M-Crawler: Crawling Rich Internet Applications Using Menu Meta-Model

Suryakant Choudhary

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

Web applications have come a long way both in terms of adoption to provide information and services and in terms of the technologies to develop them. With the emergence of richer and more advanced technologies such as Ajax, web applications have become more interactive, responsive and user friendly. These applications, often called Rich Internet Applications (RIAs) changed the traditional web applications in two primary ways: Dynamic manipulation of client side state and Asynchronous communication with the server.

At the same time, such techniques also introduce new challenges. Among these challenges, an important one is the difficulty of automatically crawling these new applications. Crawling is not only important for indexing the contents but also critical to web application assessment such as testing for security vulnerabilities or accessibility. Traditional crawlers are no longer sufficient for these newer technologies and crawling in RIAs is either inexistent or far from perfect.

There is a need for an efficient crawler for web applications developed using these new technologies. Further, as more and more enterprises use these new technologies to provide their services, the requirement for a better crawler becomes inevitable.

This thesis studies the problems associated with crawling RIAs. Crawling RIAs is fundamentally more difficult than crawling traditional multi-page web applications. The thesis also presents an efficient RIA crawling strategy and compares it with existing methods.

Acknowledgement

First and foremost, I am indebted to my supervisors, Dr. Guy-Vincent Jourdan and Dr. Gregor v. Bochmann for their constant support, encouragement and guidance. They have always been very amiable and approachable advisors. Their brilliance, perseverance, and pursuit of excellence will continue to inspire me for years. This thesis would not be possible without their knowledge, encouragement, and guidance.

I would also like to thank my colleague Emre Dinçtürk and the members of the Software Security Research Group at University of Ottawa, including Dr. Vio Onut (IBM) for all the insightful discussions and suggestions. I would also like to thank Seyed M. Mirtaheri for helping me with all the test applications for experimental studies. I would like to convey my regards to Bo Wan for all the help and advice.

In addition, I am grateful to IBM and the National Science and Engineering Research Council (NSERC) of Canada for all their help and support. I am also grateful to the AppScan® team at IBM.

I would like to thank my dear friends Aman Ahuja, Rushi Patel, Tarush Saul, Vaibhav Sharma, Priyambada Misra and Nikhil P.H for providing support and help all the way and making my graduate life very fulfilling and enriching.

Lastly and most importantly, I would like to express gratitude to my parents, Sahadeo Choudhary and Tarkeshwari Choudhary for their constant love, encouragement, and confidence in me and my decisions. I am grateful to my brother, Sandeep Choudhary, and sister, Priyanka Kumari for their love and encouragements. I cannot imagine completing this dissertation without the support of my family.

Table of Contents

1. Introduction 1

1.1 Web applications 1

1.2 Traditional Web Applications 2

1.3 Rich Internet Applications 3

1.4 Web Application Crawling 4

1.4.1 **Importance of Crawling** 4

1.4.2 **Crawling Strategy** 6

1.5 Crawling Rich Internet Applications 6

1.6 Motivation 7

1.7 List of Contributions 10

1.8 Organization of the Thesis 11

2. Challenges and Assumptions 12

2.1 State Definition – DOM-ID 12

2.2 DOM Equivalence 13

2.3 Event Identification 16

2.3.1 Event Equivalence 18

2.3.2 Event Execution Order 18

2.4 Deterministic Behaviour of the web application – Statelessness of the server .. 19

2.5 User Inputs 20

3. Related works 21

3.1 General RIA Crawling 21

3.2 Model-Based RIA Crawling 23

3.2.1 Overview 23

3.2.2 Crawling Phases 25

3.2.3 Hypercube crawling strategy 27

4. Crawling Strategy 29

4.1 Overview 29

5. Menu Crawling Strategy 33

5.1 Menu hypothesis 33

5.2 Overview 35

5.3 Architecture 39

5.4 Menu state exploration phase 40

5.4.1 Overview 40

5.4.2 Events categorization 42

5.4.3 Modified application graph 45

5.4.4 Menu Crawling strategy 46

5.4.5 Shortest path and Event assumptions 47

5.4.6 Violation of assumptions and adaptation of Strategies 49

5.5 Menu transition exploration phase 51

5.5.1 Overview 51

5.5.2 Graph walk 52

5.5.3 Violation and Strategy adaptation 55

5.5.3.1 Overview 55

5.5.3.2 Examples of violation instances 57

5.5.4 Walk generator algorithm 65

5.5.4.1 Overview 65

5.5.4.2 Chinese postman problem 65

5.5.4.3 Tour sequence 68

5.6 Other transition exploration heuristics 71

5.6.1 Greedy Algorithm 71

5.6.2 Rural postman problem and Travelling salesman problem 71

6. Experiments and Evaluation of Results 75

5.1 Overview 75

5.2 Experimental Setup 77

5.3 Test Applications 79

5.4 Menu state exploration results 85

5.5 Menu transition exploration results 94

5.6 Crawling strategies results evaluation 98

5.7 Transition exploration phase heuristics 99

5.7.1 Overview 99

5.7.2 Experimental Results 100

5.7.3 Transition exploration heuristics evaluation 102

7. Conclusion and Future Work 103

7.1 Summary of contributions 104

7.2 Future work 105

References 108

List of Figures

Figure 1: Web application overview (adapted from [3]) 2

Figure 2: Events registered to HTML Anchor element 17

Figure 3: Hypercube of dimension 2 and 4 [9] 27

Figure 4: Menu Category Events 33

Figure 5: Sample web applications following our hypothesis 34

Figure 6: Path from the current state to state, Snext, where the next event can be executed 38

Figure 7: Path from the current state to state, Snext, where the next event can be executed 39

Figure 8: State Exploration 41

Figure 9: Event prioritization 45

Figure 10: Graph showing virtual edges and known transitions 46

Figure 11: Event path from current state, Scurr to state with next event to execute, Snext 48

Figure 12: Violation instance 50

Figure 13: Graph showing virtual edges, known transitions and resets 53

Figure 14: Event execution violation instance (resulting in wrong state) during transition exploration phase 58

Figure 15: Event execution violation instance 1 (resulting in a new state) during transition exploration phase 60

Figure 16: Event execution violation instance 2 (resulting in a new state) during transition exploration phase 62

Figure 17: New virtual edges added to the web application graph 64

Figure 18: A graph with three connected component 66

Figure 19: Graph with strongly connected components marked [38] 67

Figure 20: Strongly connected components of a graph and augmentation edges to make the graph strongly connected 69
Figure 21: A sample directed graph 70
Figure 22: Original and Transformed graph with edge costs 74
Figure 23: Arc cost calculation 74
Figure 24: Clipmarks website 79
Figure 25: Periodic Table Website 81
Figure 26: Test RIA Website 82
Figure 27: Altoro Mutual website 83
Figure 28: Hypercube10D website 84
Figure 30: State exploration cost for Clipmarks web application 86
Figure 31: State exploration cost for Periodic Table web application (Logarithmic Scale) 87
Figure 32: State exploration cost for Periodic Table web application 88
Figure 33: State exploration cost for TestRIA web application (Logarithmic scale) 89
Figure 35: State exploration cost for Altoro Mutual web application (Logarithmic scale) 90
Figure 37: State exploration cost for Hypercube10D web application (Logarithmic scale) 92
Figure 38: State exploration cost for Hypercube10D web application 92
Figure 39: State exploration statistics for each crawling strategies 93
Figure 40: Total cost to crawl Clipmarks web application 94
Figure 42: Total cost to crawl TestRIA web application 95
Figure 43: Total cost to crawl Altoro Mutual web application 96
Figure 44: Total cost to crawl Hypercube10D web application 96
Figure 45: Transition exploration statistics for each crawling strategies 97
Figure 46: Total cost to crawl Clipmarks web application 100

Figure 48: Total cost to crawl TestRIA web application 101

Figure 49: Total cost to crawl Altoro Mutual web application 101

Figure 50: Lucid Desktop [15]screenshot 106

1. Introduction

The World Wide Web or simply the web is a global communication channel facilitating communication and collaboration among users to share information via computers connected over the Internet. The introduction of the web has had a major impact on how information and services are offered and accessed over the internet. Today the Internet has revolutionized the way we live and share information to such an extent that almost all phases of our life are in one way or another connected and controlled by the web. The web has become such an integral part of our lives that the disruption of any of these services can have severe impact on the day-to-day workings of modern day civilization.

1.1 Web applications

Over the past decade, the web has also evolved in significant ways from a system that used to deliver static content in the form of static web pages to a system that now supports distributed applications. These applications are commonly known as web applications. These applications are computer software developed using browser-supported programming languages and technologies such as javascript [1], HTML etc. and are accessible by web browsers.

Web applications have become one of the most extensive technologies to provide information and services in today's world. The popularity of web applications can be attributed to several factors, such as availability, reachability, cross-platform compatibility, fast development etc. One important reason that has contributed to the success of web applications is the ubiquity of web browsers, and the convenience of using a web browser as a client, sometimes called a thin client. In addition, web

applications provide the convenience of making the application available instantly to all the users without actually requiring them to be installed on potentially thousands or millions of client computers. The advent of newer and richer technologies such as Ajax (Asynchronous JavaScript and XML) [2] has further enhanced the experience of the web applications with better interactiveness and responsiveness.

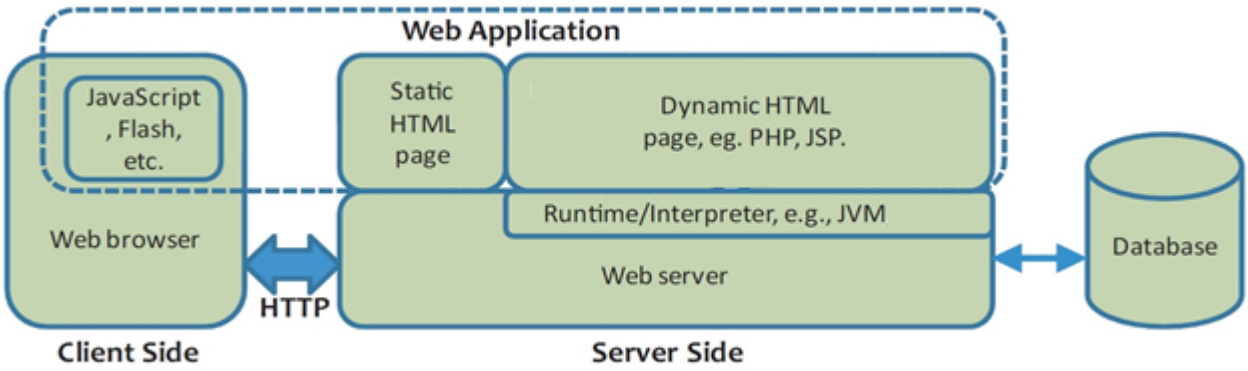

Figure 1: Web application overview (adapted from [3])

1.2 Traditional Web Applications

Initially, web applications were simple HTML pages on the client side. The complete application logic is present at the server end and the clients behave as placeholders for the contents. Each of these web pages had a unique Uniform Resource Locator (URL) to access it. Any access in a traditional web application starts with the user or client (web browser) submitting the request for these web pages identified by URLs to the web server. The web server in turn generates the requested web page as a response. The client then entirely replaces the previous content with the response

received from the web server. Traditional web applications contain different sets of documents and static HTML pages to provide all the information or functionality to the user. In addition, each web page will have information to fetch more web pages, embedded in the form of HTML links, which are essentially URLs of other web pages. To access other pages, the client simply needs to use the URL corresponding to that page and send the request to the server. One major drawback of this technique is that all the communication between the client and server are synchronous. The client waits for the server response before performing an activity or allowing the user to access the contents. This results in user activity being suspended until the new page is loaded.

1.3 Rich Internet Applications

Web applications have been evolving extraordinarily fast with new programming models and technologies, resulting in major changes on how the web applications are designed and developed. In recent years, Rich Internet Applications (RIAs) or Web 2.0 have become the new trend for web applications defying the notion of web applications running exclusively on the server side and client as placeholders. With the introduction of newer and richer technologies for web application development, web-applications have become much more useable and interactive. One of the most important ones is the migration of server-side application logic to client-side scripting—most often in the form of Javascript. This load sharing of functionality between client and server was further made more seamless with the development of technologies such as Ajax, which form foundational component of RIA and the next-generation of Web 2.0 applications. Although fundamentally still being a web application, these applications represent a paradigm shift in how Web applications are designed and developed. RIAs changed the traditional web applications in two important aspects: First, client–side scripting languages such as javascript have allowed the dynamic modification of the

web page by updating the Document Object Model (DOM) [4], which represents the client-side "state" of the application, sometimes without any communication with the server all together. Second, using technologies like Ajax the client can communicate asynchronously with the server, without having the user to wait for the response from the server. Hence, in RIAs we can reach to a new client-side state (web page) without requiring a new URL load but by executing one or more events. In addition, in both the cases the URL of the web page typically does not change during these client side activities. Consequently, we can now have a quite complex web application addressed by a single URL. The notion of one URL per web page or client side "state" is lost with these new web applications.

1.4 Web Application Crawling

Crawling is the process of browsing a web application in a methodical and automated manner. The result of crawling is the "Model" of the application which contains information about all the discovered web pages, information about reaching these web pages from the initial web page (web page corresponding to the initial URL of the web application where the crawling process started) and also from one web page to another.

1.4.1 Importance of Crawling

Crawling is an important reason for the popularity and usefullness of the web. An important functionality of the web in general is the information it provides. Rather the web started as a platform for inforamtion sharing and availability. This information can only be made available if the different information sources can be discovered and their content made available for access. This is

mostly done by the search engines such as Google, Yahoo, Bing etc. These search engines depend upon web crawlers to discover different web pages over the internet and index their content for access. Web crawlers are computer programs which browses World Wide Web in an automated manner. If search engines are not able to crawl websites with information, they will not be able to index them. Hence one of the main strenghts of the web will be lost.

In addition, crawling is also required for any thorough automated analysis of the web application such as for security and accessibility testing. As web applications are increasingly used to deliver security critical services, they become a valuable target for security attacks. Many web applications are used to provide critical services such as banking transactions and often interact with back-end database system, which may store sensitive information such as financial, health etc. A breach in the web application security would result in unauthorized access to sensitive information leading to severe economic losses and legal consequences. A security report by Verizon [5] shows that web applications now rank among the top both in number of security breach incidents, and amount of data compromised. In addition, a high percentage of web applications deployed on the internet have security vulnerabilities [3]. The security concern is becoming more serious with development and availability of tools that enable development and deployment web applications in a fast and easy way. According to a report by the Web Application Security Consortium, about 49% of the web applications being reviewed contain vulnerabilities of high risk level and more than 13% of the websites can be compromised completely automatically [6]. Another security report [7] reveals that most websites were exposed to at least one serious vulnerability every day of 2010, or nearly so (9–12 months of the year) and the average website had 230 serious vulnerabilities.

As a result, a considerable research effort has been put to develop tools for automated assessment of web applications for security. Crawling is an integral part of web application security assessment

tools. Without an effective crawling capability, the efforts to secure web applications will not be able to produce satisfactory results.

1.4.2 Crawling Strategy

Crawling strategy refers to the algorithm which guides the process of crawling a web application i.e. crawling strategy decides how the web application exploration proceeds. For example, a crawling strategy might decide which event or link from the current page needs to be explored next in order to efficiently discover all the web pages of the web application. The most fundamental requirement of a crawling strategy is the construction of a correct model of the web application. Correctness of the model makes sure that the model built as a result of the crawl is indeed the correct representation of the web application. We believe that along with correctness, the model should also be complete in the respect that the strategy should be able to discover all the web pages along with the information to reach them, at least if sufficient time is given to the crawling strategy. The information provided by the crawling strategy will be simple URLs in the case of traditional web applications; however, with RIAs it might include a sequence of events along with the URL.

1.5 Crawling Rich Internet Applications

Rich internet applications introduced several improvements over the traditional web applications, including better responsiveness and information flow, improved user experience, enhanced interactivity etc. but on the other hand introduced new challenges. One of the important problems is the difficulty to automatically crawl these websites.

Each web page in a traditional web application is uniquely addressed by an URL and contains URLs of all other web pages that can be reached from the current page as embedded HTML links. To crawl a traditional web application, the crawler would just be required to extract these URLs from a web page starting from the initial web page and traverse them in an efficient sequence. Hence to crawl a traditional web application it is sufficient to perform link discovery.

In RIAs, on the contrary the current web page can be changed dynamically, sometimes without even requiring user interactions and communication with the server. This is due to the ability of modifying web page DOM using javascript execution. A script may possibly add or replace content that can be received from the server asynchronously. This means many of the client states in RIAs might be only reachable by executing a sequence of javascript events (which are actions that can be detected by javascript such as onClick and are typically triggered by user interactions) starting from an URL of the application. In addition, automatic interaction is more difficult in a RIA environment because of more complex script actions such as timers, mouse movement etc.

This dynamic nature of the RIA changed the notion of mapping the URL to the web page entirely. All these changes has rendered the traditional crawlers inefficient and unable to crawl RIAs, except for a few pages that have distinct URLs

1.6 Motivation

The concerns on the security of the web applications have grown along with their popularity. One of the responses to these concerns about security issues was the development of automated tools for testing web applications for security.

There are various commercial and open-source black-box web application security scanners available (see [8] for a recent survey). A black-box web application security scanner is a tool that aims at finding security vulnerabilities in web applications without accessing the source-code. That is, a black-box scanner only accesses the client-side just like a regular user of the application. When a black-box security scanner is given the URL pointing to the initial page of the application (together with other minimal information that may be required, such as username and password, if the application requires login), it simply tries to discover all web page (client state) of the application that are reachable from the initial page. As the new web pages are discovered, the tool scans each one for possible security vulnerabilities by applying test cases and reports any detected vulnerability to the user. These tools can easily apply a large number of security tests automatically at each discovered page which would otherwise require a long time if done manually.

Crawling is also an integral part of any web application vulnerability scanning. It is clear that effectiveness of a security scanner depends not only on the quality and coverage of the test cases but also on how efficient it is at discovering the pages (client-states) of the application. Unless we can discover all the web pages of the web application, we will not be able to perform a thorough security test. For web application vulnerability scanners, such as IBM® Security AppScan®, one of the most important challenges, apart from keeping updated about new security concerns and breaches, is to keep abreast with this advancement in the web application technologies from traditional application to RIAs. A web scanner's ability to crawl a web application has and will always be integral to testing it for security. The offloading and sharing of logic and functionality between the server and client has had major implications for automated web application crawler and in turn web application vulnerability scanners.

Crawling provides the web scanner a similar ability of accessing the web application in the same way a user would. Essentially, a web application vulnerability scanner will still be testing for the same vulnerabilities in case of RIAs, but it needs to explore all the webpage or client-side states to make sure it has covered all the functionalities and security vulnerabilities. An efficient and effective web scanner not only needs to be intelligent enough to discover all the links but should also be able to emulate all the events to discover all the states of the web application to provide maximum application coverage.

To our knowledge, none of the current search engines, web application testers and analyzers has the ability to crawl RIAs [8] .The problem gets increasingly important as more and more developers and organizations adopt these newer technologies to put their information on the web. A substantial amount of research efforts have been devoted to this problem with a number of techniques developed for crawling RIAs which we will discuss in detail in Section 3. However, many of these techniques still rely of standard Breadth-First and Depth-First strategies. We will discuss these techniques in related works section. In addition only one of the techniques focuses on the efficiency of the crawler. Thus, it is desirable and urgent to provide a systematic crawler which is not only able to crawl RIA completely but efficiently.

There had also been efforts to identify general patterns in RIAs and use those patterns to come up with a reasonable anticipation of the application to design an efficient crawler. [9] introduces such a notion called “model-based crawling”. However, the assumptions used are too strict to be practical for most RIAs. We will discuss the details in related works section.

We have also integrated the ideas of this thesis as a crawling strategy in IBM® Security AppScan® Enterprise [10] , a security scanner for web applications.

1.7 List of Contributions

The following list describes the contributions of this work:

1. A new meta-model, “Menu Model” based on the concept of model-based crawling for crawling RIA is introduced.
2. An algorithm for event prioritization for event-based crawling is presented.
3. A complete crawling strategy for crawling rich internet applications based on menu meta-model is presented.
4. A technique for modifying the initial strategy when the web applications contradict the menu-model hypothesis is also presented to produce efficient results.
5. A prototype which is used to crawl real and test RIAs.
6. Evaluation of the research is presented in comparison with other crawling strategies
7. Experimental results and evaluation of three different heuristics has been presented to help design final crawling strategy.

A paper has been published which extends the work presented in this research [11]. Another paper suggesting efficient methods for solving the challenge presented in Section 2.2 has been published [12]. In addition, one patent covering the research presented in chapter 4 is in filing process at IBM (which has delayed the process of presenting the contents of this research at appropriate conference).

1.8 Organization of the Thesis

This document is organized as follows: Chapter 2 discuss the challenges faced while designing efficient crawling strategies along with assumptions about the web application. Chapter 3 provides an overview of work which is related to this research. Chapter 4 presents an overview of crawling strategy followed by chapter 5 which explains the complete strategy for crawling RIA based on menu model. Chapter 6 contains the experimental results and comparison against other crawling strategies. Finally, the document ends with a conclusion and discussion of future work in Chapter 7.

2. Challenges and Assumptions

An efficient crawling strategy is solution to just one facet of the multi-facet problem of being able to crawl RIA for information gathering and security analysis. Being able to discover all the web pages of the RIA not only requires an efficient crawling algorithm but also largely depends upon the application itself and also on the purpose of the crawl.

In this thesis, we have mostly focused our attention on devising efficient crawling algorithms. However the complete RIA crawling system is composed of many components. The following section will describe in brief, all the challenges and assumptions made about the application and the crawling system without going into much detail. Each of these challenges needs to be addressed as separate research efforts to be able to arrive at an efficient system to crawl RIA. In addition, the application is assumed to meet all the simplifying assumptions to be able to work efficiently with our crawling strategy.

2.1 State Definition – DOM-ID

The purpose of crawling a RIA is to discover all the reachable web pages of the web application. These web pages are the building blocks of the complete web application and are the placeholders for the information and services that the web application is indented to provide. Further, for any security analysis the web application security analysers also need access to these building blocks. Each distinct web page, which can be represented by its Document Object Model (DOM), is a client-side “state” or simply the “state” of the application.

