A Strategy for Efficient Crawling of Rich Internet Applications

Kamara Benjamin

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 Information Technology and Engineering
Faculty of Engineering
University of Ottawa

Abstract

This thesis studies the problem of crawling rich internet applications. These applications are built using advanced web technologies which allow them to be more dynamic and enable better user experiences. In recent years, the popularity and importance of web applications has continually increased and they are now very commonly used to complete essential tasks such as financial transactions. As a result, the need to crawl these applications goes beyond the desire to index content for search. For example, applications also need to be analyzed in order to detect security vulnerabilities and assess accessibility. In this thesis, the challenges involved with crawling rich internet applications are discussed and an efficient strategy for crawling these applications is presented. We also use this strategy to develop a prototype tool for crawling AJAX-based applications.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First, I would like to thank my supervisors, Dr. Gregor v. Bochmann and Dr. Guy-Vincent Jourdan. This thesis would not be possible without their knowledge, encouragement, and guidance. They have always been receptive to my ideas, and their constructive criticisms have facilitated this work. I would also like to thank the other members of the Software Security Research group at the University of Ottawa, including Dr. Vio Onut (IBM) and my colleague, Emre Dinçtürk. The exchange of ideas between members of this group has been essential to this research.

In addition, this project would not be possible without the support of IBM and the National Science and Engineering Research Council (NSERC) of Canada. I am also grateful to the AppScan team at IBM. Dr. Onut and his colleagues have provided a wealth of information which is important to this research.

I would also like to express gratitude to my loving parents, Peter and Tecla Benjamin. They have been the greatest influences in my life and it is impossible to capture their love and support in only a few words. In addition, I would like to thank a few of my other major supporters. I am grateful to my brother and sister, Kayode and Kaisha Benjamin, and my girlfriend, Edith Brumant, for the encouragement that they have given me.

1 Introduction

Over the last three decades, the internet has become an essential part of everyday life. Users rely on the internet for tasks related to communication, information, and commerce among others. In addition, the popularity of web-based applications has exploded over the last several years with hundreds of millions of people having access to and making use of the internet. With such popularity, the importance of web applications has been magnified as they store important data, have become sources of valuable information, and provide important services for many users. As a result, there is a need to be able to crawl web applications (automatically discover all states of applications) and process them in various ways.

This work aims to advance the ability to crawl web applications, particularly rich internet applications, which are built using technologies such as AJAX [1], Flash [2], Silverlight [3], and Flex [4]. These technologies cause modern web applications to be quite different from traditional web applications and render existing crawling techniques ineffective. Before attempting to crawl such applications, the challenges which will be faced need to be identified. There is also a need to produce a strategy which allows efficient crawling of such applications. The goal is to build an initial prototype crawling tool (using this strategy) which is able to crawl AJAX-based applications producing a model which captures the different states of an application as well as the actions (transitions) which cause the application to move from one state to another.

A paper [5] has been published which covers a portion of this research. In addition, five patents stemming from this work are in the filing process at IBM. Two of these cover the work presented in Chapter 4. The other three cover the work presented in Chapter 5.

1.1 Traditional Web Applications

Web applications have traditionally consisted of a collection of static documents encoded as HTML. Each of these documents has a URI (Universal Resource Identifier). This identifier includes information about the host of the document, the name of the document, and the protocol used to access the document [6]. HTML (HyperText Markup Language) [7] is a markup language which allows developers to produce documents (pages) which are consistently viewable in various sizes within web browsers across various platforms. It provides semantics which allow developers to define the structure of a web document and denote components such as headings, text, tables, and pictures.

Traditional web applications utilized different documents to provide different sets of information or functionality to the user. In order to move from one document to another, there is a need for a synchronous HTTP request (which could be triggered by some action performed in the existing page) to be made to the server which stores this next document. There is then a response from the server which contains this document. The document is then loaded, providing the user with access to the next page in the application. One drawback of this technique for updating the page is that moving from one document to another, the entire page has to be reloaded. This results in

user activity being suspended until the new page is loaded. Depending on the size of the document and the speed of the internet connection being used, this period of suspended activity could be anywhere from a fraction of a second to a few seconds.

1.2 JavaScript

More recently, the documents which comprise web applications have included client-side scripts. JavaScript is one of the languages in which scripts may be written. Scripts contain functions which perform some programming logic. When a user interacts with a web application, certain actions may trigger events (for example, an *onclick* event) which call these functions, causing client-side updates. This is made possible by the DOM (Document Object Model) which defines the structure and content of a document [9] and allows access and updates to it. The inclusion of JavaScript means that web applications should be considered differently since for a given URL, they can exist at different states.

1.3 Advanced Web Technologies

One advancement of web application technology is AJAX (Asynchronous JavaScript and XML), which allows additional content to be retrieved from the server without requiring the page to be completely reloaded. This results in applications that are much more responsive and dynamic as they continuously change and feature updated information without having to interrupt the user's experience. This is all accomplished through the ability to send asynchronous requests to the

server, and process the responses in the background. The use of AJAX means that there is the potential for a given page to have even more states because these asynchronous updates allow the document to be modified to include both additional HTML and JavaScript code which is retrieved from the web server.

AJAX is one example of a technology which utilizes asynchronous communication with the server. There are additional technologies such as Adobe Flash, Microsoft Silverlight, and Adobe Flex which also do the same.

1.4 Crawling Modern Applications

Web application technology has seen significant advancement, giving browser-based applications ever more capabilities and allowing them to approach the functionality and user experience of desktop applications. Technologies such as AJAX, Flash, Silverlight, and Flex, along with the current browsers which support them, enable these improvements in the capability and functionality of web applications. As a result of these improvements, tasks which were once limited to desktop applications, such as word processing and image editing, are now being completed via web applications.

Also, with the rise in the popularity of cloud-based services, people now make use of the internet to store larger amounts of sensitive data. It is now common to have pictures, business documents, and even health records stored on servers which are under the control of other entities. Web

applications are then used to access and modify this data. In addition, it is now very common to complete many tasks that usually require a high level of confidentiality via web applications. Passport applications, financial transactions, and school assignments are all completed online.

Given the ubiquity of the internet and the vastness of the data that is stored and exchanged via web applications, there is a need to be able to automatically uncover all states of an application in order to index information (for example, for search) or to analyze an application for a variety of reasons, including the detection of security vulnerabilities, and the assessment of accessibility. Most current crawlers were developed for traditional websites and are very limited in their ability to explore applications which may be updated client-side via scripts. Therefore, they are unable to uncover all states of such applications, compromising the ability to analyze and index these applications.

1.5 Motivation

This research aims to make advancements in the crawling of web applications that feature advanced web technologies. There is a need for better crawlers that are able to discover all states of an application which features asynchronous updates to the page. The long-term goal is to produce a crawler that is capable of crawling rich internet applications efficiently. However, the initial prototype crawling tool will focus on AJAX-based web applications.

This research is funded in part by IBM [10]. IBM produces a line of products, called Rational AppScan [11], that are capable of crawling applications and performing tests on them including automated evaluations for security vulnerabilities and accessibility issues. These products typically perform thousands of different tests on each page of an application. The Rational AppScan products will directly benefit from the results of this research.

1.6 List of Contributions

The following list describes the contributions of this work:

1. A compiled list of challenges which will need to be addressed over time in order to produce a crawling tool that is able to crawl rich internet applications.
2. An initial strategy for crawling rich internet applications that conform to the structure of a hypercube in a minimum number of paths, transitions, and resets.
3. A technique for modifying the initial strategy when, inevitably, there are web applications which do not follow a hypercube structure.
4. A method of determining whether to execute events or follow URLs when crawling web applications. A method of determining *which* event to execute is also provided.
5. A complete strategy for crawling web applications which consist of both asynchronous and synchronous requests to the server.
6. The identification of a class of states in rich internet applications that we call intermediate states.

7. A description of some factors which should be taken into account when determining state equivalence.
8. A technique for automatically excluding the irrelevant portions of the DOM when determining state equivalence.
9. An initial prototype tool which is used to crawl test AJAX applications and allows for some comparison between different crawling strategies.

1.7 Organization of the Thesis

This document is organized as follows: Chapter 2 provides an overview of work which is related to this research. Chapter 3 discusses the challenges which will be faced in working towards a solution for improved crawling. Chapter 4 discusses some ideas about state equivalence and describes some factors which should be taken into account when determining state equivalence. Here, we also discuss how some irrelevant portions of the page may cause complications when determining state equivalence and provide a method for automatically excluding these portions from the equivalence calculations. In Chapter 5, a complete strategy for crawling modern web applications is presented. In Chapter 6, the implementation of the prototype crawling tool (which is based on the event-based strategy described in Chapter 5) is described. In Chapter 7, the results obtained from some initial testing conducted with the prototype crawling tool are presented and discussed. Finally, the document ends with a conclusion and discussion of future work in Chapter 8.

2 Related Work

2.1 Crawling AJAX Applications

Significant progress has been made in the area of web crawling. Notably, this includes [12], which is an introduction to Google [13]. There are also many other examples of crawlers, some of which drive other leading search engines such as Yahoo! [14] and Bing [15]. Others enable page discovery for processing (security scanning for example) in products like AppScan [11]. However, these crawlers were designed to work with traditional web applications. The authors of [16] describe the general process of crawling such applications. First, a URL is used to load the first page. The page is then parsed to harvest all URLs. Then, the first two steps are repeated for any URLs that have not been previously encountered. Crawling ends when there are no URLs for which these steps have not been completed.

There are a limited number of papers published which specifically deal with the task of crawling AJAX applications. This is not particularly surprising given the relatively short history of AJAX and AJAX-based applications, with the term AJAX being less than a decade old [1]. Additionally, most research in the area of crawling aims to improve the ability to crawl websites for the purpose of search and indexing. Research to improve the crawling of applications which provide task-based functionality (such as financial transactions) is very limited and search engines do not have much motivation to discover all the states of an application which lets users

transfer funds from one back account to another. Only recently have search engines, such as Google, shown an interest in crawling applications of this nature, such as social networking applications. However, even in this case, the goal is to index the content which is produced by these applications.

The major publications related to AJAX crawling come from the work done at ETH Zurich [17] (in [18], [19], and [20]) as well as from research done in connection with the CrawlJax [21] tool. In [18], [19], and [20] the research is focused on crawling for the purpose of indexing and search. They are primarily concerned with their ability to crawl AJAX applications, index content, and process queries. In the case of CrawlJax, the tool is positioned to allow crawling of AJAX applications and the conversion of such applications to ones which simply consist of static HTML pages with hyperlinks linking them. In [22], the aim is to make those applications fully accessible to search engines which are not AJAX-friendly. Additionally, the authors of [23] focus on regression testing of AJAX applications while the authors of [24] look at security testing and the authors of [25] look at user interface testing.

In both [18] and [25], AJAX applications are modeled using transition graphs. Logically, nodes represent the client-side state of the application, which is determined by the structure and content of the page at a given time. Additionally, edges indicate transitions, which occur due to the execution of some event (which is enabled in the current state) and may cause the page to be altered and result in the arrival at another state. In [26], graphs are also used to represent AJAX applications. However, model creation is accomplished using both dynamic analysis and static

analysis of code. In addition, the states of the graph are abstracted. For example, in a site such as an e-commerce website, a state could be determined by values such as the current number of items in the shopping basket and the current total cost of those items. Transitions would then occur when items are added or removed from the shopping basket causing a change to those values.

A few papers recognize the importance of being able to differentiate between states given that multiple states can exists with the same URL. The issue of identifying duplicates is not limited to rich internet applications. Traditional applications often have duplicate URLs [27] in which different URLs correspond to separate pages which are almost identical. In terms of modern applications, Matter [18] identifies that in AJAX web applications, there is usually a portion of the page that is "stable" (for instance, contains menu items which do not change much from state to state) and a portion that is more dynamic. Duda et al. [19] determine whether or not states are the same by "comparing the hash value of the fully serialized DOM". However, they do admit that this means that only in the case of identical states will duplicates be identified. This method of identifying duplicates is too strict and may lead to difficulties as we will see in Chapter 3. Similarly, Mesbah and van Deursen [22] compute a "hashcode" which is used to compare states. They also mention another technique for comparing states, which makes use of the Levenshtein [56] method (which determines the minimum number of operations necessary to convert one string to another) to calculate an "edit distance" between two states. If this distance is found to be within a particular threshold (0.0 - 1.0 is used), the two states are considered the same. The use of such an approach allows for some differences in two states that are considered equivalent. In [26], another technique is used, called simhash, which utilizes a hashed value to determine whether or

not two documents/states are equivalent. This is accomplished by dividing a document into a set of weighted “features”. This data is used with simhash to produce a fingerprint of the document. Fingerprints can be compared to identify duplicate documents based on the similarity in hash values.

These approaches do not evaluate state equivalence (as discussed in Section 3.3). They evaluate the distance between states or the differences between states. However, these relations are not transitive. In addition, these approaches appear to be specific to crawling for the purpose of indexing and search, or more specifically, for identifying when the content of the page differs. They do not take into consideration how the purpose of the crawl affects state equivalence. These considerations are discussed in Chapter 4.

In terms of crawling strategy, Matter [18] makes use of a breadth-first crawl. In addition to this, an effort is made to reduce the amount of AJAX calls which are made during the crawl. To accomplish this, whenever a specific AJAX call is made for the first time, the response from the server is cached. In the future, if there is an event which uses the same function and parameter(s) for the execution of an event, rather than actually making an AJAX call, the resulting content is retrieved from the cache. This approach does not seem to take into account the fact that calling the same method, with the same parameters while in a different state could lead to different results. This could be due to factors such as a change in the server state. In a different approach, the authors of [22] use a depth first crawl. This is combined with a variable which is used to limit the maximum depth explored.

Looking at existing research, it is found that there are various tools which can be used to simulate the actions of a browser thus enabling crawling. Matter [18] and Frey [19] make use of the Corba Toolkit [29], which is able to load HTML pages and provides DOM retrieval. The Rhino [30] framework is also used in order to process JavaScript (which is necessary for event execution). In [22], CrawlJax's browser functionality is made possible through the use of Mozilla XUL Runner [31]. This is combined with the use of Webclient [32] for access to the DOM. In [33], Watir [34] is used to perform the role of the bowser (it is used to programmatically manipulate Internet Explorer [35]) and interacts with JavaScript using rbNarcissus [36].

2.2 Model Based Testing

Given that the problem involves the generation of a model which represents a specific web application, and that the model produced could potentially be used for various purposes, including testing scenarios such as security testing, and accessibility testing, it is practical to review some of the existing research on model-based testing.

Model based testing [37] is a testing technique in which a model is created to represent the system under test. That model is then utilized in the testing of the system. It can be used to create test cases to ensure that the system operates as expected.

Fantinato and Jingo [38] point out that the activities made possible by a model depend heavily on the quality of that model. This reinforces the need to construct a model that accurately represents the application. Hierons [39] also mentions a few modeling techniques which are suitable for model-based testing including Finite State Machines (FSMs), Statecharts, and Petri Nets [40]. The authors of [38] reinforce the popularity of FSMs and their suitability for modeling systems that include various states. The authors of [41] give some examples of the different ways in which FSMs have been used, including the modeling of systems in the areas of "sequential circuits, software development and communications protocols".

