar contents, or the heuristic cannot detect them.

6. Conclusions and Future Work

This thesis provides solutions for one of the most prevalent problems in the context of crawling AJAX-based RIAs: state space explosion.

The main contribution of this thesis is presenting a novel crawling method called Component-based crawling. The method solves the problem of state space explosion in complex RIAs by identifying independent portions of a RIA and modeling the RIA in terms of components rather than DOMs. Using this method, the crawler can explore complex RIAs and finish the task in significantly less running times compared to other methods, with minimal or no loss of coverage. Moreover, this method results in a much smaller model of the RIA, which in turn allows for efficient analysis and testing of the model subsequently. This document provides description of the model as well as a complete algorithm for crawling RIAs using this model. The method is fully implemented and tested on a variety of different test cases. Experimental results verify significantly better performance and scalability of component-based crawling compared to DOM-based methods. Component-based crawling opens door to crawling new web applications that were previously uncrawlable.

In addition, Similarity Detection is introduced as a technique for diversifying the crawl. This approach allows for gathering more heterogeneous sets of data earlier during the crawl procedure, which can be of special importance during long crawling sessions. The method is implemented and tested on a variety of test cases.

This work can be enhanced in several directions in possible future works. We provide a discussion of these points as the final section of this document.

The method of detecting components based on DOM diffs has no direct correspondence to the assumption that components are indeed independent. This means that dependent components can potentially exist in the model. Dependent components violate the assumptions of our crawling method, and can result in possible loss of coverage for the crawler. A future direction for this research is to develop a method to detect dependent components and merge them, in order to ensure proper coverage of the RIA.

As another direction, static analysis of the JavaScript functions can prove helpful for this crawling method. It can provide more detailed information on events without executing them, which can help in better similarity detection. Moreover, through static analysis we might be able to discover dependencies and independencies among parts of a RIA, which can greatly improve detecting independent components for component-based crawling.

In addition, more test cases are needed for a more comprehensive set of experimental results. Obtaining test cases can itself be a challenge, since available tools offer limited support for control over JavaScript execution. Therefore, deploying each new test case often requires modifications to the RIA or the tools or both, to ensure compatibility. We expect to have more experiments in the future to test the effectiveness of our proposed method.

The heuristic functions used in Similarity Detection can be enhanced further in the future to detect more types of similar content, in order to extend the applicability of this technique to father RIAs.

Finally, adapting this method for distributed environments can help distributed crawlers in using this method in an efficient way. Given the fact that components are expected to act independently, and that the lack of knowledge about nested components does not impair crawling the results, this crawling method has a good potential to be adapted for distributed crawlers.

References

[1] J. Garrett, "Adaptive Path," [Online]. Available: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications. [Accessed 24 September 2013].

[2] Adobe, [Online]. Available: http://www.adobe.com/flashplatform/. [Accessed 24 September 2013].

[3] W3C, "HTML5," 2013. [Online]. Available: http://www.w3.org/html/wg/drafts/html/CR/. [Accessed 24 September 2013].

[4] [Online]. Available: https://developers.google.com/webmasters/ajax-crawling/. [Accessed 12 September 2013].

[5] G. E. Coffmann, Z. Liu and R. R. Weber, "Optimal robot scheduling for web search engines," *Journal of Scheduling,* vol. 1, no. 1, 1998.

[6] J. Cho and H. Garcia-Molina, "Estimating frequency of change.," *ACM Transactions on Internet Technology,* vol. 3, no. 3, pp. 256-290, 2003.

[7] D. Roest, A. Mesbah and A. van Deursen, "Regression testing ajax applications: Coping

with dynamism," in *ICST*, 2010.

[8] J. Bau, E. Bursztein, D. Gupta and J. MitchellL, "State of the Art: Automated Black-Box Web Application Vulnerability Testing," *IEEE Symposium on Security and Privacy,* pp. 332-345, 2010.

[9] A. Doupe, M. Cova and G. Vigna, "Why johnny can't pentest: an analysis of black-box web vulnerability scanners," *DIMVA'10,* pp. 111-131, 2010.

[10] A. Z. Broder, M. Najork and J. L. Wiener, "Efficient URL Caching for World Wide Web Crawling," in *12th International Conference on World Wide Web*, Budapest, Hungary, 2003.

[11] C. Duda, G. Frey, D. Kossman and C. Zhou, "AJAXSearch: Crawling, Indexing and Searching Web 2.0 Applications," *VLDB,* 2008.

[12] S. Choudhary, M. E. Dincturk, S. M. Mirtaheri, A. Moosavi, G. v. Bochmann, G.-V. Jourdan and I. V. Onut, "Crawling Rich Internet Applications: the state of the art," in *CASCON 2012*, Markham, 2012.

[13] A. Milani Fard and A. Mesbah, "Feedback-directed Exploration of Web Applications to Derive Test Models," in *24th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[14] M. Harrold, R. Gupta and S. M., "A methodology for controlling the size of a test suite,"

TOSEM, pp. 270-285, 1993.

[15] I. Onut, N. Brake, P. Ionescu, D. Smith, M. Dincturk, S. Mirtaheri, G. Jourdan and G. Bochmann, "A method of identifying equivalent JavaScript events on a page". Canada Patent CA820110107.

[16] I. Onut, P. Ionescu, O. Tripp, A. Moosavi, G. Jourdan and G. Bochmann, "A method for identifying client states of a Rich Interent Application". Canada Patent CA820120275, CA920130043CA1, 28 5 2013.

[17] "IBM Security AppScan Enterprise," IBM, [Online]. Available: http://www-03.ibm.com/software/products/us/en/appscan-enterprise. [Accessed 24 September 2013].