During the crawling process it is essential that the crawler should be able to identify whether it has already discovered a web page or not. This is important to avoid them from entering into infinite

loop also called crawler trap (i.e. exploring a series of page over and over again) while ensuring that all the relevant information about the web application has been discovered.

One can use the key elements of the web page to provide the state definition such as URL of the page, cookies (information stored on a browser by the web server) etc. However, such information will not be valid in case of RIAs as multiple states might share the same URL.

We have used Ayoub et al.`s [13] algorithm to calculate the state identifier which is calculated from the DOM of the web page and is referred to as “DOM-ID”.

2.2 DOM Equivalence

A web application is a complex application providing multiple types of information such as content, structure of the web page, user inputs etc. However, it is completely dependent on the purpose of the crawl what information is relevant. For example if a web application is being crawled for the purpose of content indexing by search engines, then the contents of the web page are important. If two pages have different textual content such as news article but identical structure such as input structure for providing user comments, they should be considered different. When assessing the web application for security vulnerabilities the elements of the web page which allows user to provide inputs are more important that the textual content. Therefore if the previously mentioned states contain identical user input structure but different news article, they should be considered the same.

To be able to design an efficient crawling strategy, we should not only be able to identify the states of the application but also be able to ignore irrelevant information (depending on the purpose of the crawl) when considering whether a state should be considered different from other states. The concept of DOM equivalence helps us achieve the same.

Mathematically, equivalence is defined as a relation that partitions a set into disjoint subsets. Two elements of a set are considered equivalent with respect to the equivalence relation if and only if they belong to the same subset. The intersection of two such subsets is empty and the union of all the subsets equal to the original set. Equality is one obvious equivalence relationship which partitions the set based on the whether the entities are identical to each other or not.

The concept of DOM equivalence takes the process of providing state definition to a level higher where we not only define the state of the application but also define the relationship that will make multiple states equivalent even if their state definition differ. This concept of DOM equivalence hence helps the purpose of the crawl guide the crawling process.

Note that DOM equivalence is a notion that should be considered independent of the crawling strategy. The crawling strategy should be able to work with any provided DOM equivalence relation.

In addition of being important for the purpose of the crawl, the choice of the DOM equivalence relation is also crucial for the correctness of the extracted model and the efficiency of the crawler. If the DOM equivalence relation is too lax (often fails to distinguish between pages that are actually distinct) then this will result in states being classified as equivalent even when they should not be. This could produce a model which is incomplete missing out states. In the scenarios where the crawling is used for security analysis of the web application, missing states would mean that some states will not be analysed and hence could result in undetected vulnerabilities.

On the other hand, if the equivalence relation is too strict then the model will end up containing more states than necessary leading to state explosion and hence inefficiency.

Unfortunately, there is no established DOM equivalence relation that can be used for all crawling purposes.

While the purpose of the crawling is important, the crawler's main responsibility is to discover all the states of the application. In [14] a basic two fold approach of defining state equivalence is provided. Two states are considered equivalent if they satisfy the following two equivalencies:

1. "Crawling Equivalence, $eq_{crawling}$: Two states can be considered equivalent if the set of states that can be reached from the two states are equivalent.
2. Purpose Equivalence, $eq_{purpose}$: Two states are equivalent if they are equivalent based on the purpose of the crawl. Therefore, $eq_{purpose}$ should be substituted according to the purpose of the crawl. For instance, it would be $eq_{security}$ if the application is being evaluated for security vulnerabilities or $eq_{accessibility}$ if the application is being assessed for accessibility."

In [14] it is suggested to consider both crawling equivalence and purpose equivalence to define the DOM equivalence relationship for crawling strategy. It is also argued that depending on the purpose of the crawl, the model of the application discovered might vary. Failing to account any one of the two equivalencies might result in missing states, incomplete or incorrect model of the application or failing the purpose of the crawl.

In addition, we suggested in [12] that some preprocessing of the page may be beneficial to increase the accuracy of the DOM equivalence relation, such as to detect the "not important" parts of the page (advertisements, counters, timestamps, session variables etc.) which should be ignored. These methods not only help to improve the efficiency of the DOM equivalence relationship but also help avoid the crawling strategy to get stuck into infinite loops or crawler traps. We suggested two methods to increase the efficiency:

1. Load-Reload: This method helps to identify and ignore the unnecessary content of the page which are dynamic and are not important for the purpose of the crawl, such as advertisements. We suggested reloading the web page twice to identify the changes that might be ignored.
2. Session Identification: This method helps to ignore the session variables which might result in multiple URLs for the same web page resulting in the crawler accessing the same web page over and over again. We suggested consecutive user login sequences to detect the session variables and ignore them.

The methods suggested above are few of the methods that might help in defining a good DOM equivalence relationship.

The problem of defining a ubiquitous DOM equivalent relationship for all crawling purpose is still an open research area. Further, developing methods to improve the DOM equivalence for specific purposes of crawling is another research area to explore.

For our purpose, we have used the DOM-ID of the web page along with the IDs of all the events enabled at that page for defining the DOM equivalence relationship.

2.3 Event Identification

Events are actions that can be detected event-driven programming languages like JavaScript, ECMAScript etc, such as onClick and event handlers/listeners on the element nodes inside a DOM tree, e.g. HTML, XHTML documents. They are generally triggered by user interactions though there are also complex events that can get triggered automatically such as timers etc.

Aside from the problem of DOM equivalence, identification of events enabled at a state is another open problem. Events identification is important for the purpose of crawling RIAs as the execution of events might result into new states. Hence, missing out events from a state might result missing some state of the application and hence an incomplete model. In addition, as mentioned in section 2.2 we consider set of enabled events at a state to define the DOM equivalence relationship. Failing to recognize events enabled at a state might result in multiple states being considered equivalent when they are not and vice versa.

```
<a onmouseover="this.style.color='red'" onmouseout="this.style.color='black'" id="anchor_id"> This is a changing color text. Try it! </a>
```

Figure 2: Events registered to HTML Anchor element

Identification of an event or calculation of event id might be simplified as identification of the HTML element that has the event registered as handler. For example, in Figure 2 one may use the path to the HTML Anchor element in the DOM tree from the root node (XPath) to identify the event along with the event type. Then the *onmouseover* event could have the ID = *"/html/body/……/div/p/a" + "onmouseover"*. However, this approach has the problem of identifying the same event as different if they are present at different XPath.

Another approach might be to use the HTML element's attributes such as id if they are present, along with the event type, to define the event. For example, the event id of the *onmouseover* event in Figure 2 could be: *"anchor_id" + "onmouseover"*.This too, is not a reliable solution as the HTML element might not have the id attribute.

The approaches discussed above are few techniques that could be used for event identifications but they have their own shortcomings. Like the DOM equivalence problem, the problem of event identification should also be considered independent of the crawling strategy. Further, the crawling strategy should be able to accommodate any event identification method. These challenges are out of the scope of this thesis and should be handled as a separate research effort.

2.3.1 Event Equivalence

The problem of event identification can further be generalized by defining equivalence relationship between the events similar to the equivalence relationship defined for states. The equivalence relationship will help to determine whether multiple events at the same state or events at different states could be considered as equivalent in the sense that they will result in equivalent states or identical states. This might help in improving the efficiency of the crawling strategy by trying to execute and explore events that are not equivalent as they might result in new states. The problem of defining event equivalence relationship is yet another open challenge.

2.3.2 Event Execution Order

In many web applications, the order in which the events enabled at a state are executed determines the resultant state. Since the number of permutations of a set of events grows exponential e.g. with the number of events, trying to verify all possible permutations might require significant amount of time.

However, trying just one possible sequence might result in an incomplete model of the application. An example of such web application is [15] which emulates the working of a Linux operating system.

The problem of defining a valid subset of event execution order to accommodate for all the states that could be reached is another open challenge.

2.4 Deterministic Behaviour of the web application – Statelessness of the server

Our crawling strategy assumes a deterministic behaviour of the web application. This means, from a given state executing an event again should always result in the same resulting state.

The biggest caveat in assuming this deterministic behaviour is the server states of the web application. In our crawling strategy we only include the client-side state of the web application. The above assumption might be contradicted in the case when the client state is the same but the server state would have changed. For example, consider a shopping website where a user can add items to the shopping cart. The event on the page might result in different actions depending upon the items in the cart. In such scenarios, the execution of an already executed event might result in a different state. In [14], the authors also realize the importance of server-side states in building an accurate model of the web application. They also suggest the idea of evaluating the possibility of making a distinction between events that generate requests to the server (and thus may change the server state) and the events that do not. Events that do not go back to the server can be crawled entirely at the client side.

We also assume to be able to reset the web application to the initial state. A “reset” or “reload” is the action of resetting the application to its initial state by reloading the web application initial URL. This action is important in scenarios where we want to go to some state S, from the current state but we have no known path to reach it. In such scenarios, we might want to reset the application to the initial state and then go to the state S. We will be able to do so as such path exists since we would not have been able to discover the state S in the first place.

The problem of including the server states of the web application in the process of crawling is another open question and should be addressed as a separate research effort. For our purpose we assume the web application to behave deterministically. If the web application does not behave deterministically we log an error and ignore the deviation. The first execution of the event from a state defines the behaviour of the event from that state for the complete crawl.

2.5 User Inputs

User inputs present another challenge in crawling web applications. The next state reached by an event may depend of the input provided by the user. The example of a shopping website where the user can add items to the shopping cart is also a relevant example for this problem.

Since the set of possible user inputs can be practically infinite, determining a valid set of user inputs which will help us analyse the web application states in an open question. In [14], the authors also recognize the automatic determination of the format and type of values which will let the web application function correctly to be an open challenge.

For our purpose, we sample the user input space and choose few user inputs by randomly selecting few representatives from the user input space for the purpose of crawling.

3. Related works

3.1 General RIA Crawling

The research area of web application crawling has made significant progress over the last 15 years. This includes remarkable papers such as [16], which is an introduction to Google. The research also facilitated the emergence of other major search engines such as Yahoo! [17] and Bing [18]. In addition to advancement in search engine technologies, the research also helped in the development of crawlers for thorough analysis of web applications for security and accessibility testing, for products like AppScan, WebInspect etc. However, the majority of the research until recently has focussed on crawling traditional web applications.

Traditional web application crawling is a well-researched field with multiple efficient solutions [19]. In addition to the fundamental problem of automatically discovering the pages of the web application, there have been research efforts to use the information discovered, for future crawling purpose such as page revisit policy or page rescheduling policy used by search engines for content indexing purpose [20] [21].

However, none the above mentioned search engines and web application analysis products, and other existing state of the art tools in the field of web application crawling, are sufficient for RIAs [22]. In the case of RIAs, the current research is still trying to address the fundamental problem on crawling i.e. automatically discovering the web pages of the application. This is not surprising given the short history of RIAs, less than a decade old.

In the last few years, several papers have been published to solve the problem of RIA crawling mostly focusing on Ajax based applications. For example, [9] [11] focuses on crawling RIAs to enable

security assessment tools to analyse RIA; [23] [24] [25]focus on crawling for the purpose of indexing and search. In [26], the aim is to make RIAs accessible to search engines that are not AJAX-friendly. In [27]the focus is on regression testing of AJAX applications, whereas [28] is concerned with security testing and [29] focuses on user interface testing. However, except for the work done in [9] [11] most of the research is concerned with their ability to crawl RIAs and not much attention has been given to the actual efficiency of crawling. Crawling RIAs in its naïve form seems like the application of standard Breadth-First and Depth-First strategies, which have been used in most of the published research with some modifications.

[29] [30] introduces a tool called “Crawljax” for crawling RIAs. This research has mostly focussed on making RIA content accessible for the purpose of content indexing by converting the RIA to an application containing multiple static pages. The tool uses a variation of the Depth-First strategy to discover and build a state machine model of the application. However, the tool explores only a subset of the events from the current state. The events that are registered to HTML elements which are different from the previous state are explored in the current state. This approach has the drawback of not being able to discover all the states of the application, as shown in [9]. In addition, the tool uses a concept of distance between the states of the application called “edit-distance”, i.e. is the number of edit operations required to transform one state to another to determine if the new state is considered a different state. Since this concept of distance is not transitive, such approximations may result in states wrongly categorized as equal or non-equal.

Another research effort [31] [32] used the Breadth-First strategy to crawl RIAs. To improve the performance, they cache the results of javascript event executions. Anytime, if the same event is called with the same parameters, the cached results will be used. The drawback with this approach

is that the event execution result might vary if the event is executed from a different state despite the same parameters and also in situations where the server state would have changed.

[33] suggested a manual method to trace and record the execution sequences in RIAs. The recorded traces are then later analysed to form the finite state machine model of the application. In a later paper [34], they introduced a tool called “CrawlRIA” to automate the tracing process of the execution sequences. They used the Depth-First strategy to execute the events starting from the initial state till a state equivalent to a previously visited state is reached. The tool will then reset back to the initial state to continue recording. All the recorded traces are later analysed later to form the finite state machine model of the application using the same approach described in their previous paper.

[26] also used the Depth-First strategy to crawl RIAs though they used a variable to limit the maximum depth explored.

3.2 Model-Based RIA Crawling

3.2.1 Overview

In the case of crawling RIAs, the research is still ongoing to address the fundamental question of automatically discovering the existing pages. This can be attributed to the lack of efficient event-based crawling strategies. At the first instance, application of Breadth-First and Depth-First strategies might look like a possible solution to the problem of event-based crawling. This is more or less the approach taken in relevant research works also as discussed in Section 3.1. But none of these researches has focused on actual efficiency of the crawling strategy. Efficiency is an important

factor when it comes to crawling RIAs as most RIAs are complex web applications with a very large state space. Breadth-First and Depth-First strategies in their standard form and under specified circumstances will eventually be able to crawl RIA. However, one important issue with these techniques is that they are too generic and inflexible in the respect that they do not use any information about the web application such as behaviour, structure etc. gathered during the crawl to improve the efficiency of the crawling process. In [9] the authors indicate opportunities to be able to design more efficient strategies by identifying general patterns in the actual RIAs being crawled and using these patterns to come up with reasonable anticipations about the model of the application. This approach has been defined as "Model Based Crawling". The anticipation can be derived from interacting with the RIA to gain a general notion of how it has been structured and expected to interact with the user. In [9] and [11] the concept of "Model Based Crawling" is defined as:

1. "First, a general hypothesis about the behaviour of the application is conceptualized. The idea is to assume that the application will behave in a certain way. Based on this hypothesis, one can define the anticipated model of the application, which is called as the "meta-model". This will transform the process of crawling from the discovery activity to determine "what the model is" to the activity of validating whether the assumed model is correct".
2. "Once a hypothesis is elaborated and an assumed model is defined, the next step is to define an efficient crawling strategy to verify the model. Without having an assumption about the behaviour of the application, it is impossible to define any strategy that will be efficient".
3. "However, it is important to note that any real world application will never follow the assumed model to its entirety. Therefore, it is also important to define strategies which will

reconcile the differences discovered between the assumed model and the real model of the application in an efficient way".

Thus, Model based crawling defines the goal of a crawling strategy as being able to anticipate automatically an accurate model of the application. However, the model and the strategy must be able to satisfy at minimum the following requirements:

1. The model produced by the strategy must be correct and if given sufficient time must also be complete i.e. it must have correct information about all reachable states and transitions of the application.
2. The model must be built in a deterministic way. Crawling the web application twice with the same strategy and given same server-side data, the model produced should exactly be the same.
3. The model should be produced efficiently. The strategy should be able to gather as much information about the application as soon as possible. Crawling an application may take considerable amount of time and can sometimes be infinite theoretically. In such cases, where the crawling process is interrupted before the end, the model produced by the strategy until interrupted should be able to provide as much information as possible.

To respect the efficiency requirement, model-based crawling defines a two-phase crawling approach described in the next section.

3.2.2 Crawling Phases

RIAs are often complex web applications that have very large state spaces. In such scenarios, it might not be feasible to wait for the crawl to complete. Hence efficiency is always important when

it comes to crawling RIA. In [9] and [14] the authors introduced a reasonable notion of efficiency stating that a crawling strategy that tends to find the states of the application early in a crawl is an efficient strategy. More precisely, a strategy which discovers a larger portion of the application state space early on will deliver more data during the allotted time, and thus be more efficient. Discovering more information will not only help the search engines index more data, but will also help security assessment tools to detect more vulnerabilities if present.

In [9] and [14] the authors defined a two stage approach according to the primary goal of finding all states as soon as possible.

1. State exploration phase: The first phase is the "state exploration phase". It aims at discovering all the states of the RIA being crawled as early as possible. This phase is important as in the given time, discovering the states are more important than discovering the transitions of the application.
2. Transition exploration phase: Once the strategy believes that it has probably found all the reachable states of the application, the strategy proceeds to the second phase, the "transition exploration phase" which tries to execute the remaining transitions after state exploration phase, to confirm that nothing has been overlooked. This phase is important as unless we have explored all transitions in the application, we cannot be sure that we have found all states.

3.2.3 Hypercube crawling strategy

To support the concept of model based crawling, in [9] a specific meta-model in the form of a hypercube have been introduced. To form an anticipated model, the following two hypotheses have been made:

1. The events that are enabled at a state are pairwise independent. That is at a state with a set {e1, e2… en} of n enabled events, executing a given subset of these events leads to the same state regardless of the order of execution.
2. When an event ei is executed at state s, the set of events that are enabled at the reached state is the same as the events enabled at s except for ei.

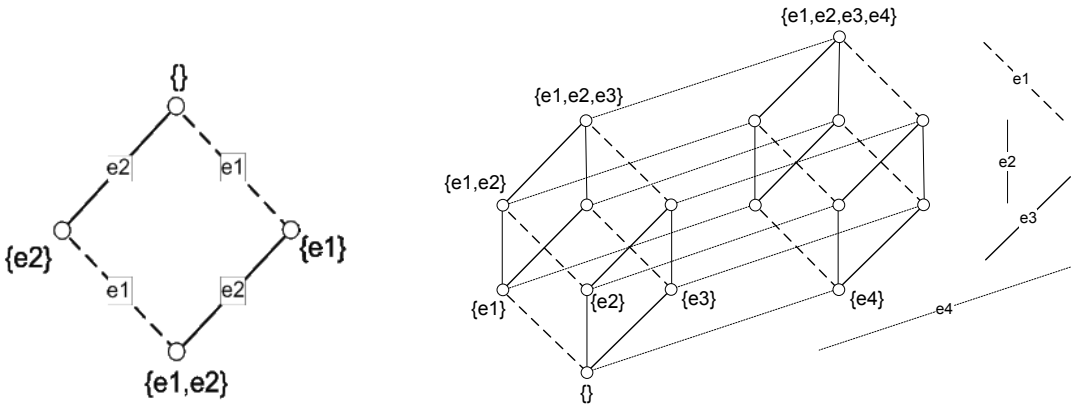

Figure 3: Hypercube of dimension 2 and 4 [9]

The above hypothesis results in an expected model of hypercube of dimension n for a state with n enabled events. There are 2^n possible subsets of n events and the hypercube has a state corresponding to each one of the subset. Figure 3 shows an example hypercube of dimension n = 2 and n = 4. In the hypercube the states are represented by the subset of n events executed to reach it. Each edge corresponds to a transition that is triggered by execution of one of the events. There

are n! different paths from the bottom state to the top state which represents the order in which the events are executed.

In [9] two algorithms have been presented to provide the crawling strategy. In addition, these algorithms have also been proved to produce optimal results if the web application follows hypercube hypothesis to its entirety. The first one called “Minimum Chain Decomposition (MCD)” algorithm provides the optimal paths to discover all the states of the application. Given that the goal is not only visiting every state as quickly as possible, but also crawling the entire application (execute every transition) efficiently, they defined the second algorithm for transition exploration called “Minimum Transition Coverage (MTC)” algorithm. This algorithm focuses on executing every possible event in as few paths as possible (requiring minimum number of resets).

In [9] the authors also presented a technique to modify the initial crawling strategy in scenarios where the web application contradicts the hypercube hypothesis.

The important aspect of the work done in [9]is that they focused on the efficiency of the crawling along with automating the process of discovering all the web pages of the web application and they proved it to be better than existing state of the art commercial tools and other research work.

However the assumptions made about the underlying model of the RIAs were too strict to be realistic for most web applications and it involves complex algorithms to be implemented.

4. Crawling Strategy

4.1 Overview

Based on the various ways in which finite state machines (FSMs) have been previously used and their suitability for capturing states, events, and the transitions resulting from event execution in Ajax applications (as found in [23] and [29]), we consider FSMs as an appropriate technique for modeling Ajax applications.

A web application can be conceptualized as a Finite State Machine with "states" representing the distinct web pages and transitions representing the event executions. The aim of the crawling process is to uncover the model of the web application. We represent the model by an underlying weighted directed graph G = (V, E) where

- V is the set of vertices such that each vertex V_i in V represents a distinct state S_i.
- E is the set of labeled directed edges (arcs) such that an edge (V_i, V_j; e) in E represents the transition from state S_i to S_j which is triggered by the execution of event e at state S_i. The label of the arc represents the event type such as onClick event.
- Event execution cost represents the weight of the edges. The cost is measured in terms of the time taken by the javascript engine to execute the event and pass back the result to the crawl strategy. For simplicity we assume all event have the same execution cost.

The crawling process proceeds by starting with a single vertex $V_{initial}$ representing the initial state $S_{initial}$ reached by loading the initial or base URL of the application. As the crawling proceeds, the model is augmented by adding new vertices for each newly discovered state and new edges are

added for newly discovered transitions. The crawling process finishes when the model cannot be augmented anymore with new information (all the states and transitions have been discovered).

The crawling strategy can discover a new web application state in RIAs in two ways. First, by following a HTML link embedded in the current web page and by execution of an event in the current page.

1. Traditional Crawling

Traditional crawling strategy focuses on exploration strategies for the URL (HTML links) embedded in the current state of the web application. In [35] the basic crawling process of such applications is described. In brief, first the URL is loaded to get the web page. The web page is then parsed to discover all the links (URLs) present in that page. All the newly discovered URLs are then in turn explored and the crawling finishes when there are no new URLs left to be explored.

2. Event Based Crawling

In the context of RIAs, the state of the application can also be changed by javascript event executions. The execution of an event can result in adding or deleting contents of the current web page by manipulating the DOM of the page or in completely new web page. In event-based crawling, the strategy focuses on deciding which unexecuted event among the enabled events at the current page or among other already discovered pages should be executed next.