Lee and Yannakakis [42] discuss testing problems in which "we have a machine about which we lack some information". In order to get this information, the machine is provided with inputs and the outputs are observed. They also describe two types testing problems. One consists of determining the current state within a finite state machine. The other involves conformance testing, where an implementation is checked to see whether or not it is consistent with a given specification in the form of a finite state machine. They also discuss adaptive testing where "the next input symbol depends on the previously observed ouputs". This type of test shares similarities with the problem of crawling web applications since the next input (event executed) is dependent on the previously observed ouputs (the set of enabled events on the page).

Lee and Yannakakis also describe five types of testing problems. In the first there is a need to identify the final state of the machine (Homing). In the second, state identification, the problem involves identifying an unknown state. Third, there is state verification where there is a need to

verify that the machine is in a given state. The forth problem is machine verification in which there is a need to check whether two machines are equivalent. Finally, there is the machine identification. In this problem, the actual implementation (black box) is tested in order to build a transition graph which models it. This aligns with what this research aims to do. Given a web application, crawling has to be performed so that a model of the application can be created. More [43] provides a solution to the problem that is exponential in terms of the number of states in the machine. Lee and Sabnani [44] show a practical use of machine identification (reverse engineering communication protocols). The authors of [42] make the assumption that the machine to be identified is strongly connected (every state is reachable from a given state). Otherwise, some states may be unreachable depending on the state in which the experiment is started. In the case of a web application, if all states are to be reached, then every state of the application should be reachable from the initial state of the application. This initial state is considered to be the state visited when the page corresponding to a given URL of the application is loaded.

As mentioned above, existing papers, including [18] and [25], use finite state machines to model AJAX applications. Based on the various ways in which 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 [18] and [25]), we consider FSMs as an appropriate technique for modeling AJAX applications and we will use this technique in this work.

3 Challenges

There are various challenges when attempting to crawl AJAX-based web applications. Some of them are relevant to the crawling of web applications in general while others are more specific to applications that update the page through asynchronous requests. With the aim to produce a strategy for crawling modern web applications, including AJAX-based applications, we have identified some issues which will need to be addressed.

3.1 Fine-Grained Control over JavaScript Events

In AJAX-based applications, when an asynchronous request takes place, some amount of time passes before the response is received and the page updated accordingly. However, between the time that the code is executed resulting in the sending of an asynchronous request, and the time when the callback method is called when the response is received from the server, the application may exist in some intermediate state which is neither equivalent to the pre-request state nor the post-request state.

As an example, a scenario where a user is logged into a social networking application is considered. In the initial state, shown in Figure 1 the user sees a list of contacts as well as a welcome message. Every user enters his or her full name before using the application but each

entry in the contact list simply displays the user's first name (all characters entered before the first space).

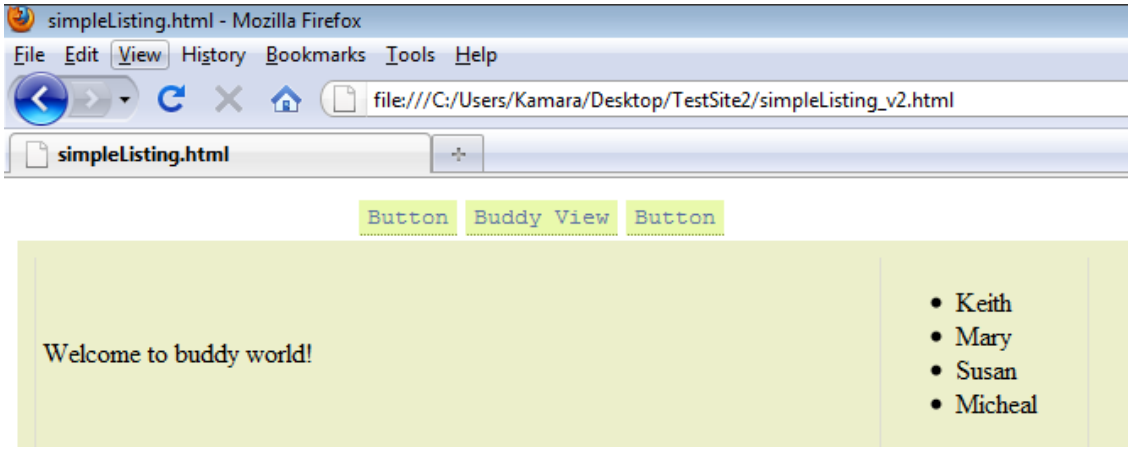

Figure 1: The initial state of the application (before the "Buddy View" button is clicked)

The user also has access to a few buttons, one of which is named *Buddy View*. Clicking on this button causes an asynchronous request to be made to the server. The response from the server will include a document containing the personal details of each of the user's buddies. Once this response has been received, the user can *mouseover* any of his or her friends in the contact list causing that person's full details to be displayed, as shown in Figure 2. This represents a new state in the application.

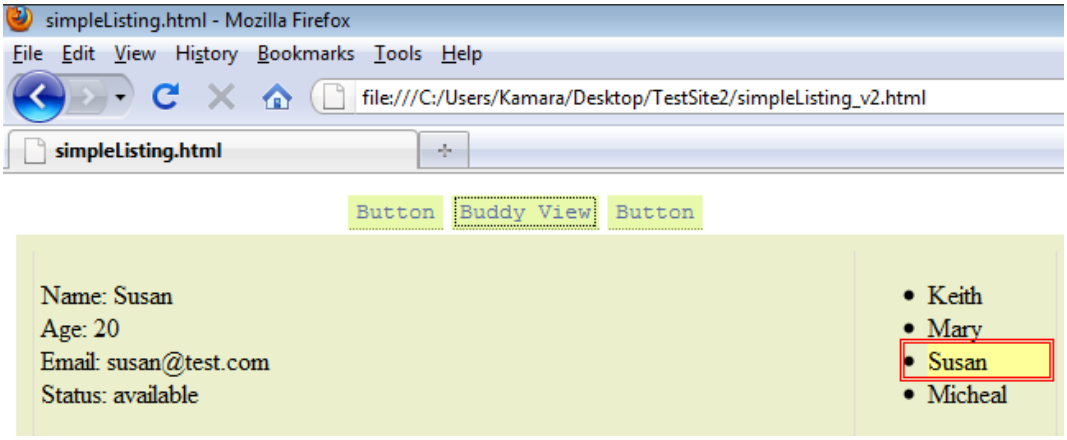

Figure 2: Susan's details are displayed

In the event that there is some delay in receiving a response from the server, the developer of this application implemented a feature to allow some information to be displayed even if a *mouseover* is done before the response is received from the server and has been processed. In this case, the developer decided to simply display the complete full name which the user had entered. This intermediate state of the application is shown in Figure 3.

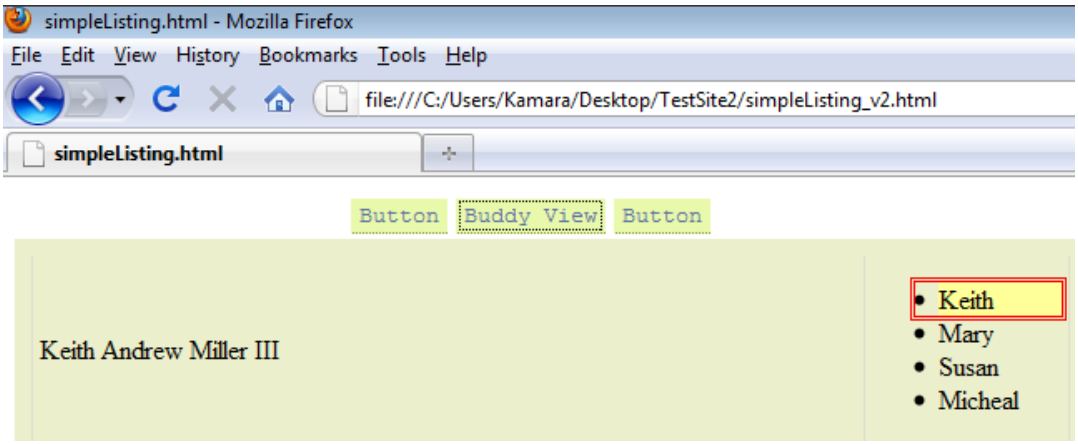

Figure 3 : The application shows the user's full name in this intermediate state

Now let us say that this application also includes a function for sanitizing data. It is used to sanitize a user's full details (after the reception of the response to the asynchronous request) before they are displayed. It is also used to sanitize the first portion of the full name (all characters before the first space) before it is displayed in the buddy list. However, the developer has forgotten to use this function to sanitize the user's full name before it is displayed while in the previously mentioned intermediate state that we have described. As a result, in the event that a user performed some sort of injection when entering his or her full name, this could potentially go unnoticed and end up being displayed in the intermediate state, as shown in Figure 4. In this case, Mary has injected a login form when entering her full name. There are actually two intermediate states in this example. The application enters the first of these when *Buddy View* is clicked and the *onclick* event has been executed. From this state, the application enters the second intermediate state (the one which we have described, where the application's vulnerability can affect a user) if the user does a *mouseover* on a contact before the callback has been executed.

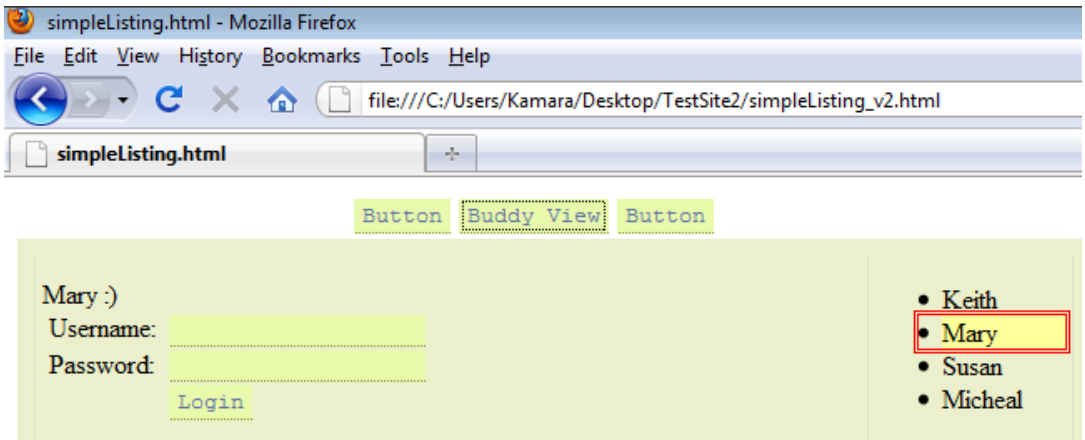

Figure 4: The result of the injection attack is seen in this intermediate state

This example is summarized in Figure 5. In the state *BuddyClicked*, the *onContactMouseOver* event is executed resulting in state *MousedOver* which exists until the callback method is executed causing a transition to the state *ContactDetailsDisplayed*. As the example illustrates, it is important that these intermediate states are reached and processed, particularly in tasks such as crawling for security scanning. As a result, in AJAX applications, it is necessary to be able to have complete control over the processing of JavaScript events at the client-side. In general (for all rich internet application technologies), this means that it is necessary to have the knowledge of which events are available and their types. There is also a need to have the ability to control the execution of any sequence of these events. Also, whenever an HTTP request is made to the server, there is a need to be able to control when the resulting callback is executed. This will allow capturing of these intermediate states.

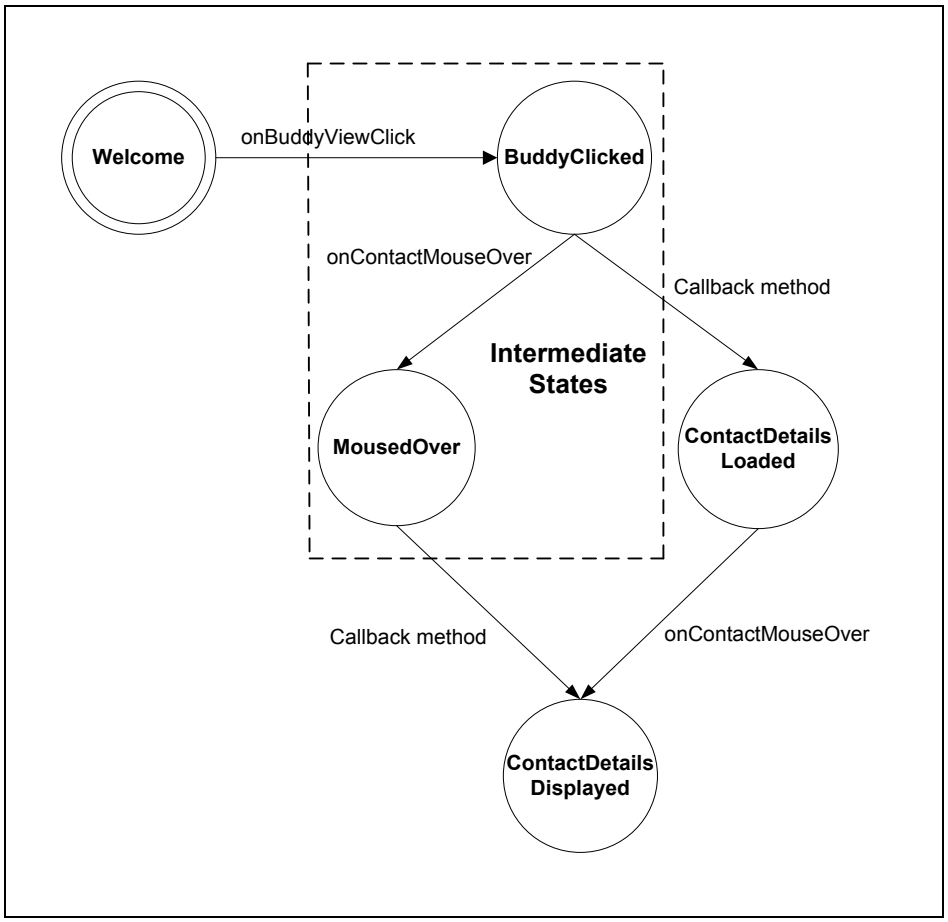

Figure 5: A model of the example website highlighting the intermediate state

3.2 Control over Application Flow

In order to complete the crawl of a given application, it is also necessary to revisit states in order to follow a different path compared to the ones that were previously followed. It may be possible to complete this task of undoing the affects of the previous actions by resetting the application to its initial state, then traversing the application again until the desired state is reached. However, this approach may prove to be extremely inefficient if the steps that are required to reset the

application and return to the desired state are too numerous. It is therefore a challenge to find ways to reduce the number of such steps.

3.3 State Equivalence

Mathematically, an equivalence relation is one which divides the elements of a set into subsets where each element is in exactly one subset. These subsets form a partition of the state space. In crawling, there is a need to determine state equivalence in order to divide the states of the application into subsets. This is important because it reduces the potential size of the model of the application, since one element from each subset is included in the model. This element represents all other members of that subset.