[18] W. W. W. C. (W3C), "Document Object Model (DOM)," [Online]. Available: http://www.w3.org/DOM/. [Accessed 24 September 2013].

[19] "JavaScript," W3C: World Wide Web Consortium, [Online]. Available: http://www.w3.org/TR/REC-html40/interact/scripts.html. [Accessed 24 September 2013].

[20] [Online]. Available: http://en.wikipedia.org/wiki/AJAX. [Accessed 16 September 2013].

[21] [Online]. Available: http://en.wikipedia.org/wiki/Xpath. [Accessed 16 September 2013].

[22] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," in *27th International Conference on Very Large Data Bases*, San Francisco, 2001.

[23] L. Barbosa and F. J., "Siphoning hidden web data through keyword-based interfaces," *SBBD,* pp. 309-321, 2004.

[24] S. W. Liddle, D. W. Embley, D. T. Scott and S. H. Yaul, "Extracting Data behind Web Forms," *Lecture Notes in Computer Science,* vol. 2784, pp. 402-413, January 2003.

[25] A. Ntoulas, "Downloading textual hidden web content through keyword queries," *JCDL,* pp. 100-109, 2005.

[26] J. Lu, Y. Wang, J. Liang, J. Chen and L. J., "An Approach to Deep Web Crawling by Sampling," vol. 1, pp. 718-724, 2008.

[27] C. Duda, G. Frey, D. Kossmann, R. Matter and Chong Zhou, "AJAX Crawl: Making AJAX Applications Searchable," in *IEEE 25th International Conference on Data Engineering*, 2009.

[28] F. G., *Indexing ajax web applications,* ETH Zurich, 2007.

[29] R. Matter, *Ajax crawl: Making ajax applications searchable,* ETH Zurich, 2008.

[30] D. Amalfitano, A. Fasolino and P. Tramontana, "Reverse Engineering Finite State Machines from Rich Internet Applications," in *Proc. of 15th Working Conference on Reverse Engineering*, Washington, DC, USA, 2008.

[31] D. Amalfitano, R. Fasolino and P. Tramontana, "Rich Internet Application Testing Using Execution Trace Data," in *Proceddings of Third International Conference on Software Testing, Verification, and Validation Workshops* , Washington, DC, USA, 2010.

[32] A. Mesbah, E. Bozdag and A. v. Deursen, "Crawling AJAX by Inferring User Inferface State Changes," in *8th Int. Conf. Web Engineering, ICWE*, 2008.

[33] S. Lenselink, *Concurrent Multi-browser Crawling of Ajax-based Web Applications,* TU Delft, 2010.

[34] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of ajax user interfaces," in *ICSE*, 2009.

[35] Z. Peng, N. He, C. Jiang, Z. Li, L. Xu, Y. Li and Y. Ren, "Graph-based ajax crawl: Mining data from rich internet applications," in *International Conference on Computer Science and Electronic Engineering(ICCSEE 2012)*, 2012.

[36] K. Benjamin, G. Bochmann, M. Dincturk, G.-V. Jourdan and I. Onut, "A Strategy for Efficient Crawling of Rich Internet Applications," in *Web Engineering: 11th International Conference, ICWE*, Paphos, Cyprus, 2011.

[37] S. Choudhary, *M-Crawler: Crawling Rich Internet Applications Using Menu Meta-Model,* Ottawa: University of Ottawa, 2012.

[38] M. Dincturk, S. Choudhary, G. Bochmann, G. Jourdan, I. Onut and P. Ionescu, "A

Statistical Approach for Efficient Crawling of Rich Internet Applications," in *International Conference on Web Engineering (ICWE 2012)*, Berlin, Germany, 2012.

[39] A. Mesbah, A. van Deursen and S. S Lenselink, "Crawling AJAX-Based Web Applications through Dynamic Analysis of User Interface State Changes," *TWEB,* vol. 6, 2012.

[40] K. Benjamin, *A Strategy for Efficient Crawling of Rich Internet Applications, Master's Thesis,* University of Ottawa, 2010.

[41] K. Ayoub, H. Aly and J. Walsh, "Dom based page uniqueness indentification". Canada Patent CA2706743A1, 2010.

[42] E. Dincturk, *Model-based Crawling - An Approach to Design Efficient Crawling Strategies for Rich Internet Applications,* Ottawa: University of Ottawa, 2013.

[43] C. Bezemer, A. Mesbah and A. v. Deursen, "Automated Security Testing of Web Widget Interactions," in *Foundations of Software Engineering Symposium (FSE), ACM*, 2009.

[44] A. Heydon and M. Najork, "Mercator: A scalable extensible web crawler," *WWW,* vol. 2, pp. 219-229, 1999.

[45] "Altoro Mutual," [Online]. Available: http://altoromutual.com. [Accessed 1 October 2013].

[46] "Clipmarks," [Online]. Available: http://www.clipmarks.com/. [Accessed March 2011].

[47] "Periodic Table," [Online]. Available: http://code.jalenack.com/periodic. [Accessed 17 May 2012].

[48] "elFinder," [Online]. Available: http://elfinder.org/. [Accessed 07 October 2013].

[49] C. Duda, G. Frey, D. Kossmann, R. Matter and C. Zohu, "AJAX Crawl: Making AJAX Applications Searchable," in *IEEE 25th International Conference on Data Engineering*, 2009.

[50] D. Amalfitano, A. Fasolino and P. Tramontana, "Experimenting a Reverse Engineering Technique for Modelling the Behaviour of Rich Internet Applications," in *ICSM 2009*, Edmonton, 2009.

[51] "JQuery FileTree," [Online]. Available: http://www.abeautifulsite.net/blog/2008/03/jquery-file-tree/. [Accessed 2013].