The crawling strategy returns a list of one or more events back to the script execution engine which then executes all the events returned in one sequence. Once the script execution engine is finished executing all the events, it calls the crawling strategy to decide upon the next set of events.

There are few concerns that are specific to event based crawling as compared to following URLs in case of traditional crawling approaches.

1. Event executions such as javascript event executions cannot be reverted back by using the back or history button as in the case of HTML Link click, as the event executions do not register themselves to the history of the browser. However, the behaviour can be realized by caching the page before executing the event.
2. The server state may have changed due to javascript event execution and caching the page will just help us retrieve the client state of the web application and not the server state.
3. Caching the page may result in memory-size issues if the application has large state space and caching each state is required for the strategy to function efficiently.

The overall RIA crawling strategy should accommodate both the cases of state discovery i.e. traditional and event based crawling, to be able to build the complete model of the application. If the crawling strategy focuses on one specific approach, then it might result in an incomplete model missing out states of the web application. Although combining two strategies in an efficient manner is not the focus of this thesis and should be handled as separate research effort.

We focus only on designing efficient strategies for event based crawling in this thesis. We also use the event execution count as the metrics for evaluating the performance of the crawling strategy. Since execution of an event by the javascript engine and producing the result takes time, we believe

it is a reasonable metrics to evaluate the performance. In addition, as mentioned in Section 2.4 we also make use of a special type of event, called “reset”, which is the action of resetting the application to its initial state. We measure the performance in terms of the number of events and resets required to discover all the states first and then all the transitions of the application. We say that a crawling strategy that tends to finds the states of the application early in the crawl is an efficient strategy, since finding the states is the goal of crawling.

5. Menu Crawling Strategy

5.1 Menu hypothesis

Following the notion of model based crawling; we introduce here a novel hypothesis called “Menu Model”. To form the anticipated model, we make the following hypothesis about web applications:

“The result of an event execution is independent of the state (source state) where the event has been executed and always results in the same resultant state”

We can conceptualize this as a unique mapping between the event and resulting state.

{Event} -> {Result_State}

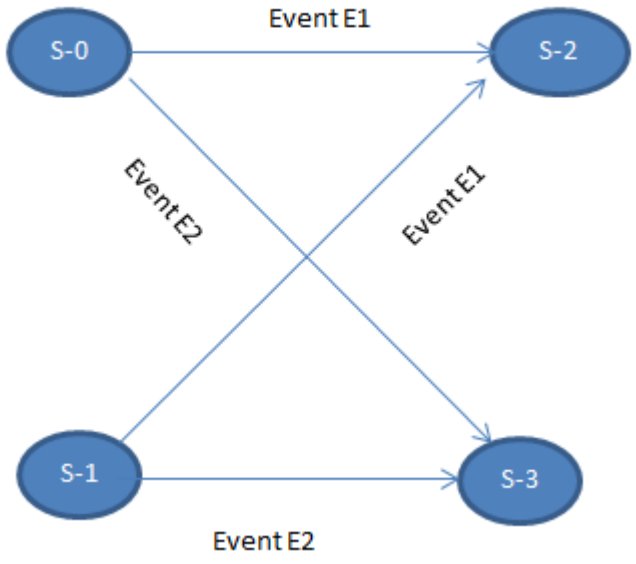

Figure 4: Menu Category Events

We believe that it is a reasonable hypothesis since for many events of the web applications such as mouse click events, execution of the event results in the same resulting state irrespective of the state in which the event was executed. Such behaviour for example is realized by the menu items present in a web application or other common applications such as “home”, “help”, “about us” etc. Executing these menu items will result in the same resulting state. We call these events menu events, the model as menu model and the corresponding crawling strategy menu crawling strategy.

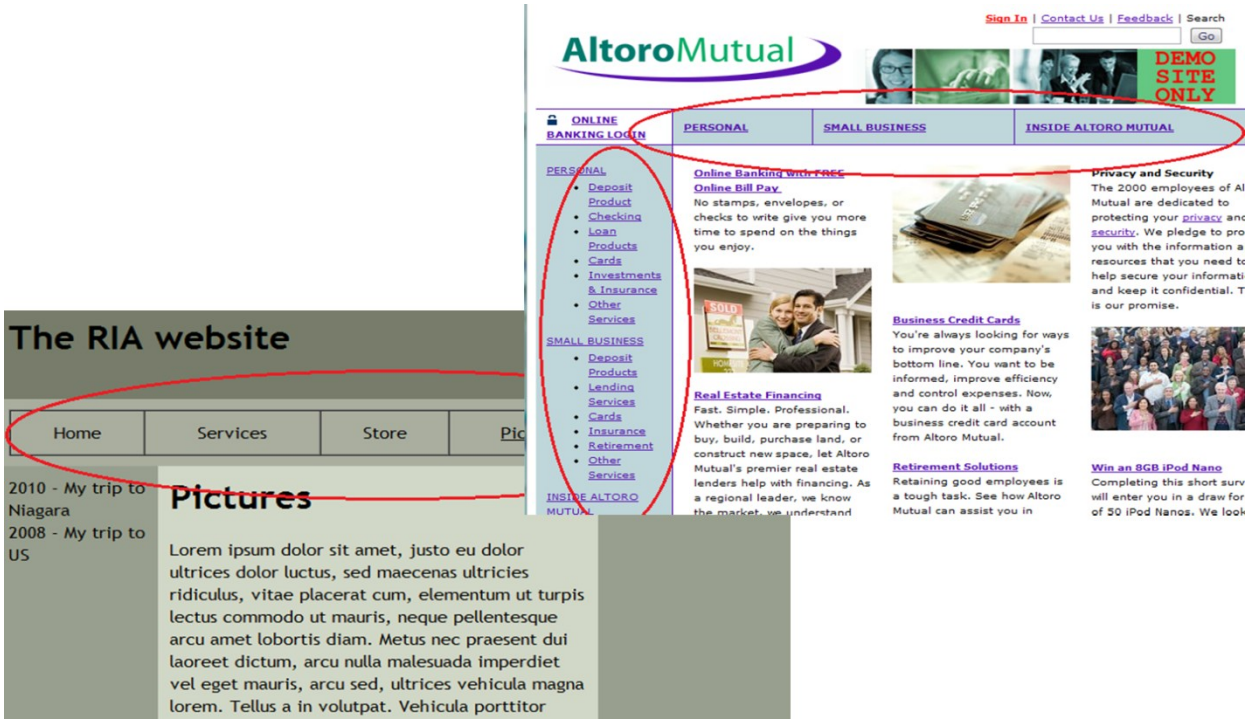

Figure 5: Sample RIAs following our hypothesis

The sample web applications in Figure 5 are examples that follow our hypothesis in the sense that clicking on the items such as home, services etc. always produces the same results i.e. the resulting web application state is always the same irrespective of the state in which the item was clicked.

Following on the guidelines of model-based crawling, we introduce crawling strategies for the two phases, namely state exploration phase and transition exploration phase. Furthermore, it is very unlikely that a web application will follow our hypothesis to its entirety such that all the events enabled in the web application will always result in the same resulting state. Rather, the resulting state might not only depend upon the event executed but also on the state from which the event was executed. To handle such violations, we also introduce adaptations to be followed by the crawling strategies to result in an efficient crawling process.

5.2 Overview

The menu model crawling strategy is an event-based exploration strategy. The underlying strategy of the menu model is the process of categorizing the events enabled in all the discovered states into different priority sets. The priority of the set is defined by the execution count of the events present in that set. The execution count of the event represents the total count of execution instances of the event from different states. For example, the crawling strategy defines a set of global unexecuted events which contains all the events which have not been executed anywhere in the application discovered so far. Similarly, it defines another set called menu events which contains all the events in the application which follow our hypothesis in the way that their execution from different states led to the same resulting state.

The priority is defined as:

1. First, the events with lower execution count have higher priority compared to events having higher execution count.

2. Secondly, the events contradicting our hypothesis have higher priority over the events that follow our hypothesis.

Events with higher execution count generally converges to a set of resulting states, if not to one in case the application contradicts our hypothesis. Hence the events higher execution count are more likely to result in one of the already discovered states and the events with less execution count have higher probability of discovering a new state. Further, the events which follow our hypothesis will always result in the same resulting state and hence it is more reasonable to prioritize events which contradict our assumption higher.

The categorization of the events is an ongoing process throughout the crawl. The priority sets are updated as new enabled events are found in newly discovered states and as more information about results of the execution instances of the events are found. In addition, the events are moved from higher priority set to lower priority as their execution count increases.

The menu crawling strategy maintains the model of the application discovered so far as a weighted directed graph, G = {V, E} where V represents all the states discovered and E represents all the transitions i.e. executed events discovered so far along with some predicted information and resets. Resets are represented as known transitions in the graph however with different cost. We will discuss the details and purpose of the predicted information and resets later in the section.

Each v_i in V contains following information about the state it represents:

1. State ID

   This is the ID of the state which is used by the state equivalence function to help the crawling strategy decide whether a state has already been visited or not. As described in Section 2.1, it contains the DOM-ID of the web page and the list of the IDs of all the events

enabled at that state. The DOM-ID and event IDs are calculated using the algorithm mentioned in Sections 2.1 and 2.3.

2. Executed events

   It contains the list of the IDs of all the events that have already been executed from this state and their resulting state IDs.

3. Unexecuted events

   It contains the list of the IDs of all the events that are enabled at this state but have not yet been executed.

Similarly, each e_i in E contains the following information about the event it represents:

1. Event ID

   This is the ID of the event represented by this transition and is calculated as described in Section 2.3. It contains the complete string of the HTML element the event is registered to and the type of the event

2. Predicted

   This flag signifies whether the execution result of the event has been assumed. The flag is false for all the known transitions i.e. all the events that have been executed. The details and the purpose of this flag will be discussed later in this section.

The menu crawling strategy also maintains the following information about all the events enabled at all the states discovered so far:

1. Event ID

   This is the ID of the event.

2. Priority

   It represents the priority of the event.

3. Resulting State

   The DOM-ID of the state reached on the first execution of the event.

The menu crawling strategy uses the event priorities to decide the next event to execute. If the selected event is not present in current state, S_{curr} then the strategy tries to find the shortest path to the state, S_{next} where the selected event is enabled and unexecuted. We use Dijktras's [36] shortest path algorithm to calculate the shortest path from the S_{curr} to S_{next}. The shortest path algorithm is run on the instance of the graph G maintained by the menu crawling algorithm. It is important to note that the shortest path will also contain the predicted transitions. In addition we also make use of "shortest known path" which is calculated similarly but does not include predicted transitions.

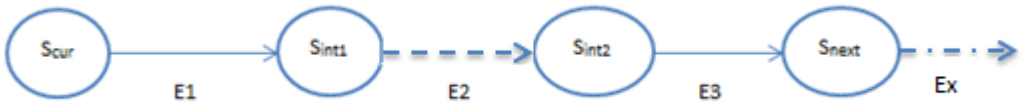

Figure 6: Path from the current state to state, Snext, where the next event can be executed

For example, in Figure 6, the shortest path function returns the event list {E1, E2 and E3}. Events E1 and E3 are already executed events, represented by known transitions. Event E2 is an unexecuted event and is represented by a predicted transition. Event Ex is the unexecuted event to be executed next by the menu crawling strategy.

Following the model-based crawling approach, the menu crawling strategy consists of two phases:

1. Menu state exploration phase
2. Menu transition exploration phase

The menu crawling strategy may alternate between these two phases multiple times before finishing the crawl. The strategy finishes the crawl when it has executed all the events in the application which guarantees to have discovered all the states of the application.

5.3 Architecture

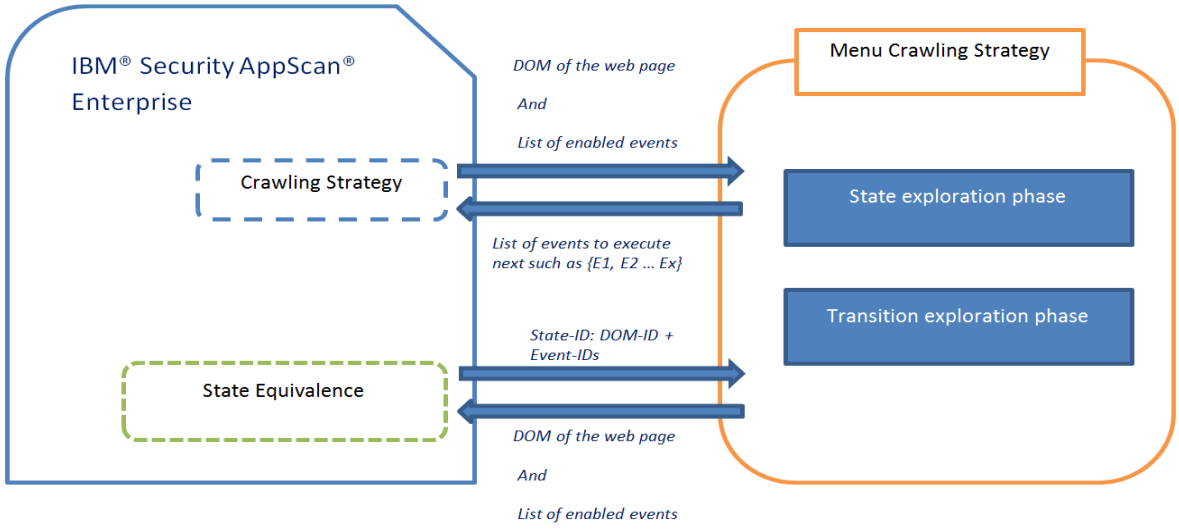

Figure 7: Path from the current state to state, Snext, where the next event can be executed

IBM® Security AppScan® Enterprise [10] is a product for analysing websites for security vulnerabilities and accessibility issues. We have integrated our strategy in the AppScan® framework to crawl the web applications. Figure 7 shows the overall architecture of the integration of menu crawling strategy with AppScan®. The crawling strategy of AppScan® calls the menu crawling strategy with the DOM of the current web page along with all the events enabled at that page. The menu crawling strategy then decides upon the next set of events to be returned back to AppScan®.

The tool then executes all the events from the list one after another in the same sequence as present in the list and returns the DOM of the resulting web page along with the enabled events to menu crawling strategy. This process continues till menu crawling strategy is finished executing all the unexecuted events in the application.

Integrating the menu crawling strategy with AppScan also allows for the possibility of using an equivalence function which also takes the purpose of the crawl into account since AppScan contains such functions (for example for accessibility or security testing). The menu crawling strategy calls the state equivalence module of AppScan® with the DOM of the page and list of enabled events. The tool returns the calculated the state id which is the “DOM-ID” of the DOM of the page along with IDs of all the events.

5.4 Menu state exploration phase

5.4.1 Overview

The primary goal of the menu state exploration phase is to discover all the states of the application as soon as possible. The menu state exploration phase will be henceforth referred to as state exploration phase.

The state exploration phase tries its best to find all the states of the application as soon as possible so that if the crawling is interrupted before the end, it would have discovered all the states or a majority of them. However, one important point is that such a best effort approach does not necessarily guarantees that the actual results will be the same as expected. The menu crawling

strategy uses a greedy approach in the sense that it tries to discover a new state at the shortest path from the current state. This means that maximizing the number of discovered states after 1 event execution might result in lowering the value after 2 or more event executions. This becomes evident in the example below where the menu crawling strategy is able to discover a new state by executing one event from the current state. However, it losses in terms of the total number of events required to discover more new states.

For example, consider the part of the application state space shown in the figure below.

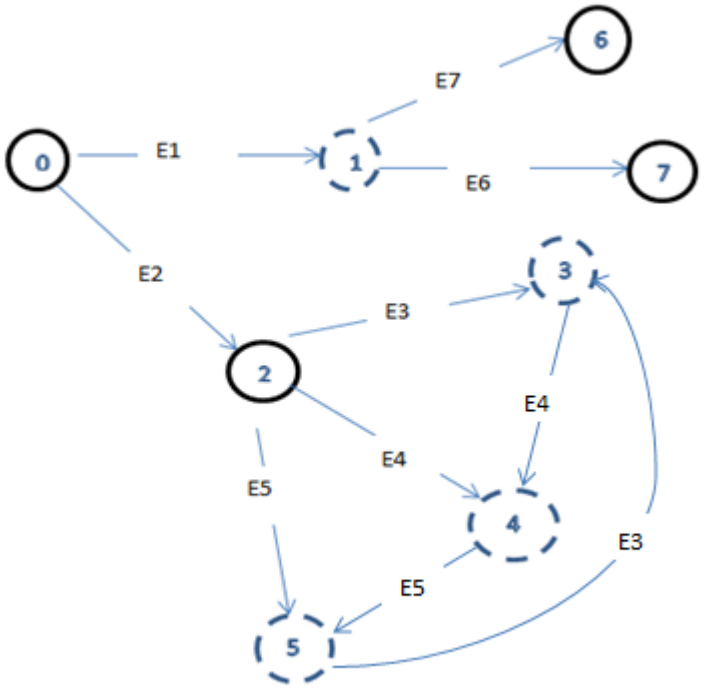

Figure 8: State Exploration

All the events in Figure 8 are already executed events from other states than the states shown in the figure. In addition, the event E1 has the highest priority among all other events. Also states 0, 2, 6 and 7 are already discovered states and states 1, 3, 4 and 5 have not yet been discovered and hence are new states of the application. The menu crawling strategy will give priority to E1 compared to others. However, it might be argued that it is preferable to execute E2 prior to E1 as it will help us discover more states with less total event executions (our primary goal).

But, since we don't have any prior information about the web application, the menu crawling strategy try to use greedy approach to try to discover new states with least cost from the current state. This will help us achieve our goal if the execution would have been interrupted just after one event execution from state 0. In this situation, we would have found one more state compared to executing E2. However, if the algorithm would have been run for more duration, executing E2 would have given us better result. Hence, the best effort strategy not necessarily guarantees optimal results.

The state exploration phase finishes when it believes to have discovered all the possible states that could be found based on the strategy.

5.4.2 Events categorization

To assign priorities to events for deciding the next event to execute, the state exploration phase defines the following categories of events based on the execution count:

1. Globally unexecuted events:

   These are the events that have not yet been executed at any state discovered so far. These events have the highest priority in the list of events to be executed next. We believe

executing a new event which has not yet been executed anywhere in the application has more probability of discovering a new state than the events that have already been executed at some state. This is reasonable as already executed events are assumed to have the same resulting state from all the states.

2. Locally unexecuted events:

   These are the events that have been already executed at some discovered state but have not been executed at the current state of the application. If the application behaves in accordance with our hypothesis, then these events will always result in already discovered state. Only in the case where the application contradicts our assumed model, these events might discover new states or result in already discovered states different than their earlier results. To help define the priority among the local unexecuted events the state exploration phase distinguishes the following sub-categories.

   2.1 Non-classified event:

   These are the events that have been executed just once. However, our hypothesis assumes that their second or further execution will result in the same state as resulted in their first execution instance. These events have the priority next to globally unexecuted events.

   2.2 Non-Menu events:

   These are the events that have been executed more than once and have resulted in different states on their consecutive executions from different states and hence are the events that contradict our hypothesis. For example events such as next or previous events which are used for navigation in a web page such as fetching

next/previous image or articles belong to this category, since every execution of the event will probably result in different state. However, one important note can be made that the different states are not necessarily new states. These events have the same priority as non-classified events.

2.3 Menu events:

These are the events that have led to the same resulting state on consecutive executions from different states. These are the events that follow our hypothesis. We believe that the probability of these events resulting in a different (already discovered) or new state on third or further executions is low, given that it has resulted in the same state on the first two executions. Hence these events are given the least priority among all events.

2.4 Self-Loop events:

As the name indicates, these are the events that have resulted in the same state from where they were executed. For example events such as refresh events belong to the category of self-loop events. Self-loops events are the events that do not cause any change in the state based on the state definition and state equivalence relationship. These events have the same priority as menu events.

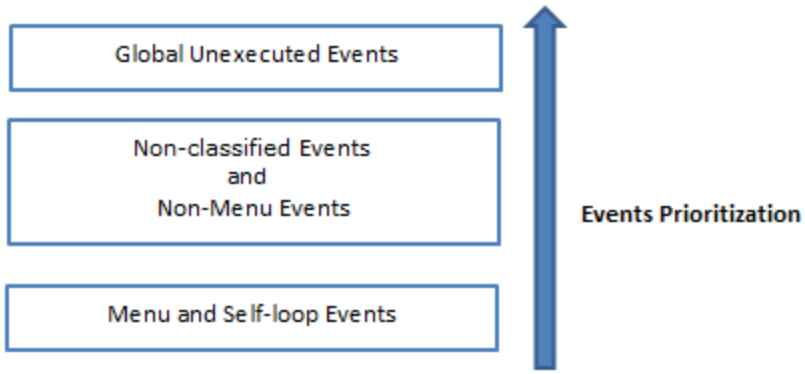

Figure 9: Event prioritization

Figure 9 shows the prioritization of different types of event to be selected for execution. The Non-classified and Non-menu events have the same priority at the application level i.e. when the menu crawling strategy searches for the next event to be executed, then a state with either events has the same priority to be picked up next. However if the next selected state has unexecuted events from both categories, then Non-classified events are prioritized over Non-menu event to be executed next.

5.4.3 Modified application graph

The menu crawling strategy maintains a modified version of the application graph. Along with the discovered states and transitions, the graph G also contains some predicted edges. These edges correspond to the non-classified, menu and self-loop events from states where they are enabled but have not yet been executed. The predicted resulting state of unexecuted menu and non-classified event is the state which was reached on their first execution. Similarly, for unexecuted self-loop events the predicted resulting state is the same state where the event is enabled.