The ideas brought forward for differentiating between states which we have seen in [18], [19] and [22] all focus on being efficient. While it is also important to work towards a state equivalence function which is efficient, the initial focus needs to be on ensuring that the solution is based on a valid equivalence relation and one that is meaningful for the application being crawled. It is necessary to think about how the equivalence of state is affected by factors such as the structure of the page, the enabled events on the page, and the text on the page. We also need to consider how the use of the model will be affected by state equivalence. Existing research has not accounted for many of these considerations.

3.4 Infinite Runs

When crawling applications, it would be logical to follow a path to its end in order to minimize the number of times that a state is revisited (or the application reset). As a result, there is always the danger of ending up in an infinite loop [18]. That is, we enter some portion of an application which results in a loop because executing a certain event or certain events continually leads to new states. In addition to possibly never finishing the crawl of an application, this problem also means that if given a specific period of time to crawl, the model that is produced at the end of that period will only have covered a very small part of the application, perhaps a particular branch which will continue to be traversed while following the loop.

There are some factors which could help prevent such situations. First, an equivalence function can help identify the arrival at a state which belongs to the same subset as a previously visited state. In addition, the crawling strategy should help to mitigate the effect of an infinite loop even when the equivalence function is unable to identify such a case. In order for this to be possible, there is a need for a strategy that provides a compromise between the desire to minimize the number of times that a state needs to be revisited and the aim to maximize the breadth of the part of the model that is built, particularly when time is limited.

3.5 Slow Executions

There are various factors that may lead to slow execution times for a crawler. Lack of efficient control over flow is one of these, since it may necessitate having to repeat a series of actions multiple times in order to return to a desired state. This means more potential page loads and event executions, increasing the duration of the crawl. The need to keep track of intermediate states could also increase the overall number of states that need to be covered, therefore increasing the length of the execution time.

3.6 State Space Explosion

Identified in [18], state space explosion is also a challenge. Many web applications consist of thousands of states which will need to be identified, compared, stored, and modeled. There is a need to crawl web applications with a focus on finding as many non-equivalent states as possible in a given amount of time. This is especially important because it may not be feasible to crawl the entire application. We also need to ensure that the equivalence function does not contribute to this challenge by being too strict, and thus evaluating two states as different when they should be equivalent.

3.7 Data Input Values

In crawling, data input values are extremely important because they may determine what states are reached. Given that in many cases, the number of possible input values is practically infinite, it is a challenge to determine how to choose a realistic set of input values in order to be able to reach all possible states. It is also a challenge to automatically infer the format and types of values which will let the application function correctly.

3.8 Server States

While the client-side state is largely determined by the current DOM, server-side state may be determined by the values of variables which are stored in the server or by entries in a database. The issue of server-side states (also discussed in [18]) is very important for building an accurate model of an application. When there is a change in the server state, it could result in the same set of actions being executed at the client side but resulting in different client states. Of course, there may be some set of actions that can be taken to "reset" the application, in order to return to the initial server-side state. However, if these steps require too much work, it would have an extremely negative impact on the efficiency of crawling. One idea to evaluate is the possibility of making a distinction between events that do 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 and reset entirely at the client side.

3.9 Crawling Strategy

In trying to crawl applications including asynchronous events, it is an especially important requirement that an efficient strategy is developed which will result in the ability to more quickly crawl the entire website, visiting all states (as determined by the equivalence function), and executing all events (in these states). Since, as mentioned in Section 3.6, there may be thousands of states within an application, complete crawling may require a very long time and may not always be feasible. Therefore, a strategy which will capture as many states and transitions as possible in a given time is needed.

Consequently, it is important that the strategy aims at discovering as many states as possible using a minimum number of event executions and page reloads. If there is enough time to discover all states of an application, the crawl can then continue by confirming that various combinations of actions indeed lead to the same states. There is a need to determine which sequences of event executions and page loads would best serve these goals. In previous work in [18] and [25], the topic of a crawling strategy has not really been addressed in detail. Instead, simplified crawling strategies based on breadth-first search and depth-first search have been used.

3.10 Incomplete Model

It is necessary to ensure that the model which results from using the crawling strategy that is developed includes all states and events which exist in the application. If the model ends up missing one or more of these states and events, we consider it to be incomplete.

When several events can be executed from a given state, it is possible that executing them in different orders will lead to different results (i.e., different states). Trying all the combinations is obviously very time-consuming, but running a single one of the possible sequences is not an acceptable trade-off.

Another potential pitfall that may lead to an incomplete model is a failure to capture intermediate states. It is necessary to account for the state of the page before an event is executed, before the associated callback is executed, and after the callback has been executed. In addition, an event which causes multiple asynchronous events may result in multiple callback methods being executed and so there may be multiple intermediate states in such a case.

The ability to correctly determine state equivalence will also factor in the ability to avoid constructing an incomplete model. While it is important to avoid unnecessary state explosion as much as possible, an equivalence function which ensures that states which are distinct will not be considered equivalent is needed.

4 State Equivalence

The ability to distinguish one state from another is critical to being able to successfully crawl a web application. This is particularly true in the case of AJAX applications where one cannot rely on information such as the current URL to help identify states since multiple states may share the same URL. In Section 3.3 state equivalence is defined. Following from that definition, there is a need to determine whether or not two states are members of the same subset. This method of evaluating whether or not two states are equivalent (the equivalence function, referred to in this thesis as ≈) impacts the ability to ensure that one can crawl an application entirely as well as do so efficiently.

The equivalence function determines the number of subsets (of states) which are created (and therefore the number of states in the model). If the function used for state equivalence creates too few subsets, this results in states being classified as equivalent even when they should not be. This could produce a model which is missing states (Section 3.10). In a scenario in which crawling is used to uncover states for the purpose of security analysis, missing states would mean that some states will not be analyzed. This could result in undetected vulnerabilities.

If the state equivalence function produces too many subsets, it could mean that states that should be considered equivalent end up being interpreted as not being equivalent. This could cause the

model to have more states than necessary, affecting efficiency and leading to unnecessary state explosion (Section 3.6), causing longer runs, or infinite runs (Section 3.4).

The ideas presented in this chapter help to address these challenges by improving on existing methods of identifying duplicate states. However, additional work will need to be done to ensure that these problems are fully solved.

4.1 Different Types of Equivalence

As mentioned in Section 1.4, the purposes for crawling web application are varied. With each purpose, there are also aspects or elements of the page that are more important than others. For instance, when crawling an application for the purpose of indexing (for example, for use by a search engine), the text found in each state is essential and must be captured. Therefore, two states that are identical in structure but with different text should not be judged equivalent since users of a search engine may want to search for one or the other. On the other hand, when looking for security vulnerabilities in an application, the elements of the page which allow the user to interact with the page and related input data are more relevant. Therefore if the previously mentioned states contain different news articles, but the exact same elements and logic for allowing users to enter comments about these articles then these states should be judged equivalent. This is because a security test would not be concerned with a difference in text on the page and only seeks to evaluate the security vulnerabilities that exist on the page.

While the purpose of the crawl is important, the crawler's main job is to find as many states as possible. In other words, two states can only have the possibility of being considered equivalent if the set of states that can be reached from them are equivalents. This is where events, hyperlinks, user input controls such as forms or dropdowns, or anything else that could influence the set of states that are reachable from the current one become extremely important. Therefore, two states with identical text may be equivalent if the purpose of the crawl is to index content. However, if these two pages have a different set of enabled events, then these two states cannot be equivalent in any crawl, since they may have different sets of states which are reachable from them.

In order to discover all the unique states (in terms of the purpose of the crawl) of an application, both crawling equivalence (based on the set of states which is reachable from a given state) and equivalence based on the purpose of the crawl must be taken into consideration. It also means that depending on the purpose of the crawl the model of a given website could vary. Failing to take either type of equivalence into account could result in states being missed, an incomplete model, and the inability to fulfill the purpose of the crawl. Therefore, any function which determines whether or not two states, s_1 and s_2, are equivalent ($s_1 \approx s_2$) should evaluate to true only if the following condition holds:

$$eq_{crawling}(s_1, s_2)\ AND\ eq_{purpose}(s_1, s_2)$$

In this condition, $eq_{crawling}$ is an equivalence function based on crawling equivalence and $eq_{purpose}$ is an equivalence function 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.

Logically, if two states are identical then they are also members of the same subset. Therefore states s_1 and s_2 are also judged equivalent if the following condition holds:

$$areIdentical(s_1, s_2)$$

Therefore, the previous condition only needs to be evaluated if the two states are not identical. Otherwise, it is automatically known that they are identical.

4.2 "Load, Reload"

Web pages often contain bits of content that change very often but are not important in terms of making two states non-equivalent. These could include, but are not limited to, advertisements, counters, and time stamps. Figure 6 shows a page which highlights this type of content.

```
<html>
    <body>
        <div class = "advertisement">
            <img src=http://testsite.com/img/ad1.gif height="5" width="3">  advertisement image
        </div>
        <p>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed
        viverra interdum risus, quis congue mi pretium a. Ut quis nulla
        erat.
        </p>
        <div>
            Current Time: 11:59:59pm  time stamp
        </div>
    </body>
</html>
```

Figure 6: Example of a page with irrelevant data which changes over time

When determining whether or not two states are equivalent, there is a desire to be able to ignore these constantly changing but irrelevant portions of the page. This is especially important in AJAX-based applications since failing to identify data that should be ignored could cause an equivalence function to evaluate to false when it otherwise would not.

We have developed a technique for automatically inferring the portions of the page that should be ignored. It requires loading a given page twice. The DOM of the page at each load can then be compared and the differences indicate data that can be ignored. For example, a page *x* is loaded at a time t_1 and then again at t_2. The DOM of x at t_1 is then compared to the DOM of x at t_2 to produce Delta(X), in the form of a list of differences between the DOMs. When using an equivalence function to compare this state with another, the data in this list can be excluded. Therefore, two states can be considered identical if they are identical after the irrelevant data is excluded from both.

5 Crawling Strategy

5.1 Overview

When developing the strategy for crawling rich internet applications, it is a goal to be able to find any given state in a finite amount of time. This would make all content available for analysis or indexing. In addition to being able to uncover the complete model of an application, the process must take place in a deterministic fashion. Therefore, if the crawler is given *x* minutes to crawl an application and all other factors are also equal (for example, the server response time for each request) crawling should be completed in such a way that the model constructed (partial model if *x* minutes is not sufficient for completing the crawl) is the same on subsequent crawls of *x* minutes as long as the application remains unchanged. In a product which completes tasks such as security scanning, this is important because it means that roughly the same set states would be uncovered and available for analysis each time, providing a more predictable experience for the user.

It is also very important to recognize that given a large web application, it may not be feasible to crawl the entire application. Therefore, it is a priority to find as many states as possible within a given time. Additionally, even in circumstances where there is enough time for the crawler to uncover every state of the application, there may still not be enough time to execute every

transition. With this in mind, there is an additional priority. Once all states have been discovered, there is a desire to cover "new" transitions (ones which were not previously traversed) quickly.

To aid the development of the strategy, it is necessary to make some assumptions about the application which is being crawled. These assumptions ease or facilitate the ability to produce a strategy for modeling applications that use asynchronous requests to the server in order to retrieve data and update portions of the page using client-side JavaScript. The following are the assumptions about the application being modeled:

- It is possible to return to a previously visited state by "resetting" the application and repeating some set of actions. That is, if we start from a given URL and execute a series of actions, it is possible to "reset" the application such that beginning from the same URL and executing the same series of actions again, will produce the same results. It is not, however, assumed that we can simply "step backwards" from the current state to the previous one.
- The only source of non-determinism is concurrency. What we want to avoid is an application that will react differently, starting from the same global state, when the same input is given at two different times.
- Every interaction between the application and the user can be modeled as a choice among a known finite set of possibilities. This fits well with input such as buttons, check boxes, and down-down menus. This means that for now, applications that allow the user to enter "free text" are not considered. This would allow for an infinite set of possibilities.

It is also important to point out whether or not the crawling strategy addresses the challenges described in Chapter 3. In this regard, the strategy does not consider intermediate states (Section 3.1). Instead, events which lead to an AJAX call are treated as synchronous events. This means that the state following such an event is considered to be the one which exists after a response has been received from the server and the callback method executed. The issue of control over application flow (Section 3.2) is currently handled by using the URL to reload the page in order to reset the application. However, this will not be sufficient for all applications so this challenge will need to be further addressed in the future. The strategy also addresses the danger of infinite runs (Section 3.4) by limiting the traversal depth (described later in this section). In addition, given the means by which control over application flow is currently achieved as well as the current exclusion of intermediate states, slow executions (Section 3.5) are not an issue at this time. At present, the strategy also does not address the challenges related to data input values (Section 3.8) or server states (Section 3.9). Finally, the strategy dictates that all events in each state are executed. This would help to avoid an incomplete model (Section 3.10). However, by the given definition of an incomplete model, the lack of intermediate states means that this issue is also not fully addressed by the current strategy.

It should also be mentioned that the crawling strategy used is independent of the purpose of the crawl. Therefore this strategy would be suitable for a variety of purposes provided that the equivalence function used is based on the purpose of the crawl.

Overall Crawling Strategy

In AJAX applications, the current state of the application may change in two ways. The first is through synchronous HTTP requests to the server, for example when the user clicks on a URL which is part of the DOM of the current state. The other way to change state is through the use of asynchronous HTTP requests and local JavaScript execution. This may either be initiated by user-input or some time-out mechanism.

With this in mind, the overall crawling strategy needs to take both of these ways of state changes into account. Therefore, both traditional crawling and event-based crawling are taken into account. In traditional crawling, new URLs are followed (through, for example hyperlinks, to discover new pages). In event-based crawling, events are executed on the page (possibly causing asynchronous requests) to move from state to state in order to discover new states. There is also a parameter *k* for alternating between the two approaches. To do so, we follow *k* URLs in the traditional crawl then traverse *k* chains (we discuss *chains* later in this chapter) in the event-based crawl. This process of alternation between the different methods of crawling is continued until the crawl is completed or the crawler is stopped. In addition, a list of links (L) and a list of states (B) are kept. L represents URLs which have not been visited by the crawler. B represents a set of states which have some enabled events and which have not been completely explored by the crawler. L is updated by removing a URL when it has been visited and adding any new URL that is discovered during the crawl. Whenever the crawler arrives at a new state (with enabled

events) via synchronous communication, this state is added to B. These states are called base states. When the event-driven crawling of a state in B is completed, the state is removed from B.

The algorithm *crawlRIA(l,k)* (Figure 7) is used for crawling applications. The input l is the start URL of the application to be crawled. It becomes the first URL added to the list of links (L). The crawl is fully completed when both L and B are empty.

```
Procedure crawlRIA(l, k)
Input  l: the URL of the initial state of the application (String)
Input k: limit of exploration using either method (Integer)
begin
L = {l} ;
B = ∅ ;
while (L ≠ ∅ OR B ≠ ∅ ) {
        for (i = 1 to k ) {
                if (L ≠ ∅){
                        traditionalCrawl(L,B,);
                }
                else{   break;  }
        }
        for (i = 1 to k ) {
                if (B ≠ ∅){
                        eventBasedCrawl (L,B);
                }
                else{   break;  }
        }
}
end
```

Figure 7: Procedure for crawling

Below, the algorithm used to complete traditional crawling is discussed quickly. Following this, there is a detailed account of the strategy for event-based crawling.

Traditional Crawling

Traditional crawling is accomplished using the procedure *traditionalCrawl(L,B,k)*, shown in Figure 8. It begins by removing the next URL from the list L. A synchronous HTTP request is then made using the URL and after receiving a response, the resulting page is loaded. If this results in the arrival at a previously unvisited state, this new state is processed. Processing the state entails two steps. First, any new URLs within the current state are added to L. Second, if the state has any enabled events, it (the state) is added to the list B, which means that it will be explored at some point during the event-based crawl.

```
Procedure traditionalCrawl(L, B)
Input  L : set of URLs that are to visit
Input  B : Set of discovered states with enabled events (base states)
begin
pick and remove a URL l from L;
Let s be the state retrieved by requesting l from the server;
if (¬∃s´ in B such that s ≈ s´){
        foreach(URL l´ in s ){
                L = L ∪ {l´};
        }
        if (s has some set of enabled events) {
                B = B ∪ {s}
        }
}
end;
```

Figure 8: Procedure for traditional crawling

Event-Based Crawling

The procedure of event based crawling will be very important in determining how efficient the overall crawling strategy performs. Applying a simple breadth-first or depth-first type strategy is one way to complete the crawl. This is more or less the approach taken in [18] and [25]. However we must remember the assumption that in order to "go back" to a previous state, we need to at least load the URL of that base state again and retrace the steps to that state. Therefore, in addition to a desire to limit the amount of required transitions (events that are executed), it is also important to limit the number of resets that are required. To accomplish this goal, a more complex strategy for event-based crawling needs to be developed. For this purpose, a hypothesis is made about the application which, if true, allows the generation of an optimal strategy. The efficiency of the strategy would therefore be affected by the accuracy of this hypothesis. However, the strategy does not rely on it in order to be able to complete event-based crawling. Since the hypothesis may be invalidated, there is also a technique for adapting the strategy so that it is consistent with what has already been discovered about the application.

The hypothesis is as follows: Given a state *s* that has *n* enabled events, e_1, e_2,…, e_n, it is assumed that these *n* events are independent. When event *e* in state *s* is executed a state is reached where all events that were enabled in *s* except *e* are still enabled. This means that if one starts at *s* and executes a given subset of these events in any order, this will lead to the same state. According to this, there are 2^n possible subsets of events, which, when ordered by inclusion, define a hypercube of size *n*, consisting of *n!* different paths from the bottom to the top. The bottom of

the hypercube is defined as the initial state *s*. The top of the hypercube refers to the state in which there are no enabled events. This state can be reached by starting at the bottom of the hypercube and executing all *n* events in any order. Figure 9 is an example showing a hypercube of dimension four. There are 4!=24 different paths in this hypercube, with 2^4=16 different states. An efficient strategy for crawling this hypercube is developed. Following the goals that were previously outlined, it is important to discover all states of the hypercube first, and then ensure that all transitions are executed. In the following sections, we explain how these two objectives are reached and then give a summary of the complete procedure for event-based crawling.

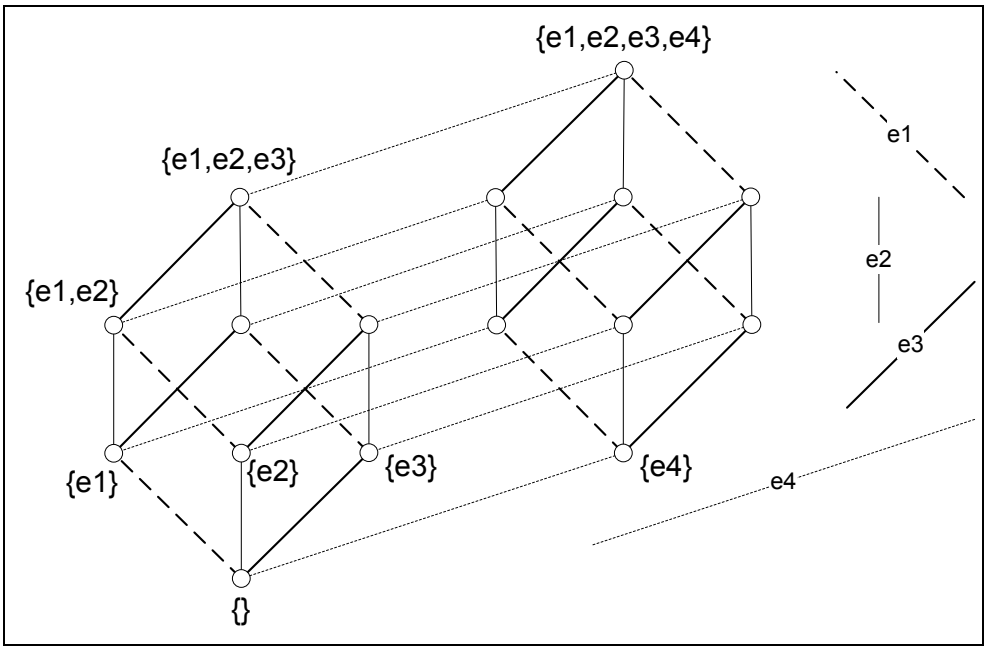

Figure 9: A hypercube of size 4 dimensions

5.2 Minimum Chain Decomposition

A hypercube is a partially ordered set (a lattice in this case), and each path of the hypercube is actually a chain of the order, that is, a set of pairwise comparable elements. The goal of visiting each state of the hypercube using a minimum number of resets is achievable using what is known as a minimal chain decomposition of the order ([45] presents an overview of these concepts). It has been proven in [47] that the minimal number of chains necessary to decompose an order is equal to the width of this order, that is, the maximum number of pairwise non-comparable elements. Therefore, since the width of a hypercube of *n* dimensions is equal to $\binom{n}{\lfloor n/2 \rfloor}$, this value also represents the number of paths (chains) necessary to visit every state of the hypercube. As an example, given a hypercube of size 4, the number of chains required to visit all states is equal to $\binom{4}{\lfloor 2 \rfloor} = 6$. Given that there are 24 (4!) paths in this hypercube, only 6 of those 24 paths are required to discover all the states.

In 1952, de Bruijn, Tengbergen, and Kruyswijk [47] provided an algorithm for decomposing certain orders, including a hypercube. In [48], Hsu, Logan, Shahriari, and Towse expose the methods as follows (adapted to the hypercube definition):

Definition (adapted from [48]): The *canonical symmetric chain decomposition*, or CSCD, of a hypercube of dimension *n* is given by the following recursive definition:

1. The CSCD of a hypercube of size 0 contains the single chain (Ø).

2. For $n \geq 1$, the CSCD of a hypercube of dimension n contains precisely the following chains:

    1) For every chain $A_0 < \ldots < A_k$ in the CSCD of a hypercube of dimension $n - 1$ with $k > 0$, the CSCD of a hypercube of dimension n contains the chains:

    $$A_0 < A_1 < \ldots < A_k < A_k \cup \{n\}$$

    and

    $$A_0 \cup \{n\} < A_1 \cup \{n\} < \ldots < A_{k-2} \cup \{n\} < A_{k-1} \cup \{n\}$$

    2) For every chain A_0 of size 1 in the CSCD of a hypercube of dimension $n - 1$, the CSCD of a hypercube of dimension n contains the chain:

    $$A_0 < A_0 \cup \{n\}$$

Applying this method to the hypercube of dimension 4 leads to the following 6 minimal chains decomposition:

1. $\{\}<\{e_1\}<\{e_1, e_2\}<\{e_1, e_2, e_3\}<\{e_1, e_2, e_3, e_4\}$
2. $\{e_4\}<\{e_1, e_4\}<\{e_1, e_2, e_4\}$
3. $\{e_3\}<\{e_1, e_3\}<\{e_1, e_3, e_4\}$
4. $\{e_3, e_4\}$
5. $\{e_2\}<\{e_2, e_3\}<\{e_2, e_3, e_4\}$
6. $\{e_2, e_4\}$

This is illustrated in Figure 10 (states are identified by the events which were executed in order to arrive there) with chains emphasized in bold. Note that two of the chains consist of just one state.

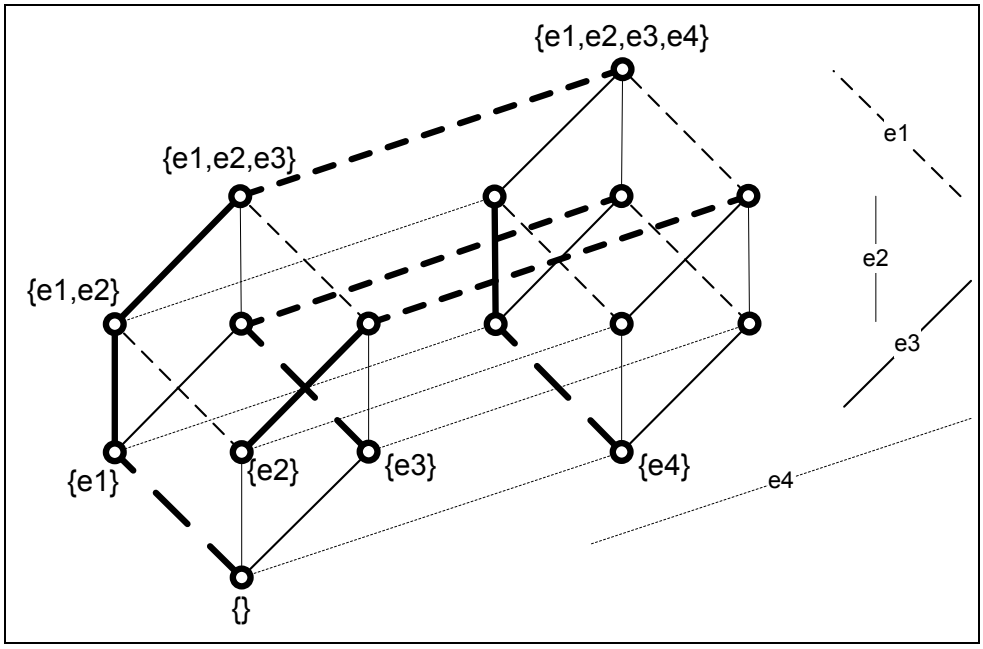

Figure 10: Minimum Chain Decomposition of a hypercube of size 4

5.3 Minimum Transition Coverage

Given that to the goal is not only visiting every state as quickly as possible, but also crawling the entire application (execute every transition) as quickly as possible, there is a need for more than just the MCD algorithm. In order to accomplish this we have developed a Minimum Transition Coverage (MTC) algorithm. This algorithm focuses on executing every possible event in as few paths as possible (requiring the minimum number of resets). However, in keeping with the goal

of first visiting every state as quickly as possible, the MTC algorithm accepts as input, a set of disjoint chains called constraints. Each of these chains becomes a sub-chain of one of the chains produced using the MTC algorithm (as discussed later in this section). Furthermore, the final set of MTC chains are ordered such that constraint-containing chains come before non-constraint containing chains. Therefore, if the chains produced by the MCD algorithm are used as constraints for the MTC algorithm, the first goal can be achieved as well.

The MTC algorithm is shown in Figure 11 . It consists of four steps. First, the middle level of the hypercube is found. Then, a set of upper chains (chains which begin at the middle level of the hypercube and go upward) is generated followed by a set of lower chains. For each constraint chain, the portion which exists above the middle level (or the full chain if it exists entirely above the middle level) becomes a sub-chain in one upper chain. The same is true for the portion which exists below the middle level (or the full chain if it exists entirely below the middle level). It becomes a sub-chain in one lower chain. Following this the algorithm enters a phase where chains covering the upper portion of the hypercube are combined with chains covering the lower portion of the hypercube. The combined chains are then extended downward to the bottom of the hypercube.

```
Algorithm MinimalTransitionCoverage
Input H: a hypercube of dimension n
Input C_C: C constraint set of chains (list of chains)
Output C_M:  MTC of H constrained by C (list of chains)
begin
        C_U = Ø;//chains from the middle level to the top of the hypercube
        C_D = Ø;//chains from the middle level to the bottom of the hypercube
        C_M = Ø;

        C_U = GenerateUpChains(H, C_C);
        C_D = GenerateDownChains(H, C_C);

        C_M = CombineChains(C_U, C_D);
        C_M = ExtendChainsDown(C_M);

        return C_M;
end
```

Figure 11: Minimum Transition Coverage (MTC) algorithm

The tasks are accomplished as follows:

Upper Chains Stage : In this stage (performed by the procedure *generateUpChains* which is shown in Figure 12), a set of chains (C_U) covering all the transitions above the middle level of the hypercube is generated. In doing this we must also take into consideration the existing chains that are present in the set of constraints (C_c). This stage begins by starting at a state in the middle level of the hypercube and building a chain upwards. A chain is built upward by first selecting a transition (t=(s-e-s')) which has not been previously used in the MTC chains. We then need to check to see whether or not this tranisition is used in one of the constraint chains.

If it is in fact used in a constraint chain and either we are still at the middle level state or the transition represents the first in a constraint chain, then the upper chain can be extended with the entire portion of the constraint chain which follows from the current state. This is called the suffix of the chain. If the transition is used in a constraint chain but we are not at a middle level state and this is not the first transition of the constraint chain, then this transition cannot be used and must be marked as unavailable. This is because we do not want to split the upper level portion of any constraint chain into multiple parts.

If the transition is not used in any constraint chain then it can be used to extend the current upper level chain. This process of extending the chain is continued until we reach a state in which we cannot find any unused and available transition. We then go back to the original middle state and repeat the process for each unused transition in that state. These actions are repeated at each middle level state until we have covered all upper level transitions while incorporating the constraints.

```
Procedure generateUpChains
Input H a hypercube of dimension n
Input C_C: a constraint set for MTC (list of chains)
Output C_U: chains from the middle level to the top of the hypercube (list of chains)
begin
        C_U = Ø;
        U_C = Ø; //current upChain
        foreach (state s in the middle level of H) {
                /*use each event available in s as the first transition in one chain (build one chain for
                each of these transitions*/
                foreach(event e in s where transition t=(s-e-s') is unvisited)
                        s_M = s; //current middle level state
                        U_C = s;
                        //add a transition to extend the chain
                        do{
                                //check to see whether this transition exists in a constraint chain
                                if(∃C ∈C_C and C contains t){
                                        if(first(C) = s or s is at middle level){
                                                U_C = (U_C – s) U suffix C_C (s);
                                                Mark each transition in U_c as visited;
                                                S = last(C);
                                        }
                                        else{
                                                Mark t as unavailable;
                                        }
                                }
                                else{
                                        U_C = U_C U e-s';
                                        S = s';
                                        Mark t as visited;
                                }
                        /*at every iteration, t is the candidate transitions for extending the chain
                        }while(∃e in s where transition t=(s-e-s') is unvisited and t is not unavailable)
                        //add chain to set of upward chains
                        if(length(U_C) > 1){
                                C_U = C_U U U_C;
                        }
                        S = s_M;
                }
        }
        return C_U;
end
```

Figure 12: Procedure generateUpChains

Lower Chains Stage : In this stage, a set of chains (C_D) covering all the transitions below the middle level of the hypercube is calculated. This is symmetric to the upper chains stage and thus is completed in the same manner.