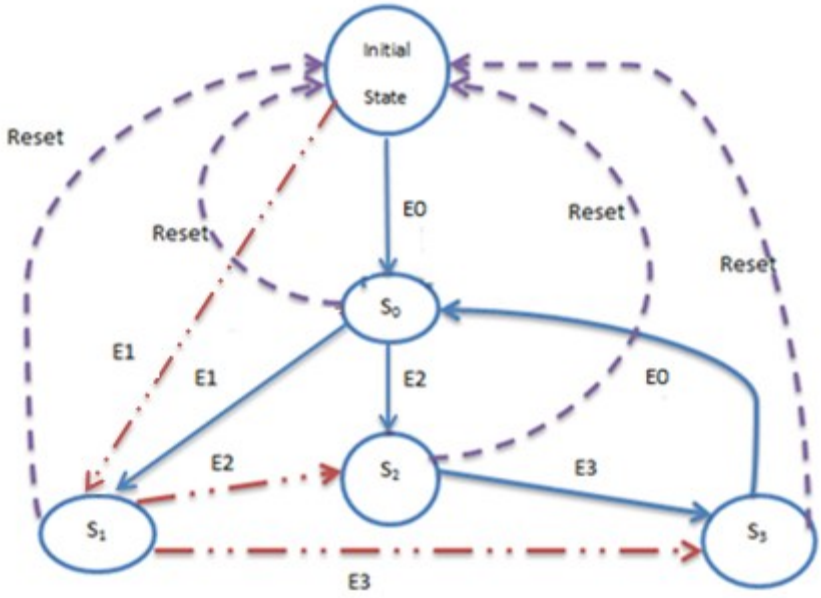

Figure 10: Graph showing virtual edges and known transitions

Figure 10 shows an instance of such a modified graph. The events represented by the bold lines are known transitions i.e. already executed events. The events represented by partial dashed lines are predicted edges i.e. the resulting state of these events have been assumed. In addition, each state has the option to go back to the initial state by using reset.

5.4.4 Menu Crawling strategy

The state exploration phase starts with categorizing the events enabled at the initial state. All the events will initially belong to the globally unexecuted category. If more than one event has the same priority at a current state then one among them is picked at random. As the crawling proceeds, the state exploration phase updates the priority sets with newly discovered information.

The menu crawling strategy uses the priority of the events to decide the next event to be executed. As stated in previous section, all the events from a higher priority set are exhausted before executing an event from lower priority set. Hence, at any given point in the state exploration phase, the whole application (discovered so far) is searched to find an event with highest priority. If the selected event is not enabled or has already been executed at the current state S_{curr}, the menu crawling strategy finds the state where the event has to be executed, S_{next}.

As the crawling proceeds and new application states are discovered, all the events enabled at the newly discovered states are placed into different set based on their priority. Also for each non-classified, menu and self-loop events discovered, the strategy adds a predicted edge to the graph to be used by the shortest path algorithm.

During the state exploration phase, the menu crawling strategy executes all the unexecuted events in the application except for categorized menu and self-loop events. These events have high probability of resulting in already discovered states and hence are not explored in state exploration phase. These events will be handled by the transition exploration phase which we will describe in later sections.

5.4.5 Shortest path and Event assumptions

The menu crawling strategy uses Dijktra's shortest path algorithm [36] to find the shortest path from the current state, S_{curr} to the next state with an unexecuted event, S_{next}. As discussed in the previous section, this algorithm uses the modified version of the graph which contains predicted edges for any unexecuted non-classified, menu or self-loop events. The shortest path is a list of one or more events. Executing the list of events is guaranteed to lead us to the state, S_{next} only in

scenarios where all the events present in the path are already executed events i.e. known transitions. When the graph contains predicted edges, we can no longer guarantee that we will reach state S_{next} on execution of the path. If any of the events whose result has been assumed, contradicts our assumption, then we might end up in some different state than the expected one.

The menu crawling strategy uses a modified shortest path function in the respect that it returns the path as a combination of multiple path segments instead of a single list of events. This is done to verify the intermediate predicted edges results. If these edges result in violation then the current path will become invalid and hence the strategy would need to adapt. We will discuss the details of strategy adaptation in next section. In addition, menu crawling strategy does not verify all the intermediate states in the path but only the states reached after the execution of predicted edges.

Each path segment is a list which might start with a reset followed by zero or more already executed events and ending with an event whose result has been predicted and the predicted resulting state.

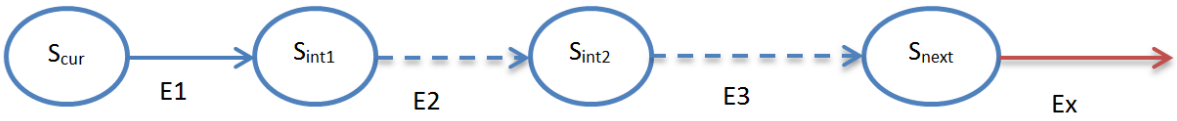

Figure 11: Event path from current state, Scurr to state with next event to execute, Snext

In the figure above, event E1 is an already executed event from current state, events E2 and E3 are menu events whose results have been assumed and Ex is the next event to be executed by the menu crawling strategy. The path function will return the following path segments to the crawling strategy:

PATH_S_{next}:

1. Path segment 1: Event List ={ E1, E2 }, Expected State = { S_{int2} }
2. Path segment 2: Event List = { E3 }, Expected State = { S_{next} }
3. Path segment 3: Event List = { Ex }, Expected State = { null }

The menu crawling strategy executes one path segment at a time i.e. it returns one path at a time to the AppScan® tool and then verifies the result obtained by comparing the resulting state against the expected state until it reaches the state S_{next} or it encounters a violation in between.

5.4.6 Violation of assumptions and adaptation of Strategies

The menu crawling strategy makes assumptions about event execution results which we refer as predicted edges. However it is very likely that the web application will sometimes contradict our assumption and the actual result of an event execution will be different from the predicted one. In such scenarios, we discard all the current path information and follow the same process of finding the path to the state with next unexecuted event from the current (violated) state. The example below describes the process in detail.

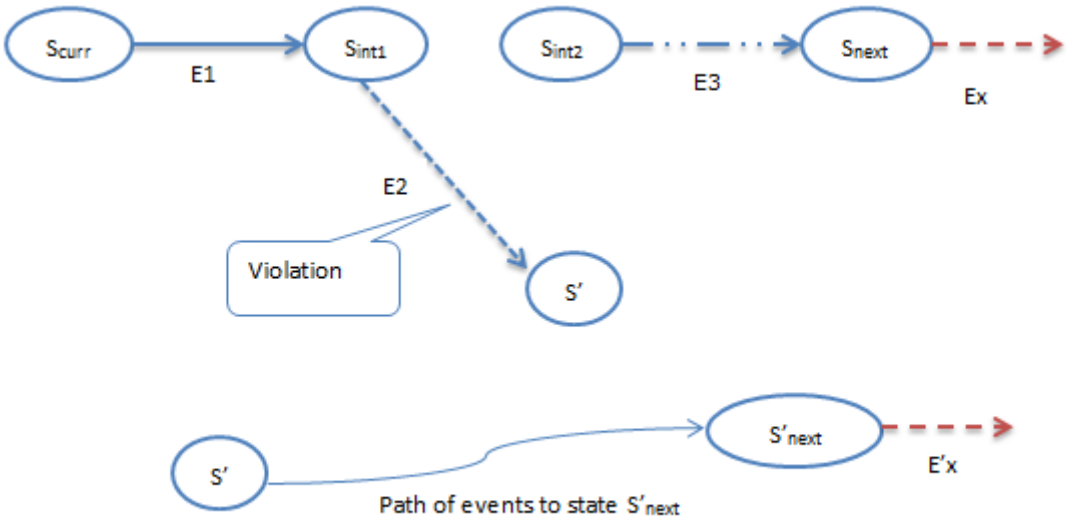

Figure 12:Example of violation instance in the path in Figure 11

For example, in Figure 12 the execution of the path segment 1 of PATH_S_{next} described earlier results in a violation and the menu crawling strategy finds itself in state S' rather than state S_{int2}. In this scenario, the menu crawling strategy ignores all the further path segments information i.e. path segment 2 :{E3} and path segment 3: {Ex} as they might no longer be valid. Instead, the menu crawling strategy starts the same process again to build a path from the current state reached to the next state with an unexecuted event. However, the next state S'_{next} decided this time might not necessarily be the same as the previously decided one, S_{next}.

The violation of the assumption might be favourable for the menu crawling strategy in the cases where the resulting state is a new state. In this case, we would have discovered a new state over a shorter path i.e. less event executions.

The menu crawling strategy ignores the violation of the predicted edges. However, if the event resulting in violation is a non-classified event then the menu crawling strategy categorizes the event before starting with the process of finding the path to the next unexecuted event.

The state exploration phase tries to execute all the events discovered except for the categorized menu and self-loop events. These are represented by predicted edges in the application graph. However, some of them might get executed as part of the path segments to reach to the state with next event to execute. After this the menu crawling strategy moves to the next phase of the crawl: Transition exploration phase.

5.5 Menu transition exploration phase

5.5.1 Overview

The transition exploration phase tries to verify the validity of all the assumptions made at the end of the state exploration phase and executes all remaining menu and self-loop events. This is important as unless we execute all the events in the application we do not know for sure whether we have discovered all the states of the application. If the application follows our hypothesis, then we are guaranteed to have found all the states of the application by the end of state exploration phase. However, in the case that some of the menu or self-loop events do not follow the assumptions, they may lead to a new state that was not discovered during state exploration phase.

The primary goal of the transition exploration phase is to find a least cost path to execute all the remaining events in the application. The cost of this path is measured in terms of the total number of events and resets required to execute all the remaining unexecuted events at least once.

A sequence $v_0, e_1, v_1, e_2 \ldots v_{n-1}, e_n, v_n$ where v_i are vertices, e_i are edges, and for all i the edge e_i connects the vertices v_{i-1} and v_i is called a walk of the graph.

We can conceptualize the transition exploration problem as finding a least cost walk of the application graph traversing through all the edges representing the unexecuted events at-least once.

The transition exploration phase does not expect to discover any new states but it may. The newly discovered state and events enabled at this new state might result in the discovery of more new states. Since we always have the priority of finding all the states of the application as soon as possible, we pause the transition exploration phase when a new state is discovered and return to the state exploration phase. The state exploration phase proceeds in the same fashion as discusses before.

When the state exploration phase finishes, the transition exploration phase starts again, in the same fashion. The menu crawling strategy might alternate between the state and transition exploration phases multiple times before finishing the crawling the process.

5.5.2 Graph walk

During the transition exploration phase, the application graph contains edges corresponding to executed events, predicted edges corresponding to unexecuted menu and self-loop events along

with reset edges from each state to the initial state. The transition exploration phase calculates a walk of the application graph to cover all the unexecuted events possibly using already executed events and resets.

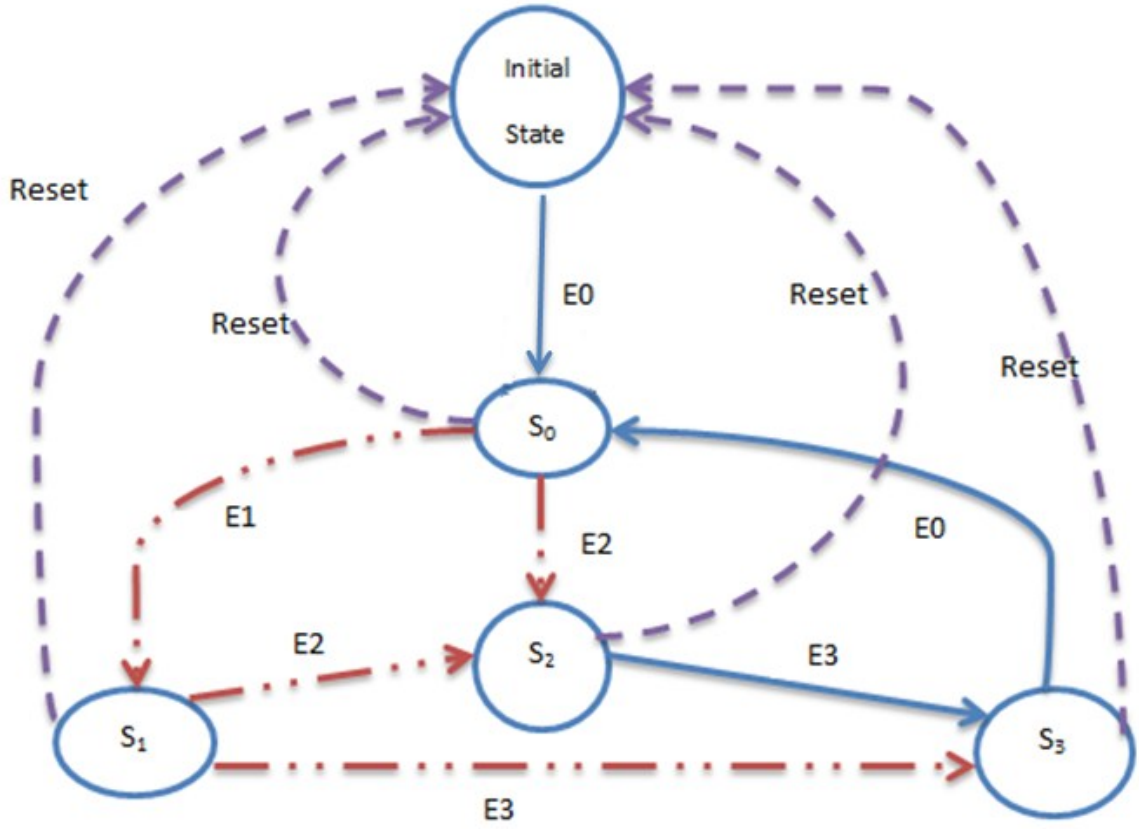

Figure 13: Graph showing virtual edges, known transitions and resets

In Figure 13, events E1, E2 (at states S_0 and S_1) and E3 (at state S_1) are unexecuted events and events E3 (at state S_2) and E0 (at state S_3 and initial state) are executed events i.e. known transitions of the graph. The results of all the unexecuted events have been assumed. A possible walk can be thought of as a sequence { E1, E3, E0, E2, E3, E0, E1, and E2} starting at vertex S_0 and

terminating at vertex S_2. There can be multiple such walks, but transition exploration phase aims for least cost walk.

The transition exploration phase uses a walk generator function to calculate the walk for executing all the remaining edges. We will discuss the details of the walk generation algorithm in later section. However, one important concern is that the application graph includes predicted transitions. Hence, executing the event sequence given in the walk might not result in the expected coverage. After a single violation the menu crawling strategy might get diverted and the event sequence might no longer be valid.

Hence, the transition exploration phase uses a step-wise construction of the whole walk similar to the shortest path construction in state exploration phase. Before returning the complete sequence of events, the walk generator function splits the event sequence into multiple walk segments. Each walk segment might start with a reset followed by zero or more already executed events and ending with an event whose result has been assumed along with the expected resulting state.

For example, the sample walk defined for the Figure 13 is { E1, E3, E0, E2, E3, E0, E1, and E2} . This walk information is segmented as:

Walk Segments, WALK_G:

1. Walk segment 1: Event List = {E1}, S_{next} = {S_0}, Expected State = {S_1}
2. Walk segment 2: Event List = {E3}, S_{next} = {S_1}, Expected State = {S_3}
3. Walk segment 3: Event List = {E0, E2}, S_{next} = {S_0}, Expected State = {S_2}
4. Walk segment 4: Event List = {E3, E0, E1}, S_{next} = {S_0}, Expected State = {S_1}
5. Walk segment 5: Event List = {E2}, S_{next} = {S_1}, Expected State = {S_2}

The event list of a segment represents the list of event to be executed before the resulting state will be verified against the expected state. The expected state represents the state that should be reached after the event sequence execution if the application follows the hypothesis. The state S_{next} represents the state from which the next unexecuted event would be executed in that segment.

The transition exploration phase will have instances where the predicted events will result in violation the resulting state of a segment will be different from the expected state. We describe the violation and strategy adaptation in detail in the next section.

5.5.3 Violation and Strategy adaptation

5.5.3.1 Overview

The menu crawling strategy executes the complete walk as a list of walk segments as returned by the walk generator function. However, it is very likely that we will have instances where the application will violate our hypothesis and the execution of an assumed event will result in some other state than the expected state. The violation will result in one of the following scenarios:

1. **Wrong state**

   The resulting state is an already discovered state but different from the expected state of the walk segment. This violation might render the walk no longer being the least cost walk of the graph. This might happen because the walk generator function had used the assumptions about the resulting state to define a least cost walk. The violation however has updated the graph in

two ways: First, the predicted edge will no longer exist and second, a new edge is added for the executed event resulting in a different state.

An option could be to use the new information and recalculate a new least cost walk. However, the walk generator function uses complex algorithms and heuristics to define a least cost walk. This operation is both CPU and time intensive and hence it is not favourable to calculate the walk over and over again. In general, the time required to calculate the shortest path from the violated state to the next state S_{next} and executing the path to realign is much less as compared to recalculation of the walk. In addition, if the application contradicts our assumption for a majority of event then the transition exploration phase would be spending most of its time in recalculation of the walk rather than executing all the remaining events.

Hence, for efficiency the menu crawling strategy ignores the violation and continues with the same walk defined before. The menu crawling strategy finds a shortest known path to the next event in sequence from the current state and continues with the walk similar to the process followed in state exploration phase. Note that as described earlier the shortest known path will not contain any predicted edges.

2. New state

In the instance when the violated state is a new state of the application, we pause the transition exploration phase temporarily and return back to the state exploration phase.

In addition, the transition exploration phase does not discard the walk calculated in the transition exploration phase. It keeps all the later walk segments from the currently executed one in a buffer before pausing the transition exploration phase. The state exploration phase continues in the similar fashion as described in the previous section. Though during the state exploration phase, the menu crawling algorithm will save and keep track of any newly discovered

information such as new unexecuted menu and self-loop events, new states and new transitions. The menu crawling strategy returns back to transition exploration phase after finishing with state exploration phase.

At this time, the walk saved in the buffer might no longer be least cost or valid. This might happen first because of the same reasons of violation discussed above. Second, the state exploration phase might have added new menu and self-loop events to the graph and they are not accommodated in the current walk. Hence executing the walk will miss these newly added unexecuted events.

However, for the same reason of efficiency as discussed above the menu crawling strategy uses the walk saved in the buffer before pausing the transition exploration phase and accommodates the newly added unexecuted events later.

The transition exploration phase realigns to the saved walk after returning back from the state exploration phase by finding the shortest known path from the current state to the state S_{next} of the walk segment just next to the violated walk segment.

Once the menu crawling strategy finishes with the current transition exploration phase i.e. executes all the walk segments of the current walk, WALK_G, it searches the application for any remaining unexecuted events which might have been discovered during the current transition exploration phase. If there are still some unexecuted events, the menu crawling strategy will again start with transition exploration phase by defining a new walk on the remaining unexecuted events.

5.5.3.2 Examples of violation instances

To avoid the cost of recalculating the walk becoming the performance bottleneck of the crawling process, the menu crawling strategy prefers to realign to the already defined walk. The menu

crawling strategy does this by finding the shortest known path from the current state to the state S_{next} of the walk segment just after the walk segment which resulted in the violation.

Continuing with the same walk defined in previous section, WALK_G, suppose the walk segment 2 i.e. event list {E3} resulted in violation with the resulting state S_v as shown in Figure 14

1. Wrong state

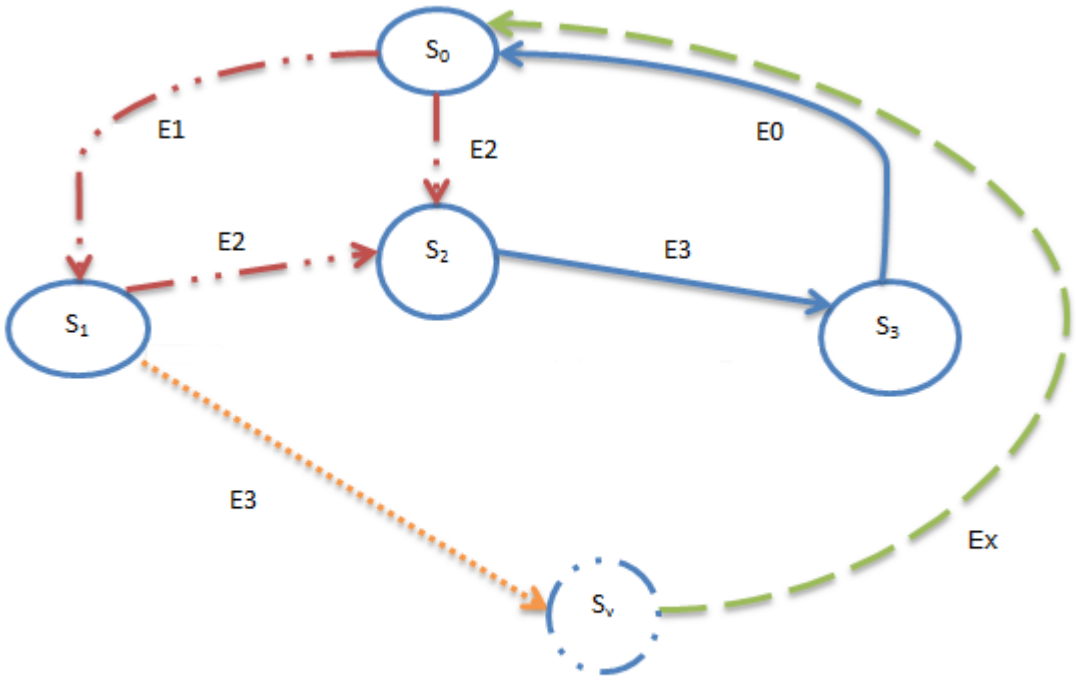

Figure 14: Event execution violation instance (resulting in wrong state) during transition exploration phase

If the resulting state S_v is an already discovered state then the menu crawling strategy finds the shortest known path to the state which has the next event to be executed i.e. the state S_0 which is present in walk segment 3. Suppose the shortest path returned by the function is Ex. The path Ex is a list of one or more already executed. The walk segment 3 is then

modified by discarding all the other events except the last unexecuted event in the list and including Ex. Hence, the new walk segments will be:

WALK_G:

1. Walk segment 3: Event List = {Ex, E2}, S_{next} = {S_0}, Expected State = {S_2}.
2. Walk segment 4: Event List = {E3, E0, E1}, , S_{next} = {S_0}, Expected State = {S_1}
3. Walk segment 5: Event List = {E2}, S_{next} = {S_1}, Expected State = {S_2}

The transition exploration phase will continue in the same fashion as if no violation happened.

2. New state

The example below describes the scenario in detail in the case when the violated state S_v is a new state. The example contains two violation scenarios i.e. the menu crawling strategy goes back to state exploration phase twice and realigns to the saved walk after it finishes each state exploration phase.