Chain Combination Stage : The procedures associated with this stage are shown in Figure 13 and Figure 14. In chain combination (performed by the procedure *combineChains*), the chains in C_U and C_D are joined into larger chains spanning both the lower and upper portions of the hypercube. First upper chains which contain a portion of a constraint chain (chains in C_{UC}) are matched with lower chains which contain a portion of the same contraint chain (chains in C_{DC}), thus keeping the constraint chains intact. When using MCD chains as the constraints, there is always a one-to-one match between constraint-containing upper chains and constraint-containing lower chains starting from each state, so this part is simple. Using a different set of constraints, there may not be a one-to-one match so there may be a greater number of chains in C_{UC} compared to C_{DC}. In this case, lower chains are matched with upper chains until there are no unmatched lower chains remaining. At this point we iterate over the lower chains, matching them with upper chains to create complete chains until there are no unmatched upper chains either. This is performed by the procedure *matchChains*. The next step is to combine non-constraint containing upper chains (chains in C_{UN}) with non-constraint constaining lower chains (chains in C_{DN}). This is also completed using the procedure *matchChains*.

Procedure combineChains
Input C_U: List of chains from the middle level to the top of the hypercube
Input C_D: List of chains from the middle level to the bottom of the hypercube
Output C_M: MTC of H constrained by C (*list of chains*)
begin
    C_{UC} = chains in C_U with constraints;
    C_{DC} = chains in C_D with constraints;
    C_{UN} = chains in C_U without constraints;
    C_{DN} = chains in C_D without constraints;
    $C_M = \emptyset$;

    //Combine constrained upper chains with constrained lower chains
    C_M = MatchChains (C_M, C_{UC}, C_{DC}); //fix

    //Combine non-constrained upper chains with non-constrained lower chains
    C_M = MatchChains(C_M, C_{UN}, C_{DN});
    return C_M;
end

Figure 13: Procedure combineChains

Procedure matchChains
Input C_M: MTC of H constrained by C (*list of chains added to this point*)
Input C_{UN}: non-constrained chains from the middle level to the top of the hypercube
Input C_{DN}: non-constrained chains from the middle level to the bottom of the hypercube
Output C_M: MTC of H constrained by C
begin
    complete = false;
    while(!complete){
        if($\exists C_U \in C_{UN}$ and $\exists C_D \in C_{DN}$ and C_U and C_D are unmatched and start(C_D) ≈ start(C_U)){
            combine C_D and C_U and add to C_M;
        }
        else if($\exists C_U \in C_{UN}$ and $\exists C_D \in C_{DN}$ and C_U is unmatched and start(C_D) ≈ start(C_U)){
            combine C_D and C_U and add to C_M;
        }
        else{
            complete = true;
        }
    }
end

Figure 14: Procedure matchChains

Chain Extensions Stage: The chain extension stage ensures that each MTC chain begins at the base of the hypercube. To do this, we take each chain that does not adhere to this rule and continually add down transitions until we arrive at the base of the hypercube. This needs to be done since we always need to start at the base state before traversing to any specific state.

Given a hypercube of size 4, the MTC algorithm produces 12 chains. This is half of the 24 paths present in a hypercube of that size. The 6 MCD chains are also covered within the first 6 chains produced. The chains produced are as follows:

1. **{}<{e_1}<{e_1, e_2}<{e_1, e_2, e_3}<{ e_1, e_2, e_3, e_4}**
2. **{}<{e_3}<{e_1, e_3}<{e_1, e_3, e_4}<**{e_1, e_2, e_3, e_4}
3. {}<**{e_4}<{e_1, e_4}<{ e_1, e_2, e_4}**
4. {}<**{e_2}<{e_2, e_3}<{e_2, e_3, e_4}**<{e_1, e_2, e_3, e_4}
5. {}<{e_4}<**{e_2, e_4}**<{e_1, e_2, e_4}
6. {}<{e_4}<**{e_3, e_4}**<{e_1, e_3, e_4}
7. {}<{e_2}<{e_1, e_2}<{e_1, e_2, e_4}<{e_1, e_2, e_3, e_4}
8. {}<{e_1}<{e_1, e_3}<{e_1, e_2, e_3}
9. {}<{e_1}<{e_1, e_4}<{e_1, e_3, e_4}
10. {}<{e_3}<{e_2, e_3}<{e_1, e_2, e_3}
11. {}<{e_2}<{e_2, e_4}<{e_2, e_3, e_4}
12. {}<{e_3}<{e_3, e_4}<{e_2, e_3, e_4}

In general, the number of paths required to complete crawling of a hypercube of n dimensions using an MTC-based strategy is equal to $\binom{n}{\lfloor n/2 \rfloor} * \lceil n/2 \rceil$. This is equal to the states at the middle level of a hypercube of n dimensions multiplied by $\lceil n/2 \rceil$, the number of transitions leaving every middle level state. Figure 15 shows how the number of paths (possible paths in the hypercube), states, MCD chains, and MTC chains compare as the number of dimensions increases. The rate of increase of MCD and MTC chains is about the same as the rate of increase of the number of states. The number of paths, however, increases at a much greater rate.

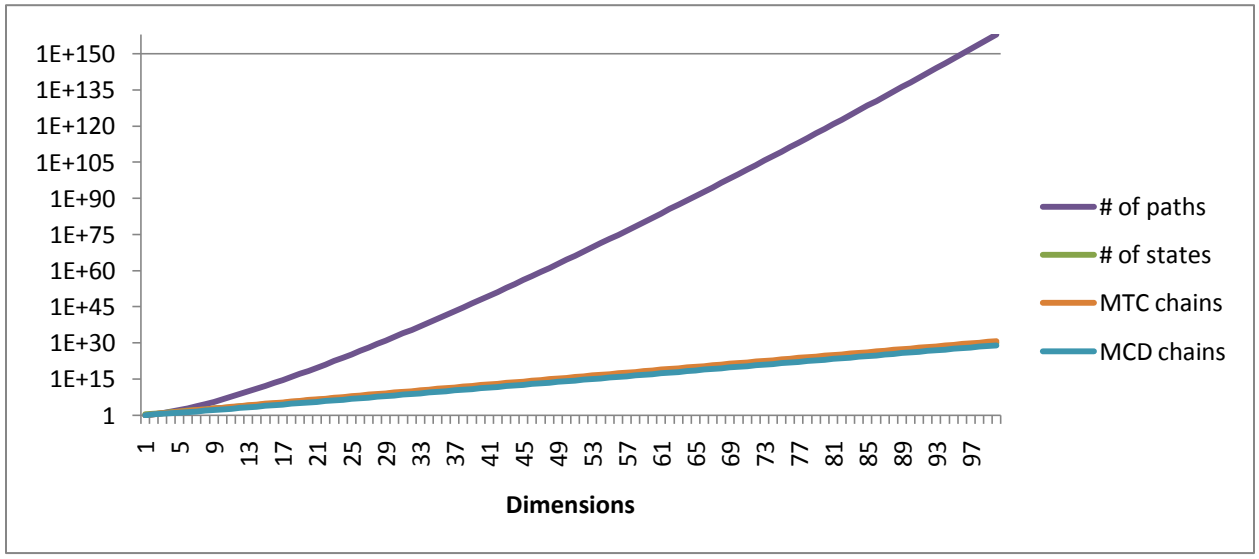

Figure 15: Rate of increase in the number of paths, states, MTC chains, and MCD chains

5.4 Adapting the Strategy

It is inevitable that, while in the process of crawling a web application, there will be a state which contradicts expectations based on the strategy generated by the MTC algorithm. This is because real web applications will not come in the form of a perfect hypercube. As a result, it is necessary to have the ability to adjust the crawling process in order to account for these instances where the actual website deviates from the hypercube structure. In order to do this, there is a need to have a means of identifying these deviations when they occur.

5.4.1 Identifying Deviations

We introduce the four possible scenarios in which the actual website deviates from the projected model. These cases are **appearing events**, **disappearing events**, **merges**, and **splits**. We explain these cases and give criteria for identifying them below. Following this, a method of dealing with them is discussed.

Appearing Events

As a web application is traversed according to the chains produced by the MTC algorithm, it can be determined in advance whether or not the arrival at a new state is expected. If it is the case that we expect to arrive at a new state and indeed do arrive at a new state but find that one or

more events which are available in this state are not included in the list of events that we expected to find, then we classify this case as one where there are appearing events. Figure 16 illustrates how appearing events can be identified. Beginning at state I, we execute event e_1 (from the set of enabled events $\{e_1,e_2,...,e_n\}$) and arrive at state S' which has previously been unvisited. We find that there is a set of appearing events $\{a_1,a_2,...,a_n\}$ which were not anticipated to be available in this state based on the MTC chains that have been produced.

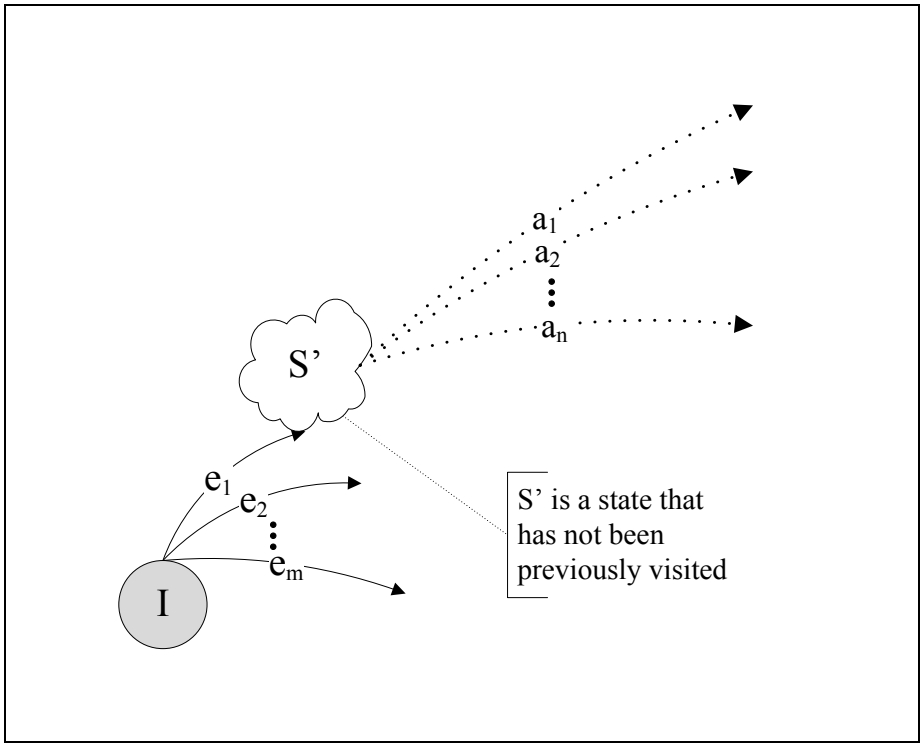

Figure 16: Appearing events

Disappearing Events

Disappearing events are very similar to appearing events in that they occur under the exact same scenario. We expect to arrive at a new state and do arrive at a new state. However, in the case of disappearing events we find that one or more of the events that we expected to find in this state

$\{ d_1,d_2,...,d_q\}$ are *not* available. This occurrence is shown in Figure 17. Arriving at state S', we find that some events, such as d_1 which we expected would be enabled after a transition to state S_x, are not present.

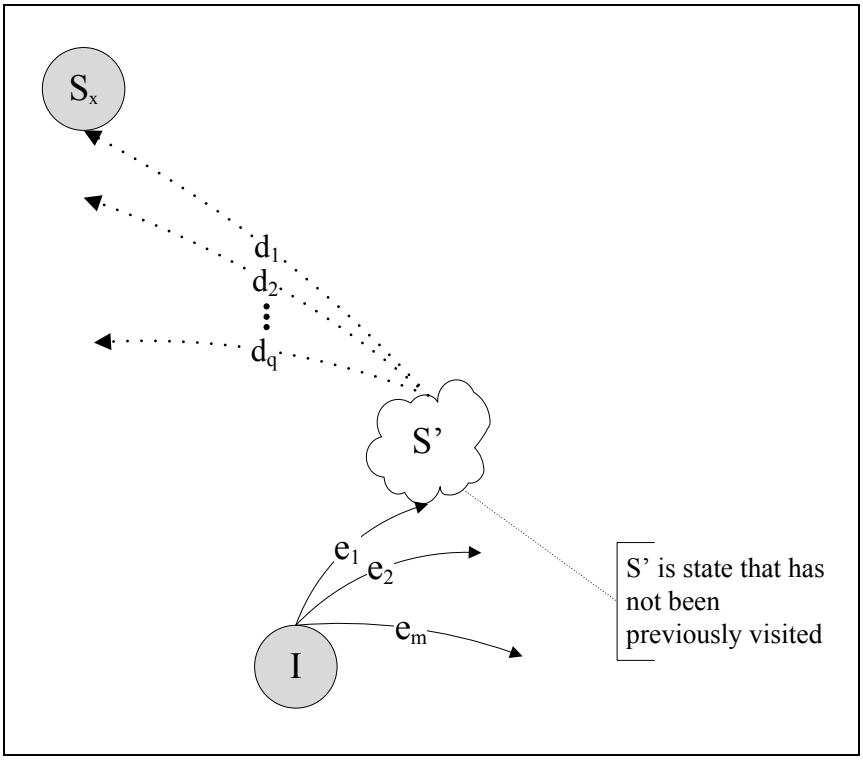

Figure 17: Disappearing events

It is very important to note that these cases (appearing events and disappearing events) are not exclusive. It is certainly possible to encounter a state which exhibits both appearing and disappearing events if the criteria for each case are satisfied when we arrive at a given state. Figure 18 illustrates this scenario

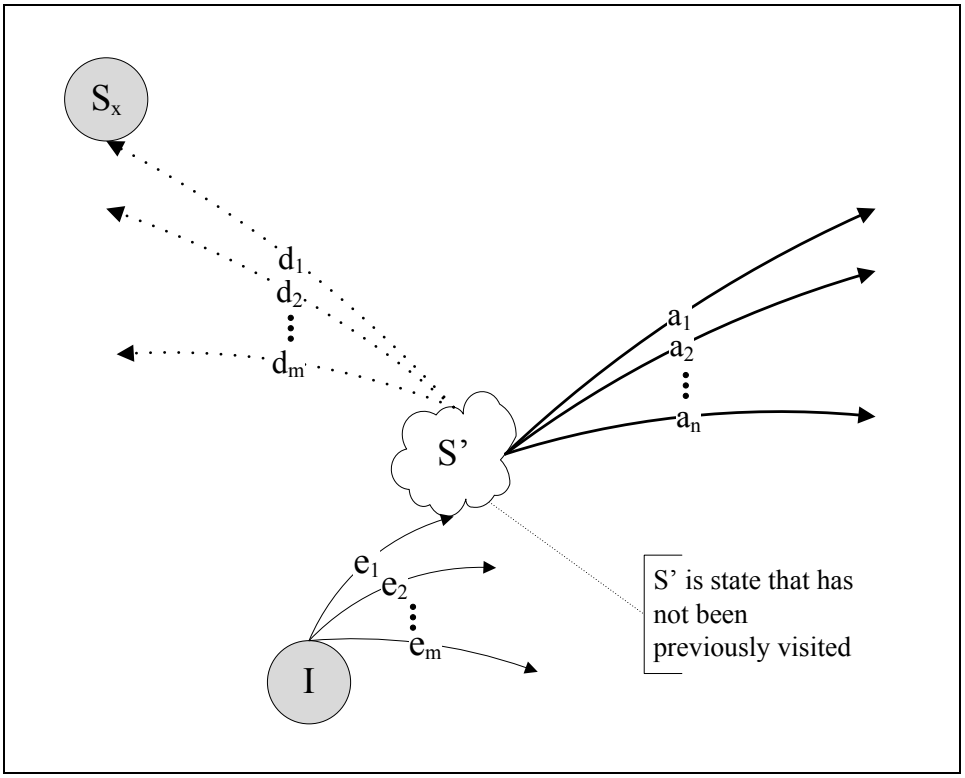

Figure 18: Appearing and disappearing events

Merge

As stated previously, at any given time, the current set of MTC chains can be used to determine what the next state that we encounter should "look" like and whether or not we expect to have previously visited that state. In the case of a merge, it does not matter what the expectations are with regard to whether the next state that we encounter should be one that we have previously visited or not. However, if the state that we arrive at is one that we have indeed been to before but not the one that we expected to arrive at (this means, that we either expected to arrive at a previously unvisited state or a state that has been visited but which is not equivalent to this state), then we say that a merge has occurred. Figure 19 illustrates a merge. We expect that executing event *e* at state I will result in an arrival at state S but this transition instead leads us to state S'.