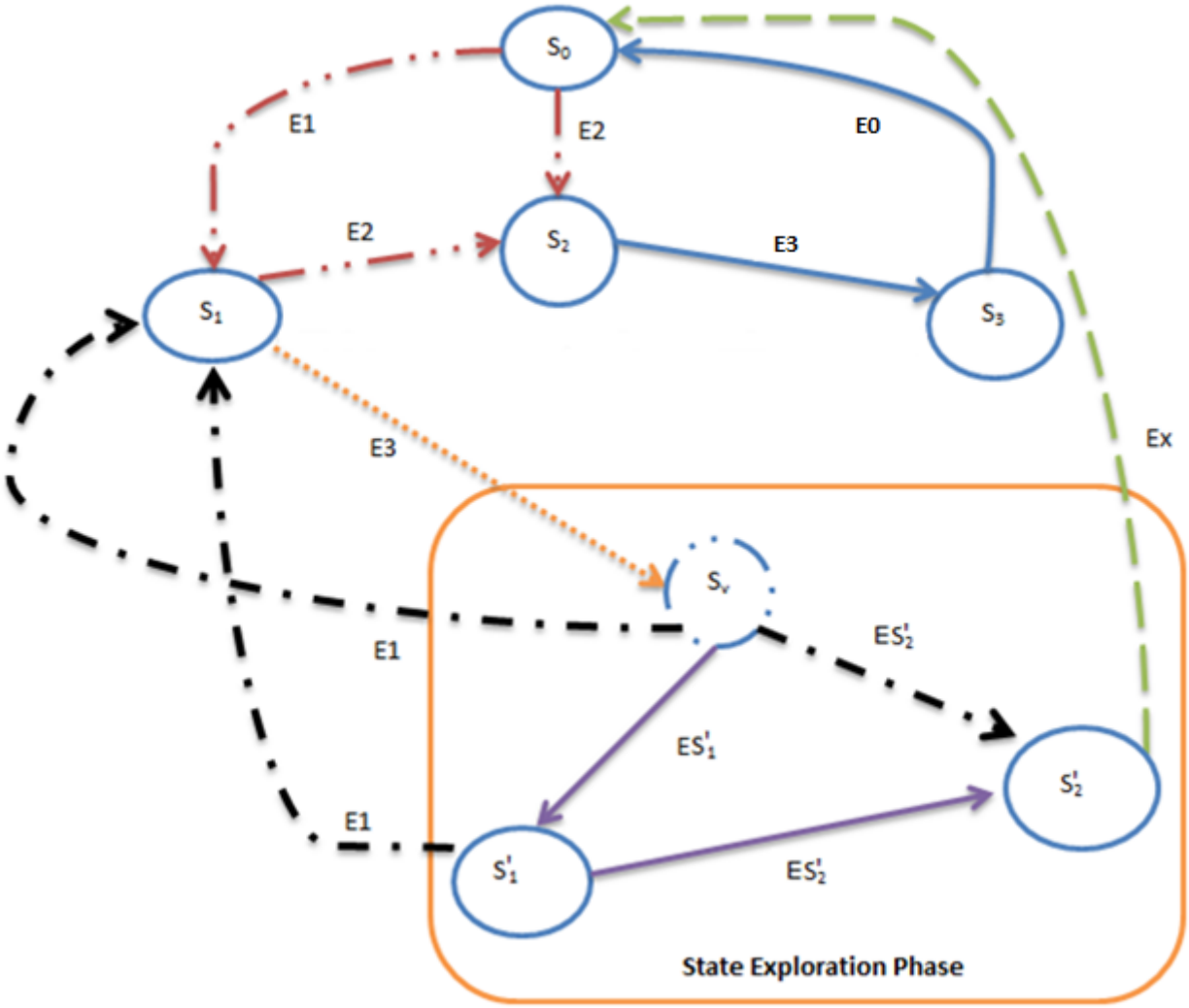

Figure 15: Event execution violation instance 1 (resulting in a new state) during transition exploration phase

In the case when S_v is a new state, the menu crawling strategy saves all the remaining walk segments. The buffer will store the following walk segments: -

1. Walk segment 3: Event List = {E0, E2}, S_{next} = {S_0}, Expected State = {S_2}.
2. Walk segment 4: Event List = {E3, E0, E1}, , S_{next} = {S_0}, Expected State = {S_1}
3. Walk segment 5: Event List = {E2}, S_{next} = {S_1}, Expected State = {S_2}.

Suppose during the state exploration phase, menu crawling strategy discovers some more new states, transitions, menu and self-loop events. In Figure 15, states S'_1 and S'_2 are newly discovered states during state exploration phase. State S_v is the violated state. Events ES'_1 and ES'_2 (at state S'_1) are newly discovered transitions and are represented by known transitions in the application graph G. E1 (at state S'_1 and S_v) and ES'_2 (at state S_v) are newly discovered menu events and are added as predicted edges.

The state exploration phase ends at state S'_2 which is the current state in the crawling process. Instead of recalculating the walk for the performance reasons as discussed above, the menu crawling strategy uses the walk which was saved in the buffer before leaving transition exploration phase. In addition it also saves and keeps track of all the new information discovered during the current state exploration phase in another buffer. However, the walk saved before coming back to state exploration phase will not accommodate for the newly added unexecuted menu events E_1 and ES'_2.

In this scenario also, the menu crawling strategy finds the shortest known path {Ex}, from the current state to the state S_{next} saved in the buffer and continues with the previously defined walk. Hence the new walk segments are:

1. Walk segment 3: Event List = {Ex, E2}, S_{next} = {S_0}, Expected State = {S_2}.
2. Walk segment 4: Event List = {E3, E0, E1}, , S_{next} = {S_0}, Expected State = {S_1}
3. Walk segment 5: Event List = {E2}, S_{next} = {S_1}, Expected State = {S_2}

The menu crawling strategy continues in the same fashion with the transition exploration phase as discussed above.

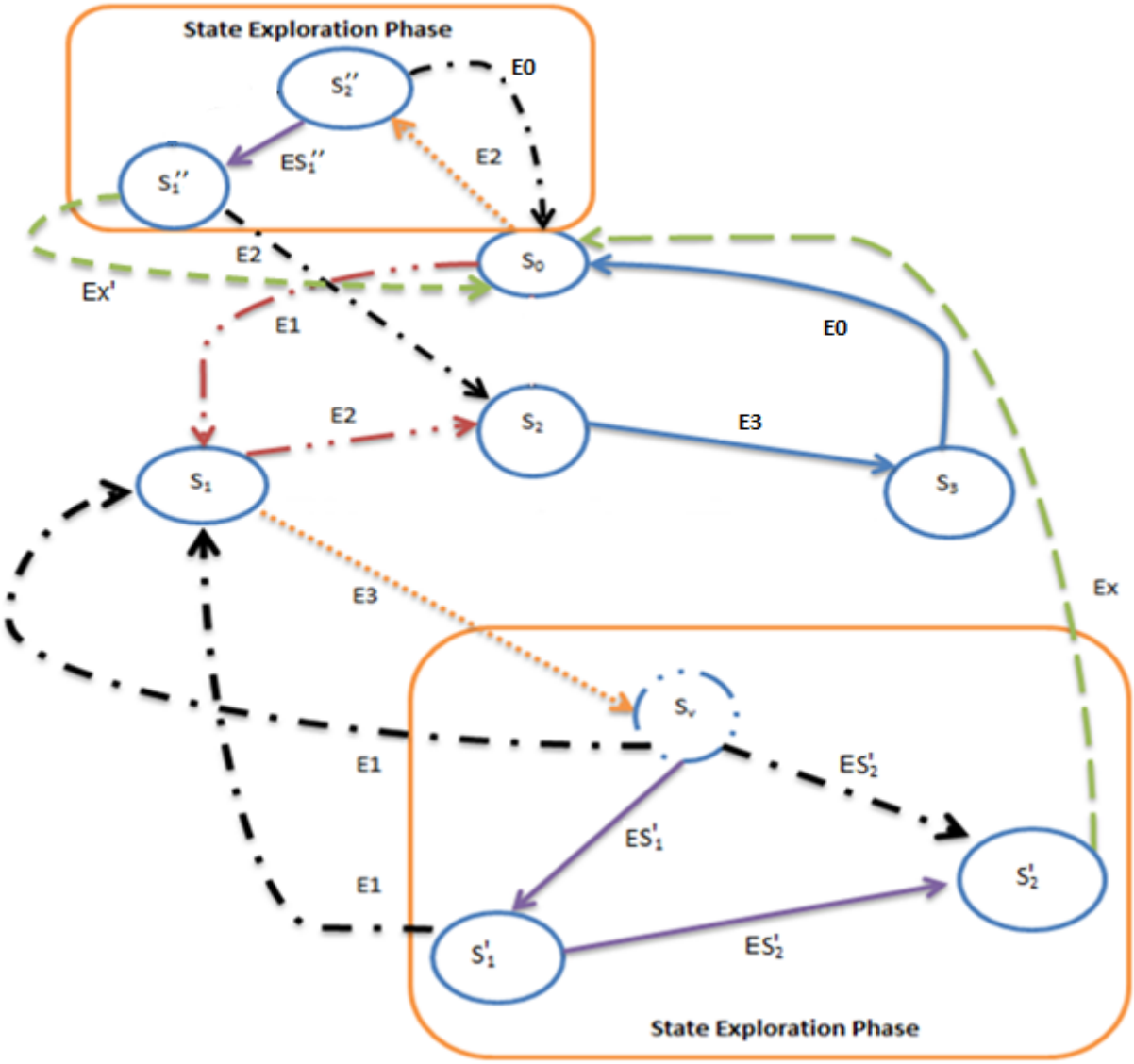

Figure 16: Event execution violation instance 2 (resulting in a new state) during transition exploration phase

Let's suppose the menu crawling strategy encounters another violation while executing walk segment 3 and it arrives at state S''_2 instead of the expected state S_2. The menu crawling strategy will again save all the walk segments after walk segment 3 and will start with the state exploration phase. In the state exploration phase, state S''_1 is a newly discovered state; events E2 (at state S''_1) and E0 are unexecuted menu events and are

represented by the predicted edges. Executed event ES''_1 is added as newly discovered transition to the application graph G.

Once the menu crawling strategy is finished with the state exploration phase, it will again calculate the shortest known path {Ex'} from the current state S''_1, to the state S_{next}. The new walk segments for the transition exploration phase will be:

1. Walk segment 4: Event List = {Ex', E1}, , S_{next} = {S_0}, Expected State = {S_1}
2. Walk segment 5: Event List = {E2}, S_{next} = {S_1}, Expected State = {S_2}

The menu crawling strategy will continue to execute the same walk ignoring any violations encountered till it executes all the walk segments. In case of newly discovered states and transition back to state exploration phase, it will keep track of all the new information discovered and added to the application graph G.

Once the menu crawling strategy finishes with the current transition exploration phase i.e. done executing all the walk segments of WALK_G, it checks for any remaining unexecuted events in the application. If there are still some unexecuted menu and self-loop events in the application, the menu crawling strategy will start with the transition exploration phase again by calling the walk generator function. This time the walk will contain only those events that have been discovered new during the complete transition exploration phase just finished.

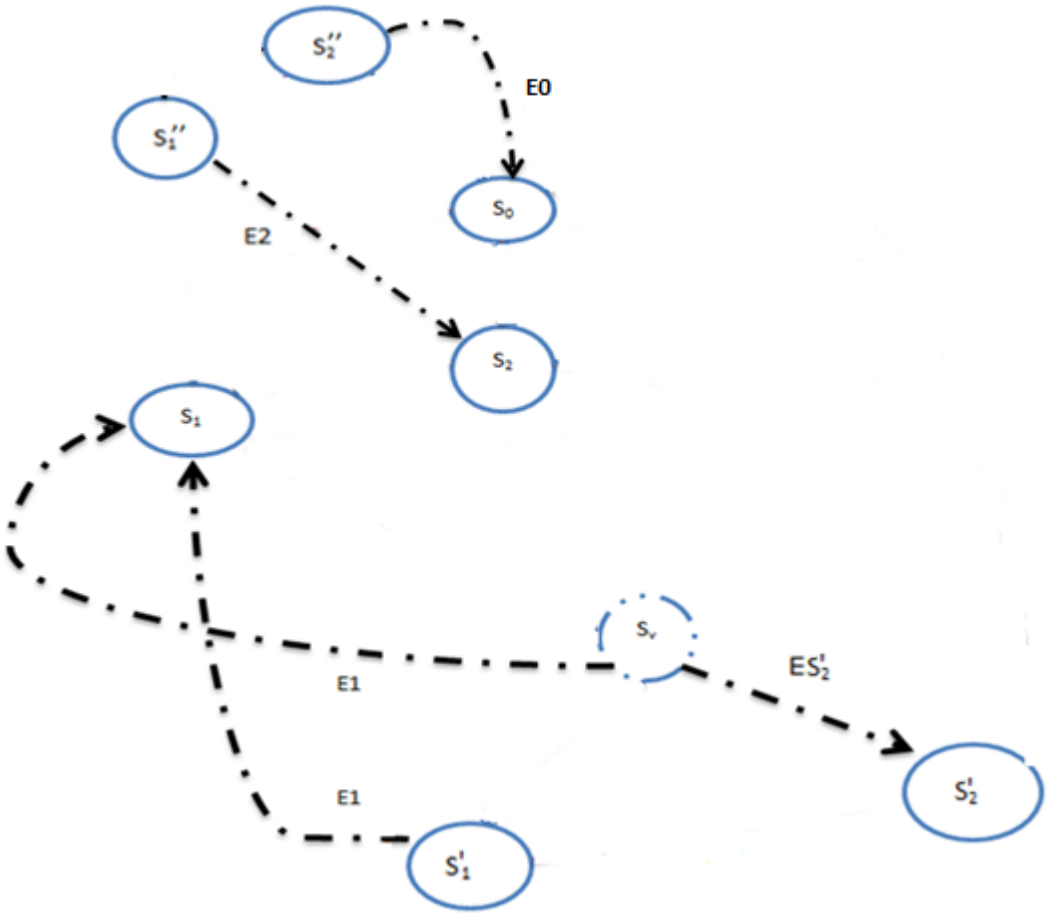

Figure 17: New virtual edges added to the web application graph

Figure 17 shows the new predicted edges added to the application graph G during the last transition exploration phase. The walk generator function will use the know transitions, new predicted edges and resets to define the least cost walk covering these predicted edges at least once.

The menu crawling strategy might alternate between state and transition exploration phase multiple times till it discovers all the states and transitions of the web application.

5.5.4 Walk generator algorithm

5.5.4.1 Overview

The walk generator algorithm is responsible for calculating the walk around all the unexecuted events in the application graph G covering each unexecuted event at least once possibly using other executed event and resets. The walk generator algorithm uses the Chinese postman tour [37] to generate the sequence of events to be returned back to the menu crawling strategy.

5.5.4.2 Chinese postman problem

In arc routing problems (ARPs), the aim is to determine a least-cost traversal of a specified arc subset of a graph, with or without constraints. Such problems occur in a variety of practical contexts and have long been the object of attention by mathematicians and operations researchers.

Before we could explain the algorithm, we should revisit some of standard graph definitions

1. **Eulerian graph and tour**

   In graph theory, a graph is said to be Eulerian if there exists a graph cycle starting and ending at the same vertex such that each edge of the graph is covered exactly once. Such cycle is called “Eulerian cycle” or “Eulerian tour”.

2. **Chinese postman problem or Chinese postman tour (CPT)**

   The Chinese postman problem is to find a shortest closed path or circuit that visits every edge of the graph. The CPT is a tour in a weighted graph, whose tour weight (defined as the sum of weight of the edges traversed by the tour), is minimum. If the graph is Eulerian then

an Euler tour is the optimal solution as it traverses each edge exactly once. The graph may be undirected, directed or mixed. Computationally, the undirected and directed cases are polynomial, whereas the mixed is NP-hard [37].

3. Connected component

In graph theory, a connected component of a directed graph is a sub graph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices.

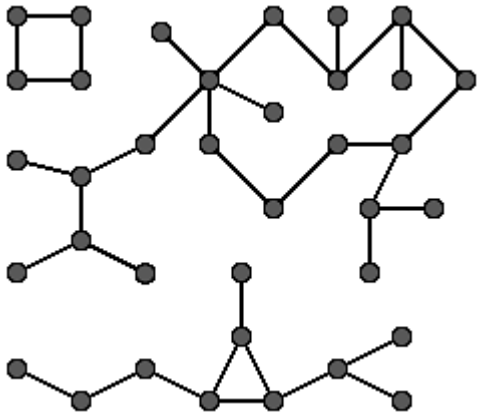

Figure 18: A graph with three connected component

For example, the graph shown in Figure 18 has three connected components. A graph that is itself connected has exactly one connected component, consisting of the whole graph.

4. Strongly connected component

A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex by traversing edges in the direction(s) in which they point. In particular, this means paths in each direction; a path from vertex a to vertex b and also a path from vertex b to vertex a.

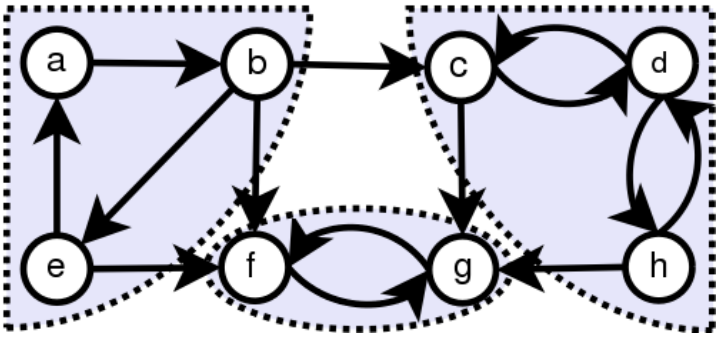

Figure 19: Graph with strongly connected components marked [38]

The strongly connected components of a directed graph G are its maximal strongly connected subgraphs. If each strongly connected component is contracted to a single vertex, the resulting graph is a directed acyclic graph, the condensation of G. Figure 19 shows a graph with three strongly connected components.

Our situation is similar to the Rural Postman Problem [39], where given a graph we want a least cost tour covering only a subset of the edges. The application graph contains known transitions corresponding to executed events and predicted transitions corresponding to unexecuted menu and self-loop events. We need a least cost tour to execute all the remaining unexecuted events.

However due to efficiency reasons which will become evident in Section 5.7 we model our problem as an instance of Chinese postman problem instead of Rural postman problem. In Chinese postman problem, given a graph we want a least cost tour of all the edges. However, the current application graph G also contains edges corresponding to executed events. Hence to model our problem as a Chinese postman problem, we consider only that part of the graph that has not yet been traversed, as a sub-graph G'. This graph contains only the predicted edges i.e. only unexecuted events from the application graph along with the vertices incident upon them. Based on this graph G', defining a Chinese postman tour (CPT) will likely produce an optimal tour covering all the remaining

unexecuted events. We say likely because one preliminary requirement to define CPT on a graph is that the directed graph has to be strongly connected which might require adding additional edges to the graph corresponding to executed events and resets.

5.5.4.3 Tour sequence

To define a CPT on the application sub graph G', the subgraph first has to be made strongly connected if it is not.

In mathematics and computer science, connectivity is a basic concept of computer science which asks for the minimum number of elements (vertices or edges) which needs to be removed to disconnect the remaining vertices from each other. For example a graph is bi-connected if removing two or more vertices makes the graph disconnected and so on. Many good algorithms have been developed for solving such problems [40] [41] [42] [43]. We can turn this idea around and ask questions about how many edges must be added to a graph to make it strongly connected to assist our situation.

One important point to note that in case of web application graphs we can always make the graph strongly connected as we can always reset to the initial state from any state and there always exists a path from initial state to all the discovered states. We need to make the subgraph G' strongly connected possibly by adding edges corresponding to the executed events from the application graph G and using resets.

We have used Eswaran and Tarjan augmentation algorithm [44] [40] to make the sub graph, G' strongly connected. In brief, this paper suggest following steps:

1. Find all the strongly connected components in the graph. This can be obtained easily using Depth-First search [42]

2. Replace all the strongly connected components by single vertex, called graph condensation (G_0).
3. Add edges in the graph G_0 only if there is an edge in the sub graph G' from one strongly connected component to another. This can also be constructed easily using Depth-First search
4. Once G_0 is built, add edges to make G_0 strongly connected which will eventually make G' strongly connected.
5. Use the augmentation algorithm provided in the paper to calculate the set of edges to be added to make the graph G_0 strongly connected. The possible set of edges might include edges corresponding to executed events from application graph, G and resets.

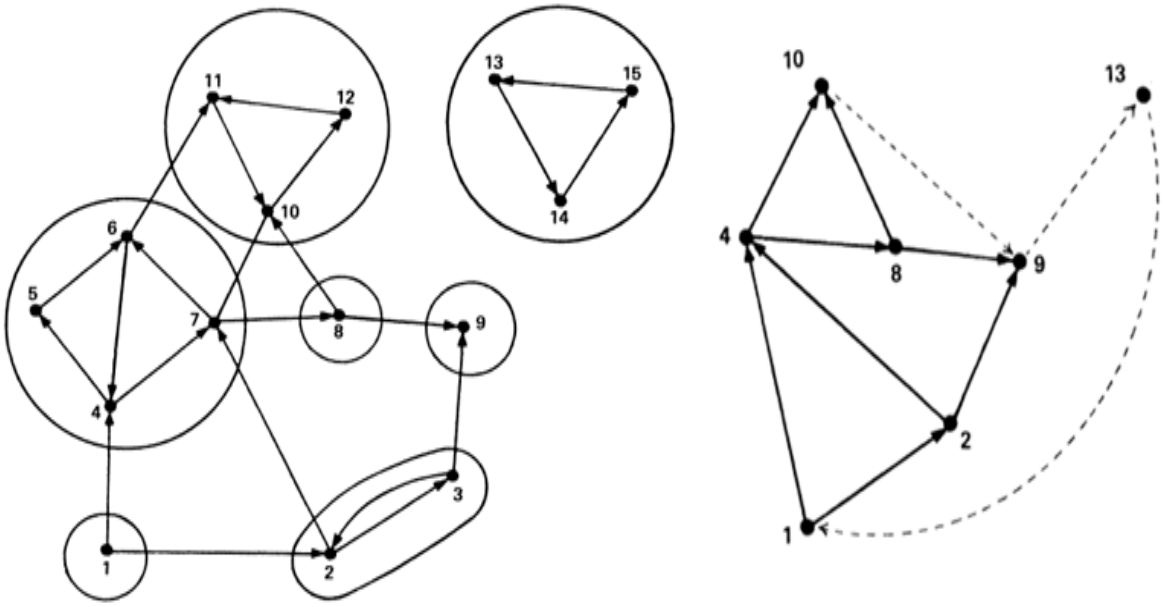

Figure 20: Strongly connected components of a graph and augmentation edges to make the graph strongly connected

Once the graph, G' is strongly connected, we use Harold Thimbleby's algorithm [45] for calculation of the Chines postman tour on the graph, G'. In short the algorithm works as follows:

1. Find all the vertices in the graph that are unbalanced i.e. the number of incoming edges and outgoing edges are not same.
2. Balance the vertices so that each vertex has the same number of incoming and outgoing edges by duplicating some edges. The optimal set of duplicate edges is decided by optimization problem solving techniques. The algorithm uses cycle-cancelling optimization technique [46]
3. Once all the vertices are balanced, define an Euler tour on the graph using standard algorithms [47].