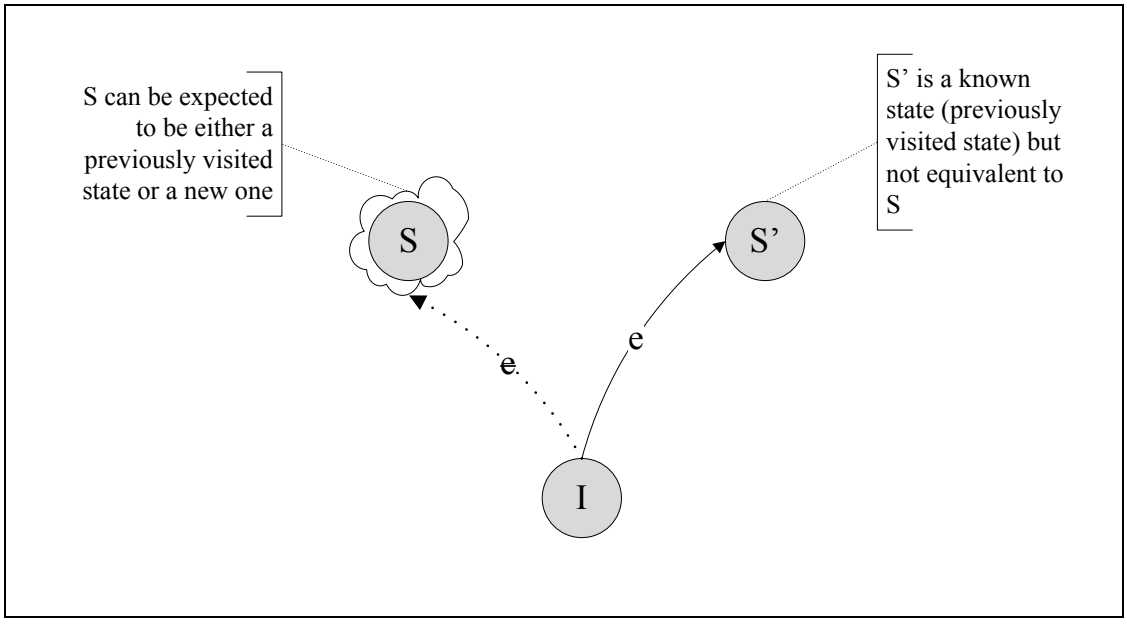

Figure 19: A merge

Split

Identifying the case of a split is very simple. It occurs when we arrive at a new state *but* had expected to arrive at some known state. Figure 20 depicts this case. Taking transition *e* from state I, we arrive at a new state S' although we had expected to arrive at some known state S.

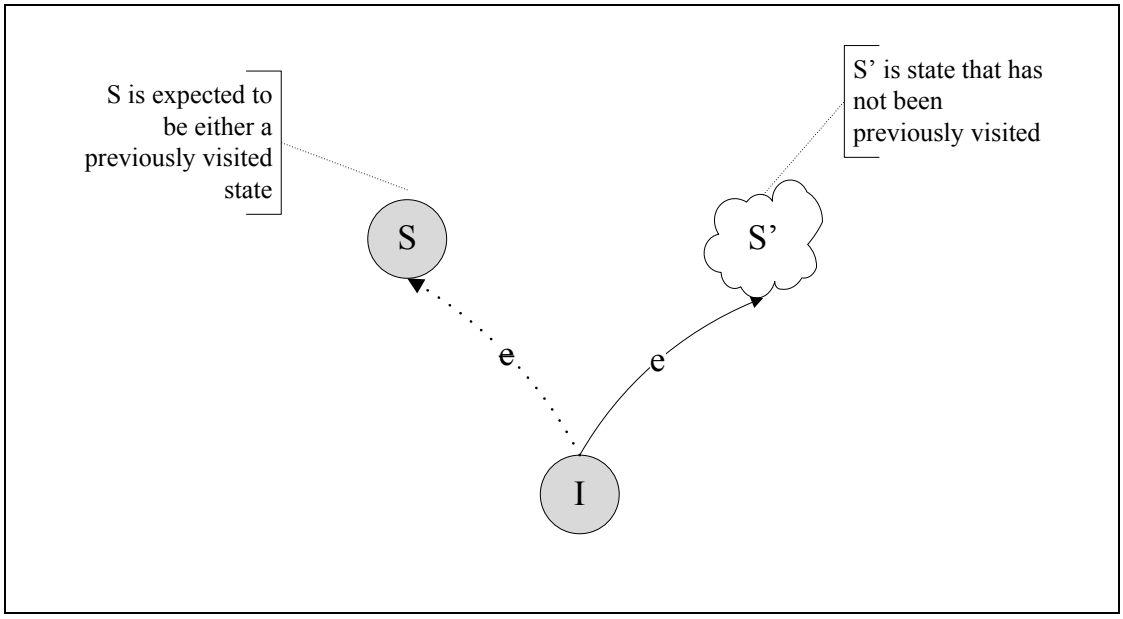

Figure 20: A split

It is also important to point out that while appearing and disappearing events may occur at the same time, merges and splits occur exclusively.

5.4.2 Revising the Strategy

An algorithm which unifies the way in which these cases are handled has been created. In simplified terms, we refer to any occurrence of one or more of the cases previously described as a deviation from the projected model which is then fixed by making the appropriate changes to the crawling strategy. Deviation detection and strategy revision is accomplished using an algorithm *reviseStrategy*, shown in Figure 21.

Procedure reviseStrategy ($Chains(s_1), C, e_i, s'$)
Input $Chains(s_1)$: strategy for the expected model based on s_1
Input $C = s_1 - e_1 - s_2 - \cdots - s_i - e_i - s_{i+1} - \cdots - s_n$: the current chain
Input e_i : The event that has just been executed in C
Input s' : The state reached by executing event e_i at s_i
begin
*/*A deviation has occurred if we expected to arrive at some known state but arrive at a different state OR if we expected to arrive at a new state with a specific event set but arrive at either a known state or a new state with an event set which is different from expectations*/*
if $((s' \not\approx s_{i+1})$ **OR** $(events(s') \neq events(s_i) \setminus \{e_i\}))$ {
    //We attempt to replace each chain which contains the same prefix as the current chain
    foreach ($C' \in Chains(s_1)$ such that $pref_{C'}(s_{i+1}) = pref_C(s_{i+1})$){
        for ($k = i + 1$ to $|C'|$){
            if ($\exists\ C'' \in Chains(s_1)$ such that $s_k \in C'' AND\ (s_i - e_i - s_{i+1}) \notin C''$){
                add chain $pref_{C''}(s_k) + suff_{C'}(s_k)$ to $Chains(s_1)$;
                if ($last(C'') = s_k$)
                    remove C'' from $Chains(s_1)$;
                break;
            }
        }
        remove C' from $Chains(s_1)$;
    if (s' is unknown) {
        Generate $Chains(s')$ for the new hypercube based on s';
        add $Chains(s')$ to $Chains(s_1)$ using $T(s') = pref_C(s_i) + (s_i - e_i - s')$;
    }
}
else if($\exists\ hypercube(s_p) \neq hypercube(s_{i+1})$ such that $s' \approx s_p$){
    remove all chains C' such that $pref_{C'}(s_{i+1}) = pref_C(s_{i+1})$ from $Chains(s_1)$;
end;

Figure 21: Procedure reviseStrategy

For a set of MTC chains (written as $Chains(s_1)$, which are generated based on the events enabled in s_1), we denote the chain that we are currently crawling as $C = s_1 - e_1 - s_2 - \cdots - s_i - e_i - s_{i+1} - \cdots - s_n$. Within this chain, e_i represents the event that has just been executed. The task is to determine whether or not a deviation has occurred based on s', the state which resulted from executing e_i. This determination depends on whether s_{i+1} has already been visited or not. If it is the case that s_{i+1} represents a state that has previously been visited, a deviation has occurred if s' is not equivalent to s_{i+1} (the condition $s' \approx s_{i+1}$ has failed). If s_{i+1} is supposed to represent a state that has not yet been visited, then based on the hypothesis of events being independent, we expect that s_{i+1} will contain all of the events that were available in the previous state (s_i) minus e_i. Therefore, if s' does not match this expectation a deviation has occurred (the condition $events(s') \neq events(s_i) \setminus \{e_i\}$ has failed where $events(s_i)$ denotes the set of events enabled at state s_i).

If we find that s' represents a deviation, we must update the chains in order to ensure that the strategy is consistent with the model that has been uncovered thus far. To illustrate what this means, it is important to discuss how a deviation impacts the strategy. When s' does not match expectations, it means that executing transition e_i at state s_i results in the discovery of some state that is not equivalent to s_{i+1}. The interpretation of this is not that it indicates that s_{i+1} does not exist or that it is not possible to reach s_{i+1}. Instead, it may just mean that we cannot reach s_{i+1} by using $pref_C(s_{i+1})$, which is the sub-path that we took attempting to reach s_{i+1} (shown in

Figure 22). This also means that any other chain that contains this same prefix will also not be able to use that prefix to reach s_{i+1}.

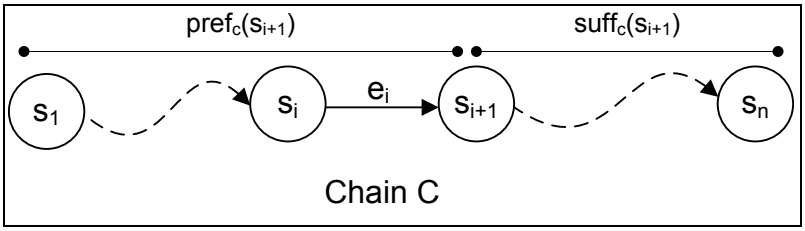

Figure 22: The prefix and suffix of a chain

In order to repair these chains so that they may be completed, we must first find some chain which includes state s_{i+1} but not the problematic sequence $s_i - e_i - s_{i+1}$. In other words, we must find an alternate route to state s_{i+1}. If we do find such a chain (C''), we must then replace every chain (C') which contains the prefix $pref_C(s_{i+1})$ with a chain consisting of $pref_{C''}(s_{i+1}) + suff_{C'}(s_{i+1})$. This would potentially allow us to reach state s_{i+1} in each case and would also allow us to complete the other transitions in the chain.

Another issue that may arise is that there may be no other chain which contains an alternate route to state s_{i+1}. In that case, for every chain(C') that needs to be repaired we instead try to find an alternate route to the next state (s_{i+2}). We do this until we come to a state for which we can find an alternate route or until we come to the end of the chain. If we come to the end of the chain without having successfully found an alternate route, then we are unable to repair the chain and simply remove it.

Responding to a deviation is not simply a case of repairing chains. We may find that s', the state in which we discovered a deviation, includes events which were not available in s_1, the base of the hypercube. This could be found both when there are appearing events and in the case of a split. It also means that s' is outside the scope of the initial hypercube. We therefore consider this to be an indication that we have a new hypercube with s' as its base. We also create a new hypercube whenever we arrive in a previously unvisited state which does not have the set of events which we expect. A new hypercube needs to have its own strategy generated and also be explored. It also extends from the initial hypercube and is not reachable by URL. We can reach the base of this hypercube using $pref_C(s')$.

In the case that s' does not represent a deviation, there may still be some "cleaning up" that needs to be done. If s' does match s_{i+1} within the current chain (C) of this hypercube strategy but is also equivalent to a state which exists in a separate hypercube, then we should remove all of the chains from the current hypercube strategy that share this same prefix, $pref_C(s')$. This is because this state would have already been accounted for in the strategy of another hypercube so we do not want to duplicate the exploration of the states and transitions that follow this state.

Choosing the Next Chain

If the application turns out to be a perfect hypercube then we will only need to generate the initial set of MTC chains in order to successfully uncover all states and use all transitions. In that case, during the course of a crawl, when we execute all events in a given chain the next chain that is selected will simple be the next chain in the sequence. Since the MTC chains are already organized to satisfy the priorities of first reaching all new states then using all unused transitions, no additional logic is needed for this selection.

However, this will likely not be the case. Instead, while crawling a given hypercube we will find that deviations occur and result in the need for chains to be repaired. In this case, it is necessary to have a technique for selecting the next chain since after revising the strategy the order of the chains may no longer reflect the established priorities. We select the next chain to crawl in a given hypercube based on whether or not all of the states in the current hypercube have already been visited. Here is the criteria based this factor:

1. If there are still unvisited states in the expected model (for a given hypercube), we select the chain for which the value *unvisited(C)*, the number of unvisited states in that chain, is greatest. Therefore, the chain that is chosen is simply chain C that satisfies the following condition:

   $$NOT\ \ EXISTS\ chain\ C' AND\ unvisited(C') > unvisited(C).$$

   This chain may or may not be a constraint containing chain.

2. If all states in the expected model (for a given hypercube) are already visited then we select the chain C for which *untraversed(C)*, the number of untraversed transitions in that chain, is greatest. The chain that is chosen is therefore the one that satisfies the following condition:

$$NOT\ \ EXISTS\ chain\ C'AND\ untraversed(C') > untraversed(C).$$

Choosing the Next Hypercube

Once multiple URLs (with enabled events) have been visited, there will be multiple base states in the list B. One option could be to crawl the hypercubes of these base states in order. That is, we could crawl all hypercubes associated with a state *s* in B before removing it and crawling all the hypercube associated with the next state in B. Another option is to make the choice of which base state will be explored (which group of hypercubes associated with a base state) before making a choice about which particular hypercube and chain will be explored. These choices can be made before every decision to choose a chain. That is, for each *k*, we would first choose the group of hypercube to explore and then choose the hypercube to explore.