For example, the Figure 20 defines the CPT on a directed graph. Taking each arc to be of equal weight, a least cost open tour is 1, 3, 0, 1, 2, 3, 0, 2, which traverses only 7 arcs.

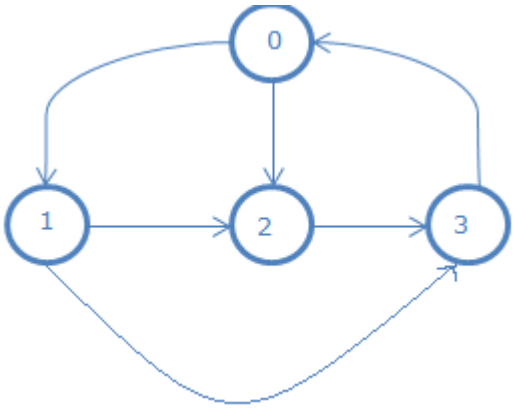

Figure 21: A sample directed graph

Once the complete sequence of events is found, the walk generator function breaks the complete walk into multiple walk segments as described in previous section.

5.6 Other transition exploration heuristics

This section explores heuristics other than the Chinese postman tour to find a better walk of the unexecuted events in the graph.

The CPT defined on the sub-graph G' might not be least cost as to strongly connect the sub-graph G', we add edges corresponding to executed events and resets. Hence we have experimented with other heuristics with an attempt to make the current walk better. However, as the experimental results will suggest, the CPT is still the best solution among all the options explored.

5.6.1 Greedy Algorithm

The Greedy algorithm consists of finding the nearest unexecuted event from the current state and executes it. The algorithm uses a shortest known path algorithm to find the path to the nearest state with an unexecuted event. The algorithm continues till it has executed all the unexecuted events in the graph.

This approach has the advantage of being simple and easy to understand and implement. Also it is easy to adapt in violations of the assumptions since one never calculates the complete path.

In addition, this approach also produces reasonable results the details of which we will discuss in the experimental section.

5.6.2 Rural postman problem and Travelling salesman problem

In graph theory, Chinese postman problem (CPP) requires a tour to traverse each edge of the graph at least once with minimum total cost of the tour. There exists another problem similar to the CPP

and it requires finding a least cost tour to traverse only a subset of the edges at least once. This problem is known as “Rural postman problem” (RPP) and unlike CPP is a NP-Complete problem [39].

This problem is exactly the situation we have in transition exploration phase. After the menu crawling strategy finishes with state exploration phase, the application graph G has edges corresponding to executed events and predicted edges corresponding to unexecuted events. At this point, we need to find a least cost tour in terms of sequence of events to be able to execute all the remaining unexecuted events in the application possibly using resets and executed events. A shortest sequence of events is desired as the cost of crawling RIA is proportional to the number of events executed which we will discuss in detail in Section 6. The rural postman problem is suitable for this problem scenario but it is an NP-Complete problem, no efficient optimal solution exists.

Yet another problem in graph theory which is helpful in our situation is the “Traveling salesman problem” (TSP). It can be stated as follows: Given n cities and the geographical distance between all pairs of these cities, the task is to find the shortest closed tour in which each city is visited exactly once [48]. More formally, the tour length

$$l(\pi) = \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(1)}$$

has to be minimized, where dij is the distance between city i and city j and ∏ a permutation of <1, 2, . . . , n>. Thus, an instance I = <D> is defined by a distance matrix D = (d)ij, and a solution (TSP tour) is a vector ∏ with j = ∏(i) denoting city j to visit at step i. TSP is also a NP-Complete problem.

The TSP has been widely used as a problem for testing new heuristic algorithms and general purpose optimization techniques. As a result, highly effective heuristics have been proposed and developed that are capable of solving TSPs for very large instances of graph [49] [50].

An approach to solve a combinatorial problem to transform the problem into some another combinatorial problem where there exist better heuristics [51] [52].

We have used the Laporte's [51] graph transformation algorithm to convert our problem to an instance of TSP. The reason for transformations is motivated by the availability of extremely efficient heuristics for TSP and easy accessibility to already implemented packages and libraries. In brief the transformation algorithm works as follows:

We consider the graph G = (V, A), where V is the set of vertices and A is the set of arcs in the original graph. Let A' be the set of required arcs i.e. the set of arcs that needs to be traversed at minimum cost and V' be the set of vertices incident upon A'. To transform an arc routing problem on a graph G' = (V', A') into an equivalent vertex routing problem on a graph $G\tilde{} = (V\tilde{}, A\tilde{})$, we proceed as follows:

1. Each vertex $v_i \in V'$ in relabeled v_{ii}
2. Each arc $(v_i, v_j) \in A'$ is replaced by a required vertex v_{ij}
3. The vertex set $V\tilde{}$ is then made up of all vertices v_{ij} defined under point 1.
4. Each vertex pair (V_{ki}, V_{lj}) in the transformed problem defines an arc of $A\tilde{}$ having a cost $C\tilde{}_{ij} = S_{il} + C_{lj}$, where $C\tilde{}_{ii} = 0$, S_{il} is the shortest known path from vertex v_i to vertex v_l and C_{lj} is the cost of the arc (v_l, v_j) in the original graph G.

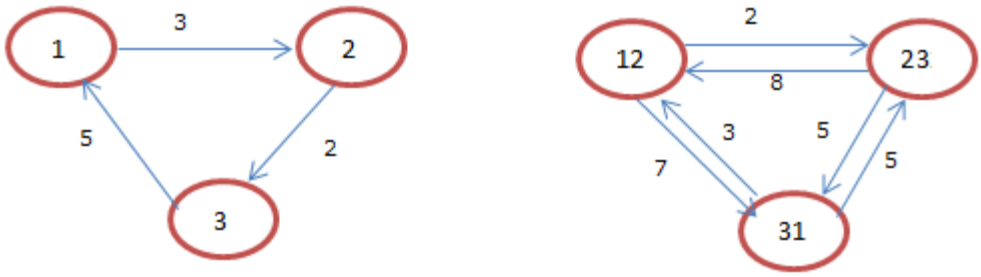

Figure 22: Original and Transformed graph with edge costs

$C^{\sim}_{ij}$	k	i	l	j	S_{il}	C_{lj}	Cost
$V_{12} \rightarrow V_{23}$	1	2	2	3	0	2	2
$V_{23} \rightarrow V_{31}$	2	3	3	1	0	5	5
$V_{31} \rightarrow V_{12}$	3	1	1	2	0	3	3
$V_{23} \rightarrow V_{12}$	2	3	1	2	5	3	8
$V_{12} \rightarrow V_{31}$	1	2	3	1	2	5	7
$V_{31} \rightarrow V_{23}$	3	1	2	3	3	2	5

Figure 23: Arc cost calculation

We have used Keld Helsgaun's implementation of Lin-Kernighman Heuristic (LKH) [53] [54] to help generate a walk for our application graph G. It is one of the current state-of-the-art TSP solvers [55] [56].

The experimental section presents the results and analysis of the three approaches (CPT, Greedy and TSP) to define a walk of all the remaining unexecuted events of the application graph G after the state exploration phase.

6. Experiments and Evaluation of Results

6.1 Overview

We evaluate the performance of the menu crawling strategy in this section and provide comparison against the Depth-First, Breadth-First and Hypercube strategies.

We have used the number of events and resets executed as the metrics to evaluate the performance of different crawling algorithms. “Reset” as mentioned in Section 2.4, is the action of resetting the application back to the initial state by reloading the initial URL. For simplicity we have combined the events and resets executed into a single cost factor expressed as the total number of events. For this purpose, we have calculated the cost of rest in terms of the number of events. The cost of reset if application-dependent and is calculated prior to the crawl by finding the ratio of average time it takes to load the initial page of the application and average execution time of randomly selected events in the application. We have used the number of events as metrics for performance evaluation, since the time to crawl is proportional to the number of events executed.

It is important to mention that we are only interested in two factors to define the efficiency of the crawling algorithms.

1. Total cost to discover all the states of the application

   The cost in terms of the total number of event executions required to discover all the states of the application. This cost is important as it might not be feasible to finish the crawling and we would want to explore as much state as possible within the given runtime of the

algorithm. Hence it is very important to find what percentage of the total state space has been discovered by the crawling algorithm at a given time during the crawl.

2. Total cost to finish the craw

   The second cost we are interested in is the total cost in terms of the number of event executions to finish the crawl i.e. finish executing all the events in the application.

We compare performance of the menu crawling strategy with standard Breadth-First and Depth-First. We compare with these strategies as they are standard graph exploration strategies and most of the published crawling results have used a variation of them as discussed in related works section. It is important to mention that our implementations of Breadth-First and Depth-First strategies are optimized to use the shortest known path to reach the next state to explore. In other words, instead of using systematic resets as required by the strategy to reach a state to be explored next, we use the shortest known path from the current state. The un-optimized versions of the strategies fare much worse.

We also present a comparison of menu strategy against the Hypercube strategy [9]. Since [9] has already proved the efficiency of the hypercube strategy against current state-of-the-art commercial products and other published strategies for crawling RIAs, we have not presented any comparison of the menu crawling strategy with other published results in the field of RIA crawling.

It is important to mention that the menu crawling strategy includes operations such as calculation of walk on the graph for transition exploration, calculation of shortest path etc. These operations are CPU and time intensive esp. the calculation of walk on the graph. This factor is important for comparison and evaluation purpose as a crawling strategy which has a lower total cost in terms of the total number of events but takes more time to finish the crawl is not efficient compared to

another crawling strategy which has higher total cost but takes less time. However, for all the test applications used for the experiments the time required to calculate the important operations of menu crawling strategy such as walk on the graph, is negligible as compared to the total crawling time. Hence, the total crawl time is proportional to the total cost of the crawl, which is used as the parameter for the comparison and evaluation purpose.

We also present, for each application the optimal number of events and resets required to explore all the states of the application. It is important to understand that this optimal value is calculated after the fact, once the model of the application is obtained. In our case the optimal path to visit all states of the application can be found by solving the Asymmetric Traveling Salesman Problem (ATSP) on the graph instance obtained for the application after the crawl. This seems a reasonable strategy as we have modelled the web application as directed graphs with states as nodes and event executions as directed edges. Before calculating the optimal cost, we define the pair wise distance or cost between all pairs of states. The cost of a path from one state to another is the number of events and resets required to go from the first state to another. All state pairwise distance is possible as all states are reachable from the initial state and from every state we can reach the initial state, possibly using reset. We have used an exact ATSP solver [57] to get the optimal path.

6.2 Experimental Setup

For experimental purpose, two states are equivalent if they have the same DOM-ID, which is the state definition calculated from the DOM of the web page and have the same set of enabled events. By enabled events, we mean all the actions that can be detected by Javascript such as onClick. As mentioned in Section 2.1 we use Ayoub et al.`s algorithm [13] for generating the DOM IDs. Also we

consider the complete string of the HTML element that has the registered event and the type of event, as the definition of the registered event in the calculation of the event ID. For example, the id of "*onmouseover"* event will be:

{" This is a changing color text. Try it! " + "onmouseover"}.

For the event IDs, if two events on the same or different page result in the same id then we consider only one of them. It just means that our current ID calculation mechanism cannot distinguish between the two events, but they are actually different events on the page.

State Equivalence Relationship: DOM ID + {Events Enabled}.

In an effort to minimize the impact that the ordering of the events in a given state may have on the performance of crawling, the events in each state are randomly ordered for each crawl. In addition, each application is crawled 10 times with each strategy. The results shown have been obtained by averaging the results of 10 crawls.

IBM® Security AppScan® Enterprise [10] is a product for analysing websites for security and accessibility issues. We have integrated all the experimental code as a module in the AppScan® framework.

The experimental results are based on testing using a machine running Windows 7 with 4GB RAM and a 1.66 GHz Intel Core i7 CPU.

6.3 Test Applications

We have used 2 real and 3 test RIAs for the experimental purpose. We start with an overview of the applications used for experiments.

6.3.1 Clipmarks

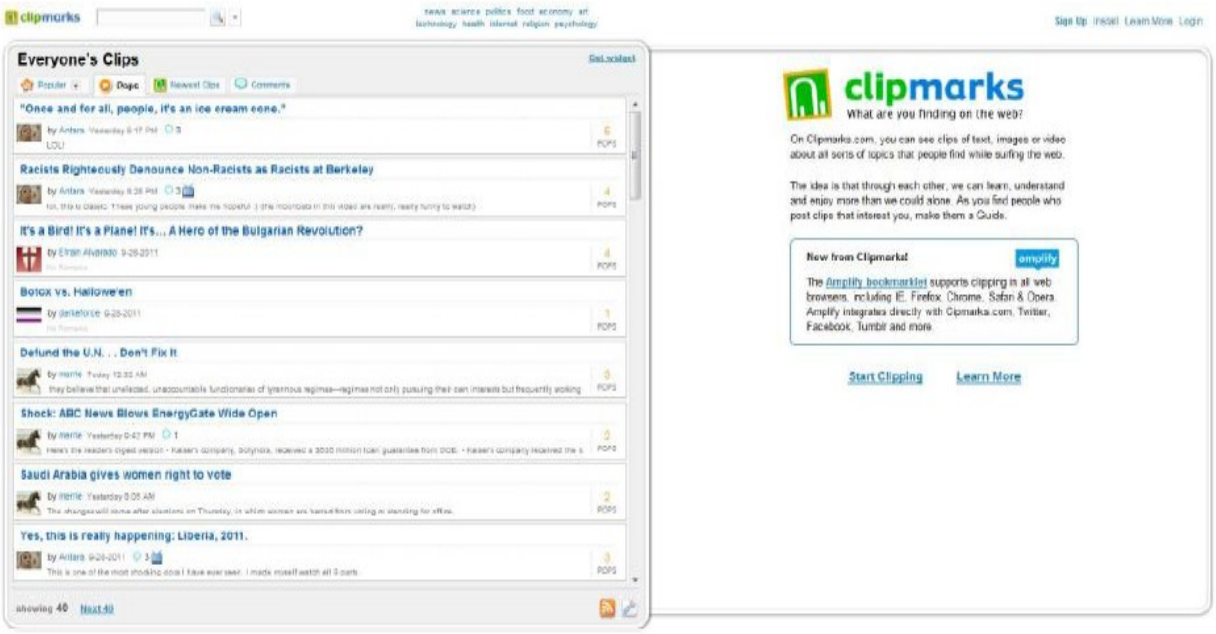

Figure 24: Clipmarks website

The first real application considered is Clipmarks [58] [59]. Clipmarks is a RIA which allows its users to share parts of any webpage (images, text, videos) with other users. Since the live version of the website changes in real time, we have used a partial local copy of the website for the experimental study in order to be consistent between successive crawls and different crawling methods. The basic functionality of the web application can be summarized as follows. The initial page of the application contains a list of clips or items which have been recently voted (popped) by users on the

left hand side of the page. For each clip, we have the title of the clip, the user who shared the clip, the number of votes received by the clip. Clicking on the clip brings detailed information on the right hand side of the web page. In addition, for each clip the user can see the list of users who voted for the clip by clicking on the number of votes, can share the clip on other social networking websites, follow the user who posted the clip and vote for the clip etc. Each of these actions opens/changes the content of the dialog DIV. The states discovered by our crawler in this application are mainly characterized by the content of clip displayed on the right hand side and the contents of the dialog DIVs that are currently open.

This web application is a very good example of how easily a RIA can have huge state space, making it almost impossible for the crawlers to completely crawl the web application in reasonable amount of time. The original web application contains 40 clips in the first page. We have limited the number of clips to restrict the state space of the application. Restricting the number of clips to just 10 clips resulted in 2663 states and more than 200,000 transitions.

For our experimental evaluations, we have limited the number of clips to 3 to be able to finish the crawling in a reasonable time and the reset cost is evaluated to be 18 i.e. resetting the application to the initial state is equivalent to executing 18 events.

6.3.2 Periodic Table

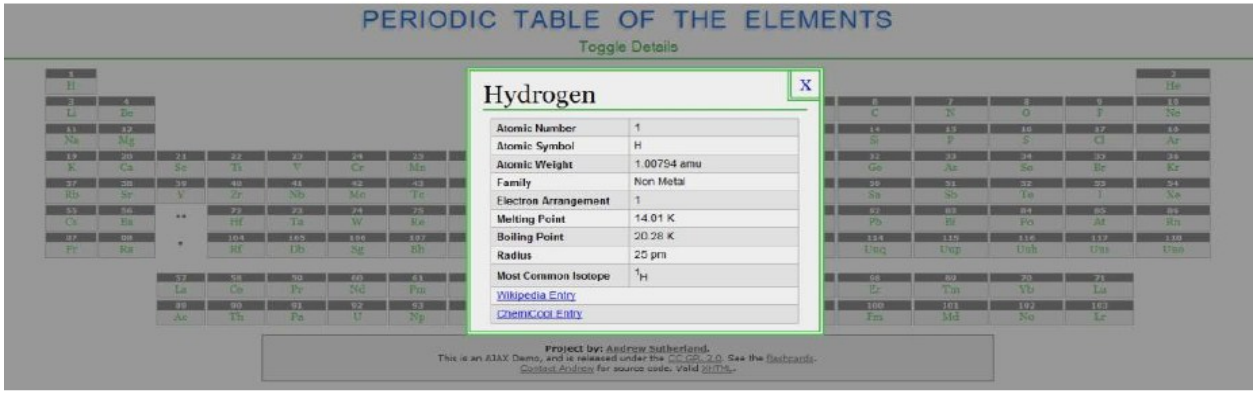

Figure 25: Periodic Table Website

The second real RIA we consider is an AJAX-based periodic table [60] [61]. The periodic table contains the 118 chemical elements in an HTML table. Clicking on each element fetches the information about that element asynchronously from the server and places it in a pop-up (a content display window), thus leading to a different state according to our DOM equivalence function. The pop-up shows information about only one element at a time. All the states except the initial state are reachable from each other, thus forming a complete graph. In addition, there is a “Toggle Details” anchor at the top of each page which switches the style of the current page between two alternate styles. Along with changing the style, the event registered to the anchor also changes and thus resulting in a different state. This result in another copy of the complete graph mentioned earlier linked where the states have the same content as the first graph with differing only in the event registered at the anchor. In total 240 states and 29034 transitions are identified by our crawler with reset cost of 8.

6.3.3 Test RIA website

Figure 26: Test RIA Website

The third application is test application we have developed using AJAX [62]. In TestRIA, we tried to mimic the basic structure of a typical company website (or a personal homepage). Each state of TestRIA contains 5 menu items at the top. They are “Home”, “Services”, “Store”, “Pictures” and “Contact”. Each of the menu items leads to different sections of the website. Home leads to the initial state. The Store and Pictures also contain the common Previous/Next events for navigation purpose such as getting next picture or store item etc. The contents are fetched asynchronously from the server. TestRIA has 39 states and 305 transitions with a reset cost of 2.

6.3.4 Altoro Mutual (Bank) website

Figure 27: Altoro Mutual website

The fourth website is a test application and it mimics the website of a bank [63]. The website contains links to perform administrative tasks, login process, normal banking transactions such as transfer funds, check account balances etc. The local version of the website is an Ajax-fied version of the original website i.e. all the HTML links have been converted to javascript events and the content is fetched asynchronously using Ajax. The website has 45 states and 1210 transitions and a reset cost of 2.

6.3.5 Hypercube website

Figure 28: Hypercube10D website

The last application, called Hypercube10D, is an AJAX test application which has the structure of a 10 dimensional hypercube. Since we are comparing our results against the hypercube strategy too, it seems a reasonable idea to compare our strategy against the hypercube strategy in its best scenario too. The Hypercube10D represents the best case scenario for the hypercube strategy. The application has 1024 states and 5120 transitions and reset cost of 3.

The Hypercube10D web application does not have any transition exploration phase for the menu crawling strategy as none of the events follow the menu hypothesis and hence the menu crawling strategy end up executing all the events in state exploration phase itself.

6.4 Menu state exploration results

As discussed in the above section, we are not only interested in the total cost to discover all the states of the application but also in what percentage of the total state space has been discovered by the crawling algorithm at a given time during the crawl.

For the evaluation purpose we have used a line graph with x-axis representing the number of states discovered and y-axis representing the number of events executed. This graph not only helps us compare the total cost of discovering all the states but also to track the performance during the crawl i.e. the number of states discovered during any point in the crawl. We aim at finding all or a majority of the states of the application as soon as possible. Hence, a crawling algorithm with a lower cost during the crawl (determined by the slope of the line representing the crawling algorithm and the intermediate points representing the cost of discovering a percentage of the states of the application) is a better choice than a crawl with a lower total cost but a worse performance during the crawl.

It is important to note that the menu crawling strategy might take more than one state exploration phase (in the case when the transition exploration phase discovers a new state and returns back to state exploration phase as discussed before) to discover all the states of the application. So the graph does not necessarily represent the result of one state exploration phase but the total cost of discovering all the states. In addition, we represent only the total cost for the optimal strategy.

Also, the graph represents the number of events required to discover the states of the application but the states might not necessarily be discovered in the same order for each crawling strategy.

We also present the total number of events and resets along with the total cost required to discover all the states of the application for each of the crawling algorithm evaluated.