Again, a priority-based system ($priority(s)$) is employed for this purpose. One possible formula that can be used to calculate the priority of a hypercube group is essentially the same as the one which is used to select the next chain to crawl. That is, we select the hypercube group (G) which

contains the hypercube having the chain (C) with the most unvisited states. We select the hypercube group (G) for which the following condition is true:

$$NOT\ \ EXISTS\ group\ G'\ WHERE\ C'\ in\ G' AND\ C\ in\ G\ and$$

$$unvisited(C') > unvisited(C).$$

We have the option of using many different priority functions but we believe that these resonate with the goal of exploring new states first, followed by new transitions. We believe that this would be the case when we select hypercube groups which contain the chains with the most unvisited states.

Summary of Event Based Crawling

Having discussed the components of the event-based crawling strategy as well as how they work in collaboration, the strategy can be summarized by the procedure *eventBasedCrawl(L,B)* shown in Figure 23. Whenever this procedure is called, we first generate chains for any base state *s* in B for which chains have not yet been generated (using the algorithm *minimumTransitionCoverage*). We then choose a base state with the highest priority and determine which chain associated with that base state will be explored next. Once we have chosen a state, we explore it until we arrive at the end or encounter a deviation (which is identified by and handled by the procedure *reviseStrategy*). If we have arrived at the end of the chain, we remove it from Chains(s).

Procedure eventbasedCrawl (L, B)
Input L : set of URLs that are to be visited
Input B: base states
begin
foreach(state $s \in B$ and Chains(s) has not yet been generated)
{
    generate Chains(s);
}
choose a state $s \in B$ such that $\forall s' \in B\ priority(s) \geq priority(s')$; //states
determine the next chain $C \in \text{Chains(s)}$ to execute;
executeChain(C, B, L);
remove C from Chains(s);
end;

Figure 23: Procedure eventBasedCrawl

6 Prototype Tool for Crawling AJAX-based Web Applications

6.1 Design and Implementation

We have developed a prototype crawling tool which implements the event-based crawling strategy. The prototype tool is capable of crawling test AJAX applications and is able to collect statistics related to the crawl.

The prototype crawling tool is implemented in Java and developed using the Eclipse IDE [49]. Java was selected mainly because the frameworks which were selected to aid in development are implemented in this language. These frameworks are:

HtmlUnit: HtmlUnit [50] is an open source framework which can be summarized as a web browser for Java programs. It can interact with web pages and simulate the actions that would normally be completed by a person using a web browser. It also has fairly good JavaScript support, which is important in order for most web applications to work correctly (also required for AJAX requests to be possible). Given that it is open source, it can be extended to support future developments in this research.

XmlUnit: XmlUnit [51] is a framework which makes it possible to unit test XML documents. It provides an API which allows Java programs to quickly compare XML

documents. For example, it can determine if two documents are identical or similar (have small differences such as the ordering of nodes). It also allows such comparisons to be made for HTML documents.

Jung: Jung [52] is an open source graphing framework which provides a library that allows easy visualization of data. It contains built-in support for producing a graph which illustrates the data. It also allows graphs to be animated as changes are made to the elements of the graph or its layout.

Another reason for selection Java is because using an object oriented programming language makes it easier to integrate the research with IBM's existing product.

In the prototype crawling tool, the overall crawling process is handled by the class *AjaxCrawler* which is located in the *Crawl* module. This class contains the procedure *eventBasedCrawl* (detailed in Figure 23). In addition, *AJAXCrawler* communicates with classes from five modules which enable the ability to perform the crawl, and track and display related results. These modules are *WebBrowser*, *Strategy*, *Modeling, Equivalence*, and *Statistics*. The architecture of the tool, including these modules and the most important classes, are shown in Figure 24.

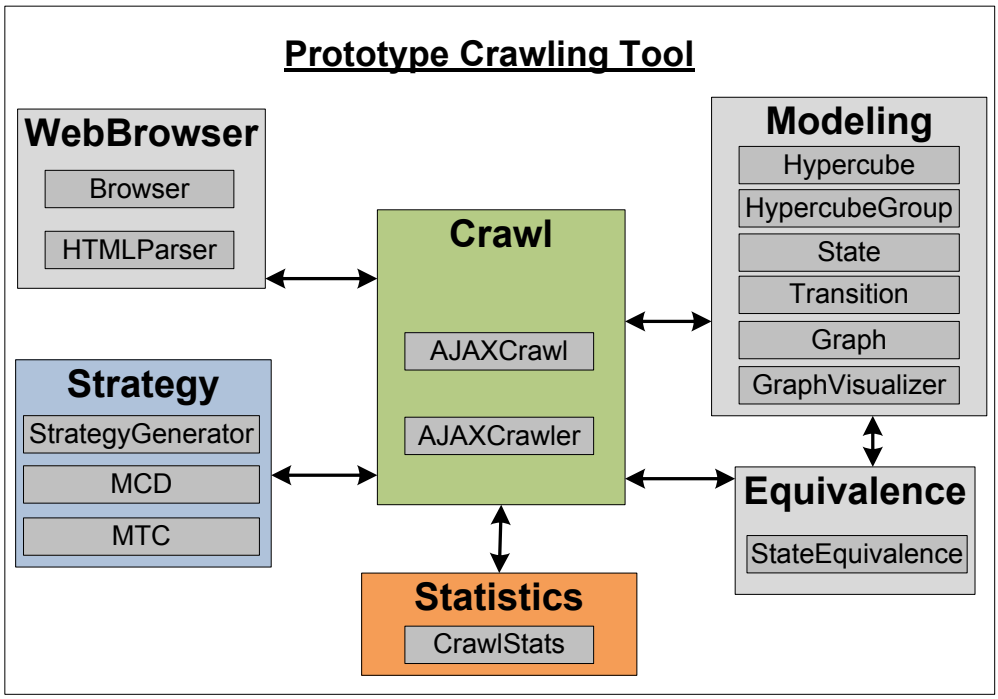

Figure 24 : The modules and selected classes of the prototype crawling tool

WebBrowser

The *WebBrowser* module is responsible for all actions that would normally be completed by the browser. It is implemented in part using the API provided by HtmlUnit. The class *Browser* within this module provides the ability to send an HTTP request to the server, given a URL. Once the response is received from the server, it loads the corresponding page. This class also handles event execution. For the handling of AJAX calls, HTMLUnit provides an AJAX controller (*NicelysynchronizingAjaxController*) class which ensures that the next line of code in the program does not get executed until a response has been received and the DOM updated. The class *HTMLParser* parses the DOM to identify various elements based on attributes, such as

their *id*, or the values of those attributes. This allows easy identification of elements which trigger events of interest.

Strategy

The *Strategy* module contains the *MCD* and *MTC* classes. These contain all the algorithms used to generate the MCD and MTC chains associated with the event-based crawling strategy. The class *StrategyGenerator* uses these classes to produce chains based on a hypercube. It also uses the procedure *reviseStrategy* (described in Section 5.4.2) to replace chains when deviations occur. The *StrategyGenerator* is also responsible for determining which events are executed and in what order.

State Equivalence

The *Equivalence* module provides all functionality related to determining whether or not two DOMs are equivalent. Using the DOMs provided by the *Browser,* the class *StateEquivalence* determines whether or not the current state is equivalent (based on the equivalence function) to one which has previously been visited. It also uses the concept of "load, reload" (discussed in Section 4.2) to identify the portions of the DOM that can be ignored. This module is implemented with the aid of the API provided by XMLUnit.

Modeling

This module keeps track of the model that has been discovered. It maintains information about the states and transitions that have been discovered, and the various hypercubes that have been generated. Information stored includes the number of states that have been discovered in a particular hypercube versus the number of states that are currently expected to be found in that hypercube. This type of information can be used when computing the next chain to crawl based on some priorities.

The *Modeling* module also leverages JUNG in the class *GraphVisualizer*. This class produces graphs that allow manual positioning of states and transitions (resulting from a crawl). This means that graphs can be arranged in a way which makes it easy to visually compare the results of the crawl with the known model of the test web application being crawled. Of course, this feature is only useful for comparing the output of crawling web applications with a small number of states. Graphs elements are also labeled. States are labeled with a unique ID which is given to each state. Transitions are labeled with the element on which the event was executed.

Statistics

The *Statistics* module consists of one class (*CrawlStats*) that records statistics during the crawl. The class keeps track of data such as the total number of transitions and the total number of

resets performed. It also records the number of transitions and resets that have been completed at the arrival of each new state and is able to display a summary of these statistics at various points during the crawl and after completion.

Communication

The sequence diagram shown in Figure 25 represents an example of communication between the different classes in the prototype crawling tool and gives a simplified view of how the prototype crawling tool works. Execution begins in *AJAXCrawl*, which initializes the *AJAXCrawler* for a given start URL, and calls the *Crawl* method to begin the crawl. The *AJAXCrawler* then calls the method *LoadPage* on the class *HTMLParser*. Once the page is loaded, the DOM and all available events are returned. The *GenerateStrategy* method is then called on the *StrategyGenerator*, and the initial set of MTC chains is produced. The events which need to be executed are then return to the *Crawler*.

At this point the program enters a loop in which the *AJAXCrawler* first calls the *ExecuteEvents* method on the *Browser*. The events are executed by the *Browser* and the resulting *DOM* is returned. Then there is a check for duplicate states using *StateEquivalence*. If the transition that was just executed is a new transition (this was the first time that it had been executed) it is added to the graph using the method *AddTransition*. The method *CheckForDeviations* is then called and the strategy revised. The *StrategyGenerator* then returns the next events to be executed and

the next iteration of the loop begins with *ExecuteEvents* being called again. If the end of the current chain has been reached meaning a new chain will be crawled, a URL is provided as a parameter for this method, and the *Browser* reloads the specified page before executing the event(s).

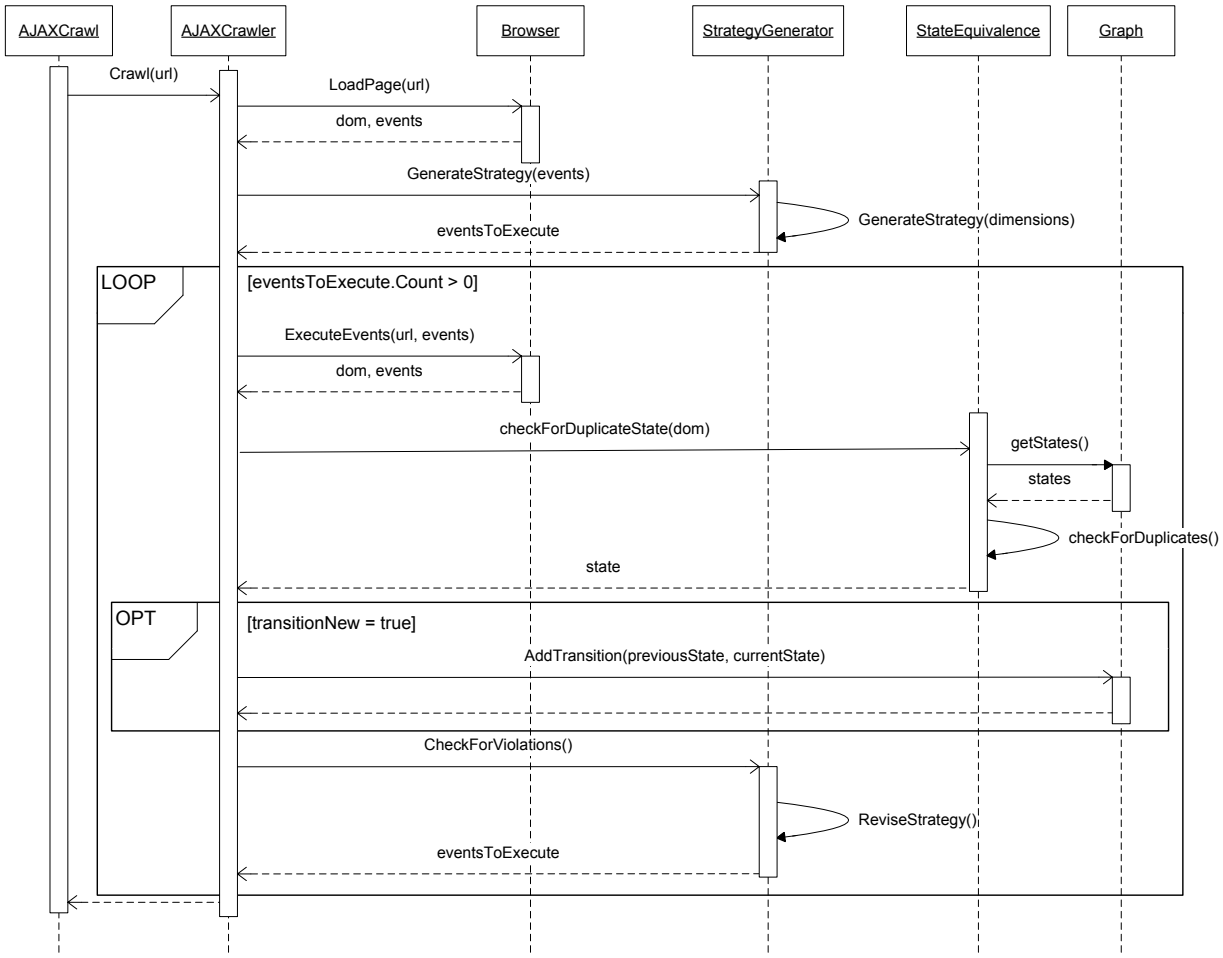

Figure 25: Sequence diagram showing the communication between the classes of the crawler

6.2 Limitations of the Prototype Crawling Tool

The prototype crawling tool has the following limitations:

Inconsistent resynchronization of AJAX calls:

HtmlUnit's *NicelysynchronizingAjaxController* class (discussed in Section 6.1) enables resynchronization of AJAX calls but it has been observed that there are instances when the next line of code gets executed before the response from the server has been received and/or before callback method execution is complete. As a result, there are cases in which events are executed but the DOM is not updated as expected. As a work around, a "sleep" delay of 6 seconds is used in these cases. This causes program execution to pause until the DOM has been updated. However, this is an awkward and unreliable solution since a delay of 6 seconds is more than required in some cases but there can be no guarantee that it will be a sufficient delay in every case. Also, this work around is only feasible when crawling smaller applications since it causes a significant increase in the duration of the crawl. Therefore, the problem will need to be addressed in order to enable support for a larger set of web applications going forward.

No support for intermediate states:

In the current implementation of the prototype crawling tool, there is no support for capturing intermediate states (described in Section 3.1).

Strategy generation limited to a maximum of 16 events:

The prototype crawling tool is able to generate MTC chains given a base state with up to 16 enabled events. If a base state has more than 16 enabled events, the prototype crawling tool is unable to successfully generate MTC chains due to insufficient memory. This problem is due to the event based strategy's current requirement that all chains for a given hypercube be generated up front. For a base state having 18 enabled events, this would require 437,580 chains to be generated.

Only *onclick* events are supported:

The initial version of the prototype crawling tool does not support other types of events such as *mouseover* events and events which are triggered when a specific amount of time passes.

In spite of the current limitations, the prototype crawl tool is still very useful since it allows initial testing of the event-based crawling strategy. This means that the real-world consequences of the drawbacks of the strategy can be observed. It also allows the strategy to be compared with other strategies.