6.4.1 Clipmarks

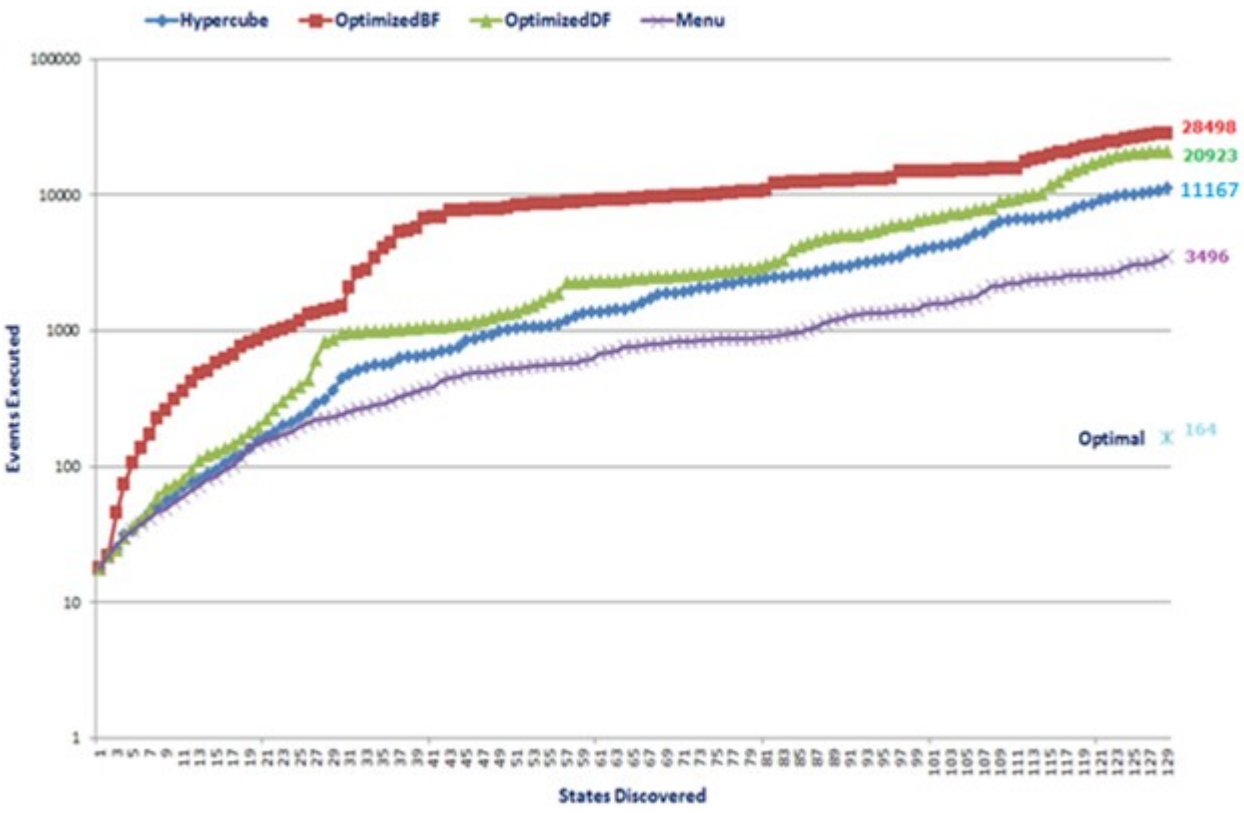

Figure 29: State exploration cost for Clipmarks web application (Logarithmic scale)

Web Application / Crawling Strategies	Clipmarks		
	Events	Resets	Cost
Menu	3,109	22	3,496
Optimized Depth-First	20,111	45	20,923
Optimized Breath-First	12,458	891	28,498
Hypercube	10,918	14	11,167
Optimal	128	2	164

Figure 30: State exploration cost for Clipmarks web application

The menu crawling strategy proves the best among all other crawling strategies for clipmarks despite the fact that a very small percentage of the web application follows the underlying hypothesis. For Clipmarks, only 500 transitions follow the menu hypothesis out of 10580. The menu crawling strategy not only has a lower total cost but also is the best among all other algorithms during the crawl progess.

6.4.2 Periodic table

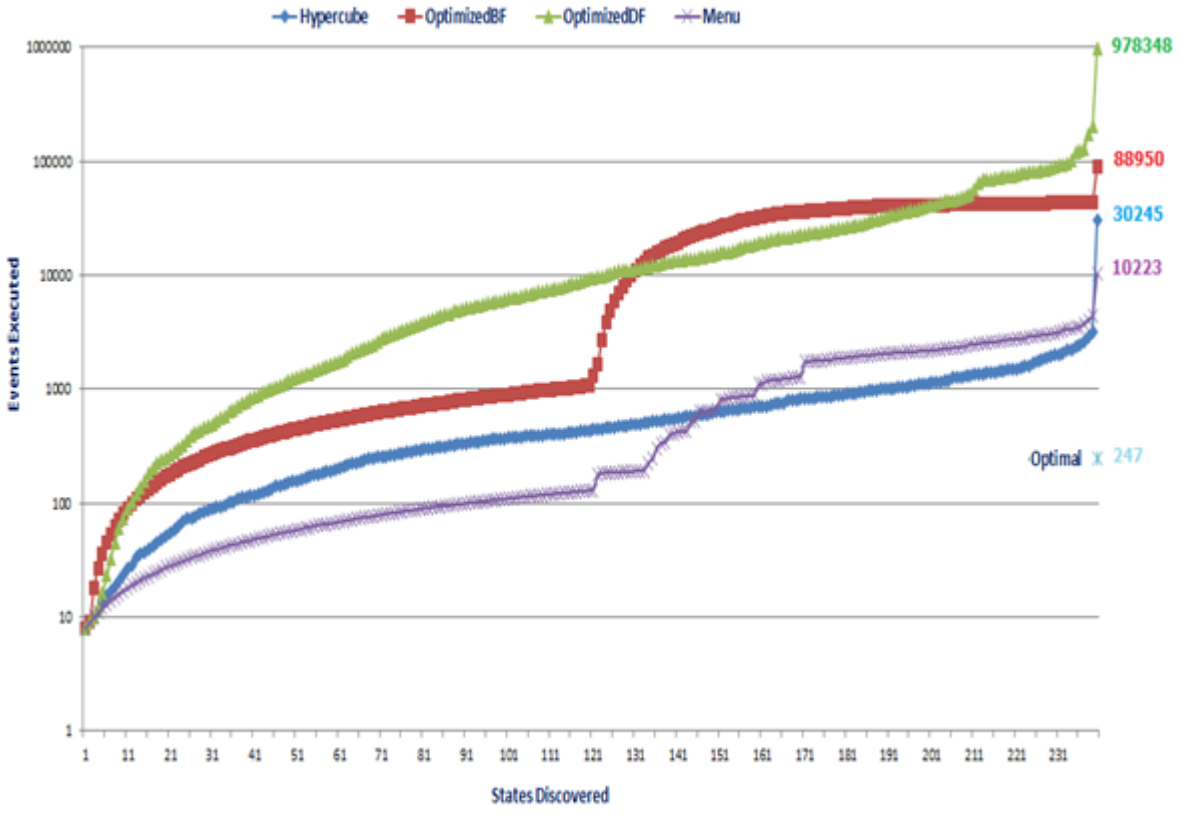

Figure 31: State exploration cost for Periodic Table web application (Logarithmic Scale)

Web Application / Crawling Strategies	Periodic Table		
	Events	Resets	Cost
Menu	10,124	12	10,224
Optimized Depth-First	977,806	68	978,348
Optimized Breath-First	28,914	7,505	88,950
Hypercube	29,684	70	30,245
Optimal	239	1	247

Figure 32: State exploration cost for Periodic Table web application

The menu crawling strategy again proves better than all other crawling strategies with the least total cost of 10244. The second best, the hypercube stratgy has lower cost after the state count 145, although the menu strategy is much better untill, 60% of the state space exploration and also with a lower total overall cost. The Depth-First and Breadth-First strategies perfomance are not even comparable with Depth-First having an staggering cost of 978348.

6.4.3 TestRIA

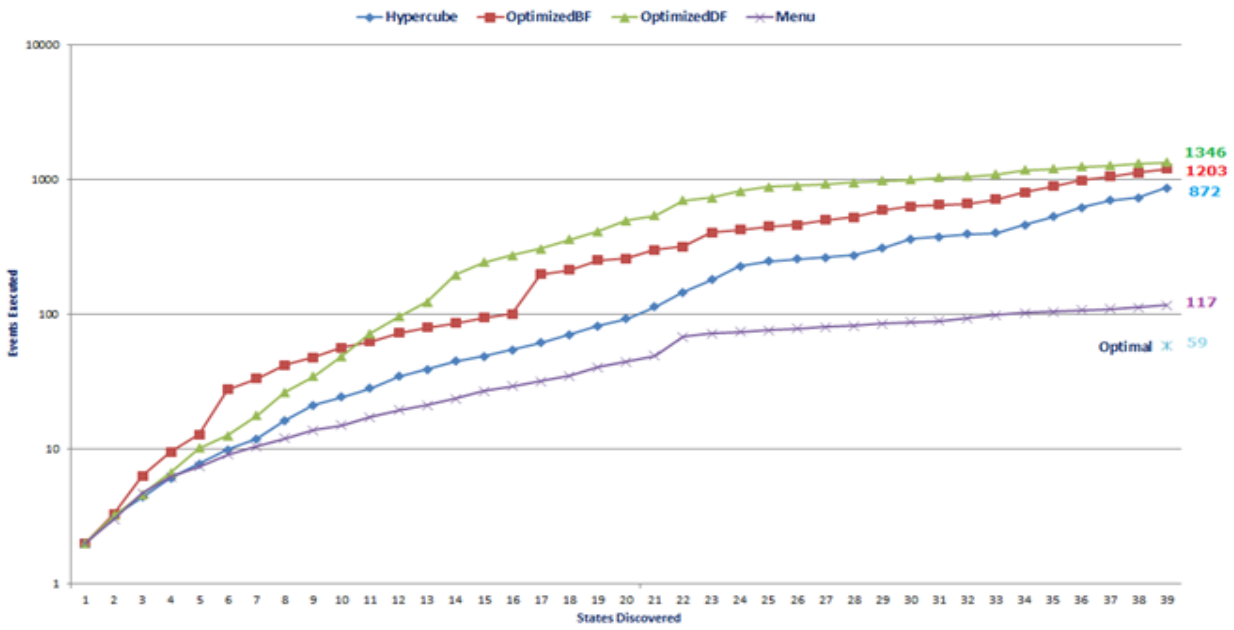

Figure 33: State exploration cost for TestRIA web application (Logarithmic scale)

Web Application / Crawling Strategies	TestRIA		
	Events	Resets	Cost
Menu	115	1	117
Optimized Depth-First	1,343	1	1,345
Optimized Breath-First	1,095	54	1,203
Hypercube	870	1	872
Optimal	57	1	59

Figure 34: State exploration cost for TestRIA web application

The menu crawling strategy provides excellent results for the TestRIA website with not only lowest total cost but also lowest cost during the crawl as seen in the Figure 34. The results are quite encouraging as the

second best, the hypercube strategy is expensive by a factor of eight (appx.), which is very high considering the application has only 39 states and 305 transitions.

This application has been designed to model general-purpose web sites such as a company or personal web site. The results makes the menu crawling strategy very favourable for general purpose crawling.

6.4.4 Altoro Mutual

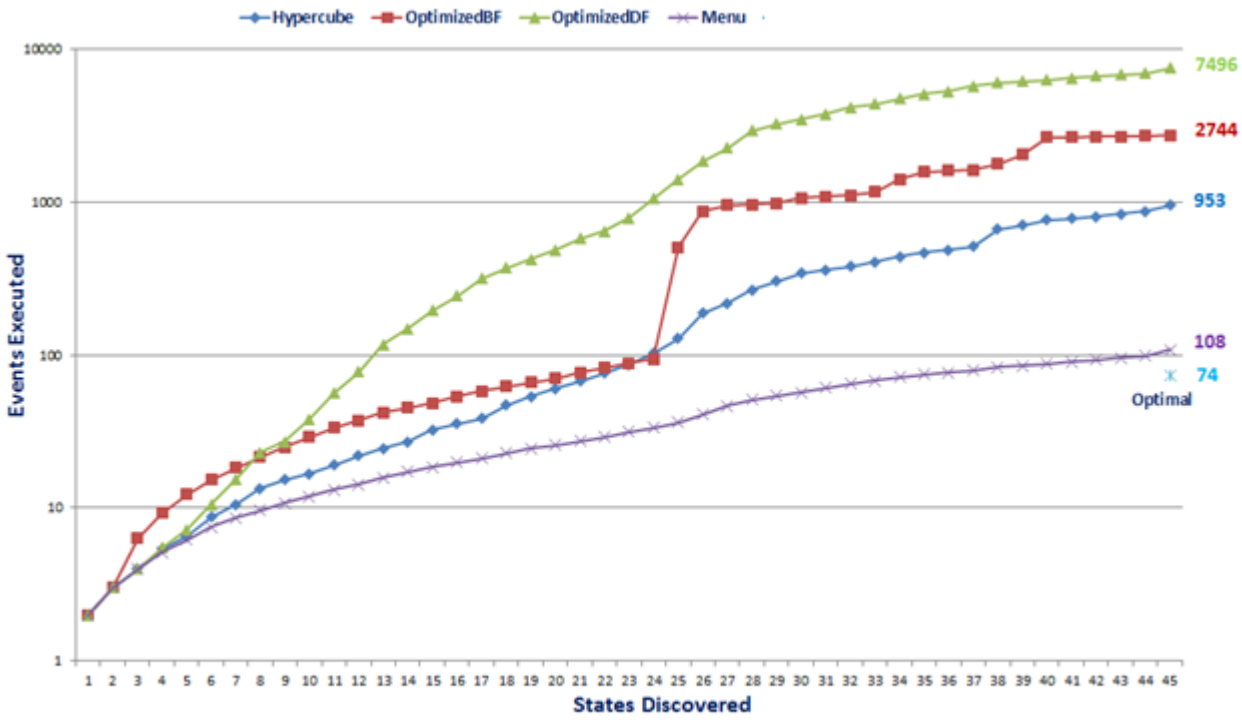

Figure 35: State exploration cost for Altoro Mutual web application (Logarithmic scale)

Web Application / Crawling Strategies	AltoroMutual		
	Events	Resets	Cost
Menu	100	4	108
Optimized Depth-First	7,469	14	7,496
Optimized Breath-First	2,081	332	2,744
Hypercube	903	25	953
Optimal	72	1	74

Figure 36: State exploration cost for Altoro Mutual web application

The Altoro Mutual web application mimics the working of a banking website and is used by the IBM® AppScan® team for tool demonstration purpose. The results are again good with the menu strategy leaving behind other strategies by significant margins not only in the total cost but also during the progress of the crawl. This makes the menu crawling stratgy not only favourable for general-purpose crawling as suggested by the TestRIA results but also a good crawling strategy for web applciation designed to cater specific functionalities such as banking operations. The second best, the hypercube strategy is almost nine times costlier, while the Depth-First and Breadth-First strategies are much farther in the comparison table.

6.4.5 Hypercube10D

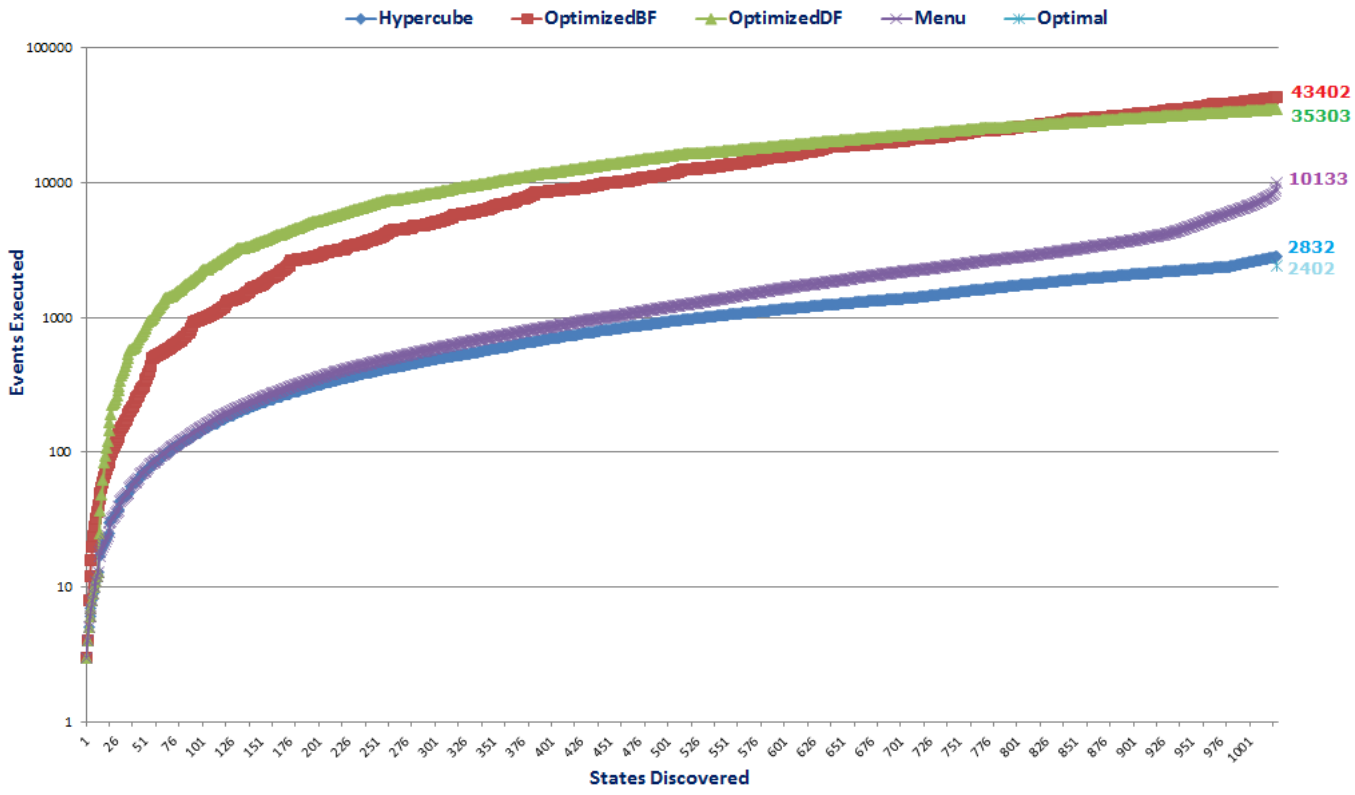

Figure 37: State exploration cost for Hypercube10D web application (Logarithmic scale)

Web Application / Crawling Strategies	Hypercube10D		
	Events	Resets	Cost
Menu	7,171	987	10,133
Optimized Depth-First	23,033	4,090	35,303
Optimized Breath-First	28,069	5,111	43,402
Hypercube	2,076	252	2,832
Optimal	1,646	252	2,402

Figure 38: State exploration cost for Hypercube10D web application

This web site represents the best-case scenario for the hypercube strategy. In contrast, this website presents worst-case scenario for the menu strategy as none of the events follow the menu hypothesis. All the events belong to the non-menu category. Even in this scenario, the menu crawling strategy performs significantly better than the Depth-First and Breadth-First strategies with the differences by a factor of three to four times.

In additon, the menu crawling strategy perfoms very comparable to the hypercube strategy at the beginning of the state space exploration; for instance the menu strategy has a cost of 466 compared to 401 for the hypercube strategy to discover the first 250 states and a cost of 1187 comapred to 929 to discover the first 500 states of the web application.

6.4.6 Summary

State Exploration															
Web Application / Crawling Strategies	Clipmarks			Periodic Table			TestRIA			AltoroMutual			Hypercube10D		
	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost
Menu	3,109	22	3,496	10,124	12	10,224	115	1	117	100	4	108	7,171	987	10,133
Optimized Depth-First	20,111	45	20,923	977,806	68	978,348	1,343	1	1,345	7,469	14	7,496	23,033	4,090	35,303
Optimized Breath-First	12,458	891	28,498	28,914	7,505	88,950	1,095	54	1,203	2,081	332	2,744	28,069	5,111	43,402
Hypercube	10,918	14	11,167	29,684	70	30,245	870	1	872	903	25	953	2,076	252	2,832
Optimal	128	2	164	239	1	247	57	1	59	72	1	74	1,646	252	2,402

Figure 39: State exploration statistics for each crawling strategies

The menu crawling strategy proves to be the best for all the websites, except Hypercube10D. In addition, the total cost of the menu crawling strategy is significantly better than other crawling strategies esp. against the Depth-First and Breadth-First strategies.

6.5 Menu transition exploration results

The transition exploration cost represents the total cost required to finish the crawl. In this section we present the total number of events and resets executed along with the total cost to finish the crawl for each crawling strategy evaluated. The crawling strategy with the lowest total cost is a better crawling strategy. However, the state exploration cost will still be a more dominating factor to determine the best crawling strategy. Hence, we will still prefer a crawling strategy with a lower state exploration cost compared to a crawling strategy with higher state exploration cost but lower transition exploration cost.

5.5.1 Clipmarks

Web Application / Crawling Strategies	Clipmarks		
	Events	**Resets**	**Cost**
Menu	11,678	68	12,893
Optimized Depth-First	20,384	73	21,692
Optimized Breath-First	15,349	931	32,098
Hypercube	11,357	56	12,357

Figure 40: Total cost to crawl Clipmarks web application

The menu crawling strategy proves efficient as compared to the Depth-First and Breadth-First strategies with a slightly higher cost than the hypercube strategy.

5.5.2 Periodic table

Web Application / Crawling Strategies	Periodic Table		
	Events	Resets	Cost
Menu	36,998	235	38,878
Optimized Depth-First	978,095	236	979,983
Optimized Breath-First	64,851	14,633	181,918
Hypercube	29,970	236	31,858

Figure 41: Total cost to crawl Periodic Table web application

The menu crawling strategy performs significantly better than the Depth-First and Breadth-First strategies. The hypercube strategy performs better than the menu strategy in this case; however the state exploration phase was favourable for the menu strategy.

5.5.3 TestRIA

Web Application / Crawling Strategies	TestRIA		
	Events	Resets	Cost
Menu	973	1	975
Optimized Depth-First	1,412	1	1,414
Optimized Breath-First	1,219	55	1,328
Hypercube	994	1	996

Figure 42: Total cost to crawl TestRIA web application

The menu crawling strategy proves the most efficient for the TestRIA application. This result is encouraging as the menu crawling strategy also performed the best for the state exploration and hence it further supports the use of the menu crawling strategy for general-purpose crawling.

5.5.4 Altoro Mutual

Web Application / Crawling Strategies	AltoroMutual		
	Events	Resets	Cost
Menu	2,464	34	2,531
Optimized Depth-First	7,493	34	7,561
Optimized Breath-First	3,074	333	3,739
Hypercube	2,489	34	2,557

Figure 43: Total cost to crawl Altoro Mutual web application

The menu crawling strategy is again the best for the transition exploration for Altoro Mutual web application. It showed similar results for the state exploration, making it a very good crawling strategy for such applications.

5.5.5 Hypercube10D

Web Application / Crawling Strategies	Hypercube10D		
	Events	Resets	Cost
Menu	8,867	1,260	12,647
Optimized Depth-First	23,050	4,098	35,344
Optimized Breath-First	28,159	5,120	43,519
Hypercube	8,860	1,260	12,640

Figure 44: Total cost to crawl Hypercube10D web application

The menu crawling strategy has a significantly better performance than the Depth-First and Breath-First strategies. In addition, it performs almost equivalent to the hypercube strategy with a difference of just seven. This is important result in favour of the menu strategy as this is the best-case scenario for the hypercube strategy and the worst-case scenario for the menu strategy.

5.5.6 Summary

Transition Exploration															
Web Application / Crawling Strategies	Clipmarks			Periodic Table			TestRIA			AltoroMutual			Hypercube10D		
	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost	Events	Resets	Cost
Menu	11,678	68	12,893	36,998	235	38,878	973	1	975	2,464	34	2,531	8,867	1,260	12,647
Optimized Depth-First	20,384	73	21,692	978,095	236	979,983	1,412	1	1,414	7,493	34	7,561	23,050	4,098	35,344
Optimized Breath-First	15,349	931	32,098	64,851	14,633	181,918	1,219	55	1,328	3,074	333	3,739	28,159	5,120	43,519
Hypercube	11,357	56	12,357	29,970	236	31,858	994	1	996	2,489	34	2,557	8,860	1,260	12,640

Figure 45: Transition exploration statistics for each crawling strategies

The menu crawling strategies has significantly better results than the Depth-First and Breadth-First strategies for all the web applications. In addition, it has very comparable results against the hypercube strategy.

6.6 Crawling strategies results evaluation

As an overall evaluation of the results, we can confidently say that menu crawling strategy outperforms the Depth-First and Breadth-First strategies by a significant margin. In addition, it outperforms the hypercube in most of the cases or is comparable in the worst-case. For websites which follow the menu hypothesis to an extent such as TestRIA, Altoro Mutual etc., the menu crawling strategy produces very promising results for both state exploration and transition exploration. Even for websites such as Clipmarks, which has just 500 transitions that follow the menu hypothesis out of 10580, the menu strategy performs significantly better than all other crawling strategies.

The results on the Hypercube10D website produce some very important characteristics about the menu crawling strategy. Since the test website follows the hypercube hypothesis to its entirety, the hypercube produces near optimal results. The menu crawling strategy produces comparable results only differing in later half of the state exploration. This is significant as the website is one of worst-case scenario for menu crawling strategy with 0 transitions that follow the menu hypothesis out of 5120. This makes the menu strategy a good candidate for general purpose crawling flaring well even in the worst-cases.

These are very encouraging results as the menu crawling strategy is simple to understand and implement. The hypercube strategy on the other hand requires strict assumptions about the web application and involves complex algorithms that will probably not be understood by most.

So our conclusion is that the menu crawling strategy is a better choice, much simpler and actually more efficient than the hypercube strategy.

6.7 Transition exploration phase heuristics

6.7.1 Overview

We also experimented with some walk generation heuristics along with Chinese postman tour for menu transition exploration phase as discussed in Section 4.3.5. These heuristics have been used to generate a walk on the graph after the menu crawling strategy finishes with the state exploration phase. The walk generated by each heuristic is used exactly in the same way as the walk generated by the Chinese problem tour was used by the menu crawling strategy during transition exploration phase. We have presented the total cost to finish the crawl for each of the heuristic. We have used a bar graph to represent the evaluation results. The x-axis of the graph represents the heuristic used to generate the walk for the menu transition exploration phase and y-axis represent the total number of events executed.

Along with the total cost of the walk generated (measured in terms of the total number of events) by a heuristic, the calculation time taken by the heuristic to generate the walk is also important. As a walk with a cost 100 events (say) more than another is considered better if the other walk generation algorithm takes 1 (say) minute more than the first one. The motivation behind this is that during that 1 minute of time, the first algorithm would have already finished.

Hence a maximum run time has been specified for the heuristics for each web application. This value is application-dependent and calculated prior to the crawl. It is approximated based on the total duration of the crawl using Chinese postman tour for menu transition exploration phase and the time to execute an event. After the run time is over, the best solution found by the heuristic till that time is used as the walk.

It is very important to mention that the time take by all the heuristics to generate the walk on the graph is negligible as compared to the total crawl time of the web application. Hence, the walk generation time has been ignored for the evaluation purpose and only the total cost of the crawl has been used for comparison.

6.7.2 Experimental Results

6.7.2.1 Clipmarks

Transition Exploration Heuristics		
Chinese Postman Tour	Greedy Strategy	TSP: LKH Heuristic
12,893	12,699	32,117

Figure 46: Total cost to crawl Clipmarks web application

As we can analyse from the graph, the Greedy and the Chinese postman tour performs much better than the LKH heuristic. Among Greedy and Chinese postman, both perform equally well with the Greedy strategy having a slight edge. This difference will become clearer as we go through the data for other websites.

6.7.2.2 Periodic Table

Transition Exploration Heuristics		
Chinese Postman Tour	Greedy Strategy	TSP: LKH Heuristic
19,715	25,392	31,917

Figure 47: Total cost to crawl Periodic Table web application

In total the web application has 240 states and 29034 transitions as identified by our crawler. However, when trying to use the graph transformation and the LKH Heuristic, the weight matrix

containing the transformation information became unmanageable and hence a shorter version of the application with 180 states and 16374 transitions was used to compare the results.

For this web application, the Chinese postman tour defeats all other heuristics by good margin.

6.7.2.3 TestRIA

Transition Exploration Heuristics		
Chinese Postman Tour	Greedy Strategy	TSP: LKH Heuristic
975	993	1,008

Figure 48: Total cost to crawl TestRIA web application

The Chinese postman tour again has the best total cost of 975. Since the web application is small with only 305 transitions, the differences between the approaches are not so significant.

6.7.2.4 Altoro Mutual

Transition Exploration Heuristics		
Chinese Postman Tour	Greedy Strategy	TSP: LKH Heuristic
2,531	3,090	2,696

Figure 49: Total cost to crawl Altoro Mutual web application

As per the statistics, the Chinese postman tour is the best among all the heuristics with the Greedy strategy having worst performance.

6.7.3 Transition exploration heuristics evaluation

Based on the results obtained, we can conclude that the Chinese postman tour provides overall better results than other heuristics.

Surprisingly a simple strategy like the Greedy algorithm has produced very good results. The Greedy algorithm gives a tough comparison however; it worsens as the website size increases as seen in the periodic table website. This is expected as the Greedy algorithm is a simple strategy and follows the same naïve approach for all the websites. Also for most other websites, the final value of the Chinese postman tour is always the best except for Clipmarks; differing only by a small value.

The LKH Heuristics have been mostly developed to work efficiently for symmetric instances. Though it also accepts asymmetric travelling salesman problem data, the results weren't satisfactory as compared to the symmetric counterpart of similar size graphs.

The Chinese postman tour looks like an appropriate fit for the menu crawling strategy.

7. Conclusion and Future Work

Web applications have come a long way both in terms of the adoption to provide information and services and in terms of the technologies to develop them. With the emergence of richer and more advanced technologies, web applications have become more interactive, responsive and user friendly. As the trend to adopt RIAs increases, a better crawling strategy for these useable and sophisticated websites will become a necessity. As seen in the above results, standard crawling strategies such as the Depth-First and Breadth-First strategies performs poorly and as the application size increases, the performance become much worse with depth first search requiring staggering 978384 event executions for the periodic table web application with just 240 states and 29034 transitions. The [9] shows that efficient strategies are possible using the notion of "Model-Based Crawling".

This thesis introduces a novel idea based on the concept of "Model-Based crawling". Our strategy aims at finding most of the states of the application as soon as possible but still eventually finds all the states and transitions of the web application. Experimental results show that this new algorithm performs very well and outperforms the standard crawling strategies by a significant margin. Further, it also outperforms the hypercube strategy in most cases and it performs comparably in the least favorable example, while being very much simpler to understand and to implement. Since hypercube has already been shown to be better than the current state of the art commercial products and other research tools, this is a significant result.

7.1 Summary of contributions

The main contributions of this thesis are the following:

1. **Menu Model**:

   A new meta-model based on the concept of model-based crawling has been introduced.

2. **Event prioritization:**

   A novel technique for prioritizing the execution of events has been introduced to help discover all the states of the application as soon as possible.

3. **Complete crawling strategy:**

   A complete crawling strategy based on menu meta-model has been presented for crawling rich internet applications. The crawling strategy uses the assumptions of menu model and event prioritization to provide an efficient crawling of the application.

4. **Adaptation strategy:**

   We will rarely encounter web applications that will follow the menu model hypothesis to its entirety. A technique to modify the crawling strategy in case of violations has also been presented to produce efficient results.

5. **Prototype crawler:**

   A prototype crawler based on menu meta-model has been presented and is integrated with the AppScan® framework for experimental study.

6. **Crawling strategies comparison and evaluation:**

   The experimental results for the standard Depth-First and Breath-First crawling strategies along with hypercube crawling strategy have been presented. We have also provided the evaluation of menu crawling strategies against these crawling strategies.

7. **Transition exploration heuristics**

We have also presented experimental results and evaluation of three different heuristics for the transition exploration phase to help design final crawling strategy.

7.2 Future work

We have presented some promising results; however, there is a lot more to be explored in the area of RIA crawling. We briefly describe few research directions in this section.

7.2.1 Notion of independent states

A probably future direction in RIA crawling is the identification of independent states. An example of such scenario will be web applications such as Google calendar [64] or Lucid desktop [15] (open source web desktop, or web Operating System). Such applications have contents that are independent of each other. For example, in Google calendar clicking on each day brings up a pop-up window displaying the schedule and events planned for the day. The contents displayed for each day is independent of the contents shown for other day. In such scenarios crawling each pop-up window can be crawled independently of each other. If we do not consider this notion of independent states then we might end up crawling the different permutations of events present in one state with events present in other states which could be theoretically infinite.

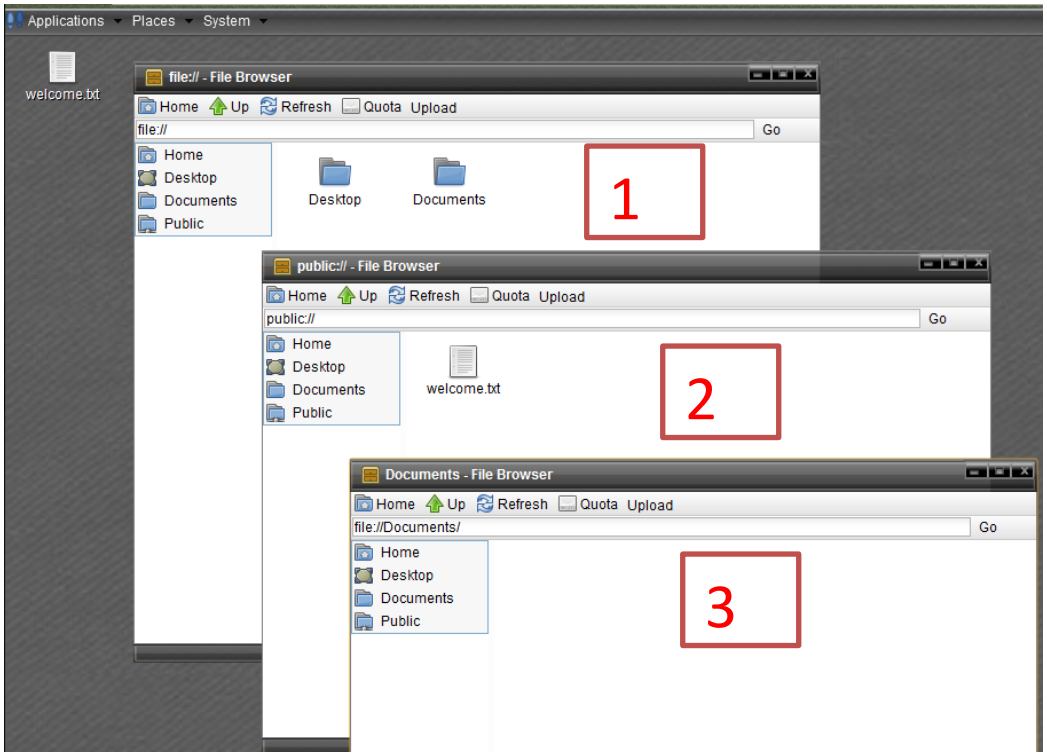

Figure 50: Lucid Desktop [15]screenshot

This situation can be easily visualized in Lucid desktop figure shown above which shows three different windows open. Each window has its own set of enabled events and each window functions independently of each other. In such scenario identification of independent states is critical to be able to completely crawl the web application in reasonable time.

7.2.2 New models

We have already seen that the concept of model-based crawling is able to provide very efficient results in scenarios where the web application follows the underlying hypothesis. We published another crawling strategy called “Probability-Model”.

[11] introduced another novel idea based on the concept of “model-based crawling” where the events present in a state are prioritized in their execution order based on their previous execution

behaviours. The probability model assigns Bayesian probabilities to events based on their chances to discover new states. The probabilities are calculated as the ratio of new states discovered on execution of an event and the total execution count of the event.

A future direction will be to explore more such possibilities of understanding the structures and behaviours of RIAs and designing new hypothesis and corresponding crawling strategies.

7.2.3 Notion of important states and events

A third future direction would be to enhance the model-based crawling with the notion of "important" states and events i.e. some states and events can be configured or determined by the crawling strategy to be more important than others and can be prioritized in the exploration phase. This might help web application security assessment tools to prioritize important parts and features of the web application to be analysed first.

7.2.4 Distributed crawling

Finally, a fourth direction is exploring and using the model-based crawling strategy for distributed crawling. With cloud computing becoming the new norm of the internet world, exploring these opportunities to crawl complex RIAs seems an appropriate step ahead.

References

[1] "JavaScript," W3C: World Wide Web Consortium, [Online]. Available: http://www.w3.org/TR/REC-html40/interact/scripts.html. [Accessed 17 May 2012].

[2] J. Garrett, "Adaptive Path," [Online]. Available: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications. [Accessed 17 May 2012].

[3] X. Li and Y. Xue, "A Survey on Web Application Security," Nashville, TN USA, 2011.

[4] W. W. W. C. (W3C), "Document Object Model (DOM)," 2005. [Online]. Available: http://www.w3.org/DOM/. [Accessed 17 May 2012].

[5] "2010 Data Breach Investigations Report," Verizon, 2010.

[6] T. W. A. S. Consortium, "Web Application Security Statistics," [Online]. Available: http://projects.webappsec.org/w/page/13246989/WebApplicationSecurityStatistics. [Accessed 17 May 2012].

[7] "WhiteHat Website Security Statistics Report," WhiteHat Security, 2011.

[8] J. Bau, E. Bursztein, D. Gupta and J. C. Mitchell, "State of the Art: Automated Black-Box Web Application Vulnerability Testing," in *IEEE Symposium on Security and Privacy*, 2012.

[9] 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.

[10] "Rational AppScan family," IBM, [Online]. Available: http://www-01.ibm.com/software/awdtools/appscan/. [Accessed 17 May 2012].

[11] 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.

[12] S. Choudhary, M. Dincturk, G. Bochmann, G.-V. Jourdan, I. Onut and P. Ionescu, "Solving Some Modeling Challenges when Testing Rich Internet Applications for Security," in *Third International Workshop on Security Testing (SECTEST 2012)*, Montreal, 2012.

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

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

[15] "Lucid Desktop," [Online]. Available: http://www.lucid-desktop.org/. [Accessed 17 May 2012].

[16] S. Brin and L.Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," in *Seventh International World-Wide Web Conference*, Brisbane, Australia, 1998.

[17] "Yahoo! Search," [Online]. Available: http://www.yahoo.com. [Accessed 17 May 2012].

[18] "Bing," [Online]. Available: http://www.bing.com. [Accessed 17 May 2012].

[19] O. C and N. M, "Web Crawling. Foundations and Trends in Information Retrieval," *Foundations and Trends in Information Retrieval,* vol. 4, no. 3, pp. 175-246, 2010.

[20] C. J and G.-M. H, "Estimating frequency of change.," *ACM Transactions on Internet Technology,* vol. 3, no. 3, pp. 256-290, 2003.

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

[22] 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.

[23] R. Matter, *AJAX Crawl: Making AJAX Applications Searchable, Master Thesis, ETH, Zurich, 2008.*

[24] G. Frey, *Indexing AJAX Web Applications, Master Thesis, ETH Zurich, 2007..*

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

[26] A. Mesbah and A. v. Deursen, "Exposing the Hidden Web Induced by AJAX," TUD-SERG Technical Report Series, 2008.

[27] D. Roest, A. Mesbah and A. v. Deursen, "Regression Testing Ajax Applications: Coping with Dynamism," in *Third International Conference on Software Testing, Verification and Validation (ICST 2010)*, 2010.

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

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

[30] A. Mesbah, A. Deursen and S. Lenselink, "Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes," *ACM Transactions on the Web (TWEB),* vol. 6, no. 1, p. a23, 2011.

[31] C. Duda, G. Frey, D. Kossmann and C. Zohu, "AJAXSearch: Crawling, Indexing and Searching Web 2.0 Applications," in *VLDB*, 2008.

[32] 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.

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

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

[35] 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.

[36] E. Dijkstra, "A note on two problems in connection with graphs," *Numerische Mathematik,* vol. 1, p. 269–271, 1959.

[37] H. A. Eiselt, M. Gendreau and G. Laporte., "Arc routing problems, part I: the Chinese postman problem," *Operations Research,* vol. 43, no. 2, pp. 231-242, 1995.

[38] "Strongly Connected Component," [Online]. Available: http://en.wikipedia.org/wiki/File:Scc.png. [Accessed 17 May 2012].

[39] H. A. Eiselt, M. Gendreau and G. Laporte., " Arc routing problems, part II: the Rural postman problem," *Operational Research,* vol. 43, 1995.

[40] K. P. Eswaran and R. E. tarjan, "Augmentation problems," *SIAM Joumal on Computing,* vol. 5, pp. 653-665, 1976.

[41] R. Tarjan, "Testing graph connectivity," in *6th Ann. ACM Symp. on the Theory of Computing*, Seatte, WA, USA, 1974.

[42] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing Journal,* pp. 146-160, 1972.

[43] J. Hopcroft and R. Tarjan, "Efficient algorithms for graph manipulation," *Comm. ACM,* vol. 16,

pp. 372-378, 1973.

[44] S. Raghavan, "A note on Eswaran and Tarjan's algorithm for the strong connectivity augmentation problem," *Operations Research,* vol. 29, pp. 19-26, 2005.

[45] H. Thimbleby, "An algorithm for the directed Chinese Postman Problem (with applications)," Middlesex University School of Computing Science, London, 2000.

[46] A. V. Goldberg and R. E. Tarjan, "Finding Minimum-Cost Circulations by Canceling Negative Cycles," *Journal of the ACM,* vol. 36, no. 4, pp. 873-886, 1989.

[47] "Eulerian path," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Eulerian_path#cite_note-0. [Accessed 17 May 2012].

[48] "Travelling Salesman Problem," Wikipedia, [Online]. Available: www.en.wikipedia.org/wiki/Travelling_salesman_problem. [Accessed 17 May 2012].

[49] G. Gutin and A. Punnen, "Traveling salesman problem and its variations," Dordrecht: Kluwer Academic Publishers, 2002.

[50] C. Nilsson, "Heuristics for the traveling salesman problem," Linköping University, Sweden, 2003.

[51] G. Laporte, "Modeling and solving several classes of arc routing problems as traveling salesman problems," *Comput Opns Res,* vol. 24, pp. 1057-1061, 1997.

[52] M. Blais and G. Laporte, "Exact Solution of the Generalized Routing Problem through Graph Transformations," *The Journal of the Operational Research Society,* vol. 54, no. 8, pp. 906-910, 2003.

[53] K. Helsgaun, " An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic," *European Journal of Operational Research,* vol. 126, no. 1, pp. 106-130, 2000.

[54] K. Helsgaun, " An Effective Implementation of K-opt Moves for the Lin-Kernighan TSP Heuristic.," *DATALOGISKE SKRIFTER (Writings on Computer Science), Roskilde University,* vol. 109, 2006 (Revised November 2007)..

[55] "The Traveling Salesman Problem," [Online]. Available: http://www.tsp.gatech.edu/. [Accessed 17 May 2012].

[56] "TSPLIB," [Online]. Available: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. [Accessed 17 May 2012].

[57] G. Carpento, M. Dell' amico and P. Toth, "Exact solution of large-scale, asymmetric traveling

salesman problems," *ACM Trans. Math. Softw.,* vol. 21, no. 4, 1995.

[58] "Clipmarks," [Online]. Available: http://www.clipmarks.com/ . [Accessed 20 February 2012].

[59] "Clipmarks (Local Version: replicated on: 2011/3)," [Online]. Available: http://ssrg.eecs.uottawa.ca/clipmarks/. [Accessed 17 May 2012].

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

[61] "Periodic Table (Local version:replicated on: 2011/3)," [Online]. Available: http://ssrg.eecs.uottawa.ca/periodic/. [Accessed 17 May 2012].

[62] "TestRIA," [Online]. Available: http://ssrg.eecs.uottawa.ca/TestRIA/. [Accessed 17 May 2012].

[63] "Altoro Mutual," [Online]. Available: http://altoromutual.com. [Accessed 17 May 2012].

[64] "Google Calendar," [Online]. Available: http://calendar.google.com. [Accessed 17 May 2012].