6.3 Integration with AppScan

Following the development of the initial prototype, Emre Dinçtürk[1] and I worked to implement components of the MTC-based crawling algorithm with the current AppScan product. In order to

[1] Emre Dinçtürk is a PhD candidate at the University of Ottawa.

accomplish this assignment, there was a need to re-implement the core algorithms since AppScan is developed using C#. In addition, the logic of the crawler needed to be updated to fit the work flow of that product. Integrating the tool 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 and security testing).

Following 2 months of work at IBM, components of the MTC-based crawling algorithm were successfully added to the current AppScan product. This produced a prototype AppScan which is capable of utilizing portions of the event-based strategy for state discovery. When crawling AJAX applications, AppScan is now able to discover a significantly larger number of states than before. An initial demonstration of these increased capabilities has already been conducted for members of the AppScan team.

7 Experiments and Evaluation of Results

This chapter is divided into two parts. In Section 7.1, "load, reload" (see Section 4.2) is tested. In Section 7.2, experiments are conducted to evaluate the performance and potential of the crawling strategy. Particularly, it is important to see how quickly the initial set of chains (MTC chains) is generated, the maximum number of events (dimensions) that the prototype crawling tool can handle, and the ability of the tool to model applications. We are also interested in the performance of the strategy based on the number of transitions and resets required to discover each state or transition in an application. In experiments, the performance of the prototype crawling tool (which utilizes MTC-based strategy generation) is also compared to the performance of a crawler which has been implemented using a breadth-first crawl strategy and one which uses a depth-first crawl strategy.

7.1 "Load, Reload"

The "Load, Reload" technique is tested using 30 popular websites (listed in Appendix A: Test Websites for "Load, Reload"). First, the URL of each website is loaded twice and the pages compared after the second load to determine whether or not the pages have some differences. Following this, the "load, reload" technique is used on each website to determine how many of the web pages which had been different after consecutive loads, would now be considered

identical. When loading pages, 10 seconds are allowed to pass before the page is loaded an additional time.

After loading each URL twice and comparing the pages, only 4 of 30 websites (13.33%) produce an identical page when the URL is loaded the second time. When the "load, reload" technique is used, 22 of the 30 websites (73.33%) produce a page which is identical when then URL is loaded the second time and irrelevant portions of the page ignored. This additional 18 pages which are identical after using "load, reload" represents an increase of 450%. However, there are still 8 pages which are not identical even after the use of this technique. One potential reason for this is the duration between reloads. For example, an application may have a page which displays a new advertisement every 25 seconds. Therefore, if a page is loaded at time $t = 0$ seconds, and then reloaded again at $t = 10$ seconds, the advertisement may still be the same and therefore not automatically considered irrelevant content. If the page is then loaded again at $t = 35$ seconds, a new advertisement may cause this page to be classified as not identical even though "load, reload" is used. This technique therefore does not remove irrelevant content in all cases. However, as the statistic regarding the increase in identical pages identified shows, "load, reload" is still very useful in limiting the number of states that will have to be further evaluated for equivalence.

7.2 Crawling Strategy

7.2.1 Strategy Generation

This test determines the maximum number of concurrent events enabled in a given state (dimensions) that the prototype crawling tool can handle. That is, the largest hypercube for which it can create a set of MTC chains. Results show that the prototype crawling tool is currently able to generate MTC chains for a maximum of 16 events. This requires the production of 102,960 MTC chains. At 16 dimensions, an "out of memory" exception is observed. These results are based on testing using a machine running Windows Vista with 2GB RAM and a 2.10 GHz Intel Core 2 Duo CPU.

An experiment is also conducted to find the length of time taken to generate MTC chains for a hypercube of *n* dimensions. For dimensions 1 - 16, the prototype crawling tool is used to generate MTC chains at each dimension and the time taken to generate those chains is recorded. Table 1 shows this data and Figure 26 shows a graph of these values.

Dimensions	**1**	**2**	**3**	**4**	**5**	**6**
Chains	1	2	6	12	30	60
Time(Seconds)	0.00	0.004000187	1.00E-03	0.002000093	0.006999969	0.005000114

Dimensions	**7**	**8**	**9**	**10**	**11**	**12**
Chains	140	280	630	1260	2772	5544
Time(Seconds)	0.010999918	0.023000002	0.059999943	0.13499999	0.347000122	0.81099987

Dimensions	**13**	**14**	**15**	**16**
Chains	12012	24024	51480	102960
Time(Seconds)	2.177999973	4.305999994	9.376999855	20.48900008

Table 1: Time taken to generate MTC chains

for dimensions 1 – 16

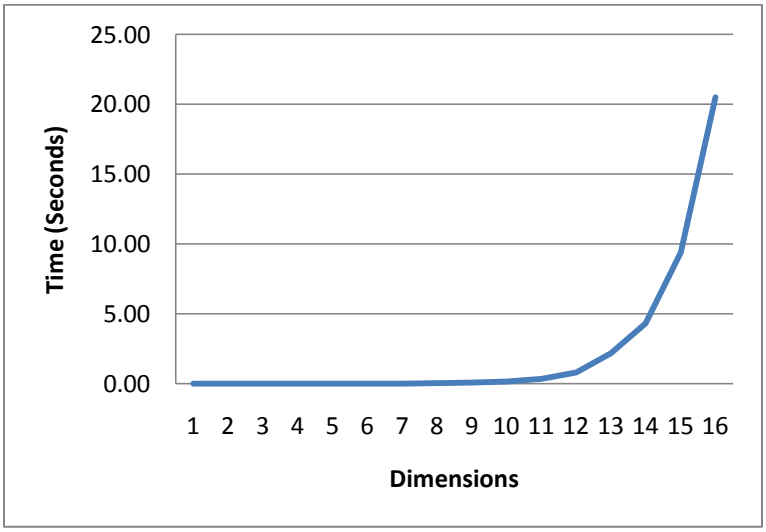

Figure 26: Time taken to generate MTC chains

for dimensions 1 – 16

7.2.2 Model Building

Before determining the efficiency of the crawling strategy, it is necessary to verify that the prototype crawling tool is able to use this strategy to craw some test AJAX-based applications and produce the correct model. This is done by testing the prototype crawling tool with some applications and comparing the resulting model (the model created using the prototype crawling tool) with the known model of the application. These applications as well as the results of the crawls are detailed below:

4 Dimensional Hypercube Web Application

The prototype crawling tool is used to crawl a web application which follows the structure of a hypercube of 4 dimensions. The application therefore has 16 states and 32 transitions. Figure 27 displays the actual model of the application (on the left) and the model produced by the prototype crawling tool (on the right). A visual comparison confirms that the model produced matches the actual model of the application.

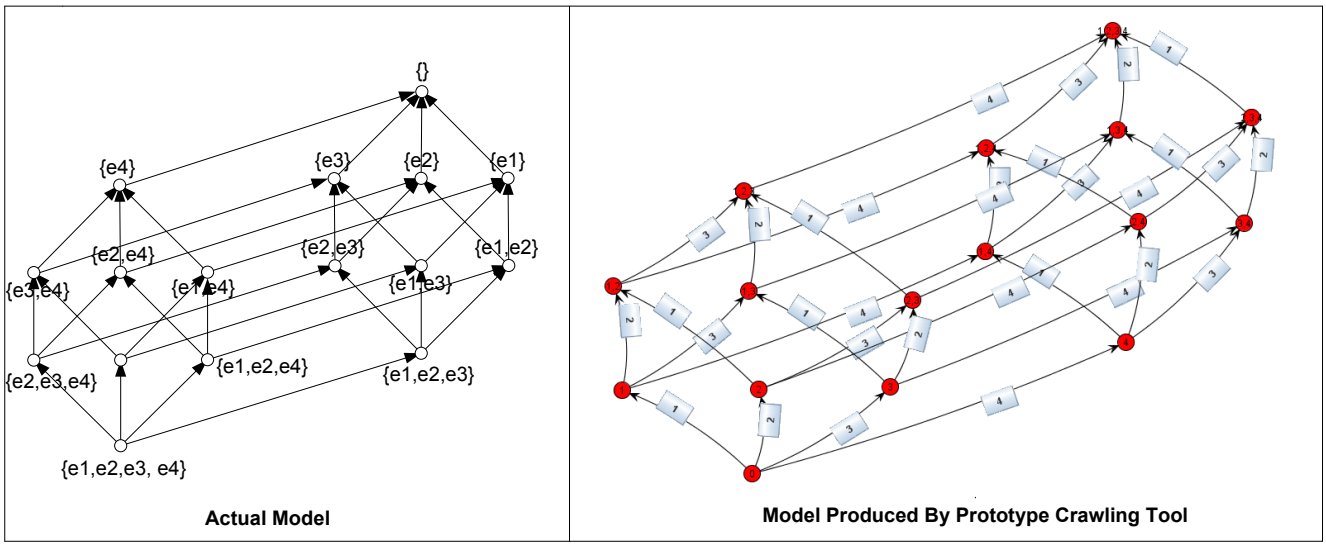

Figure 27: 4 dimensional hypercube web application - actual model vs. created model

Non-hypercube Web Application #1

This application does not follow the structure of a hypercube. It has 8 states and 12 transitions. Figure 28 shows that the crawling tool is able to crawl the application and produce the correct model.

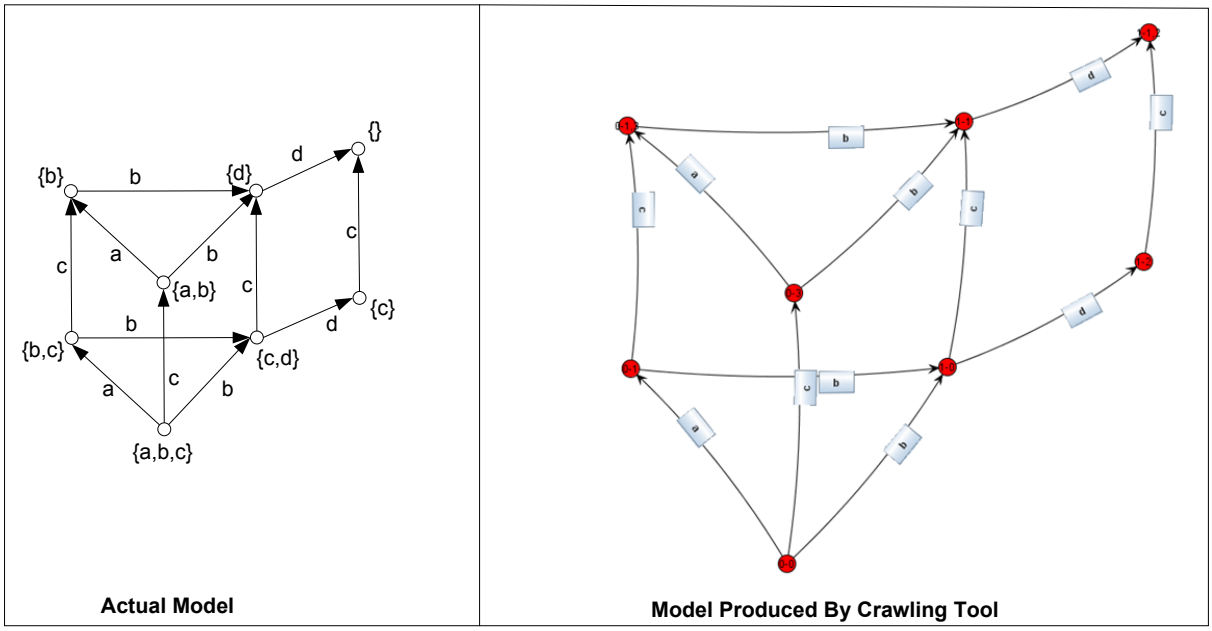

Figure 28: 4 Non-hypercube web application #1 - actual model vs. created model

Non-hypercube Web Application #2

This application has 13 states and 15 transitions. Figure 29 shows that the crawling tool is again able to crawl the application and produce the correct model.

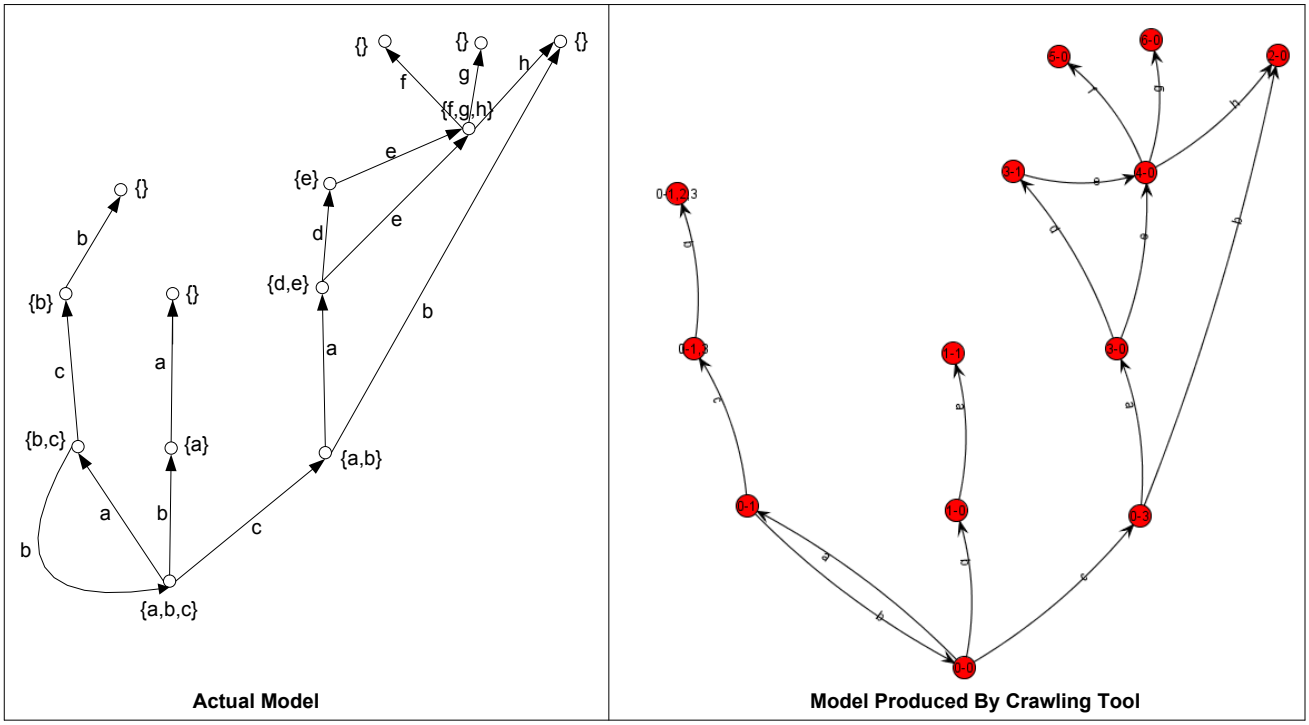

Figure 29: Non-hypercube web application #2 - actual model vs. created model

Non-hypercube Web Application #3

This application consists of 24 states and 32 transitions. The prototype crawling tool produces the correct model. Figure 30 shows that the model produced by the prototype crawling tool is accurate.

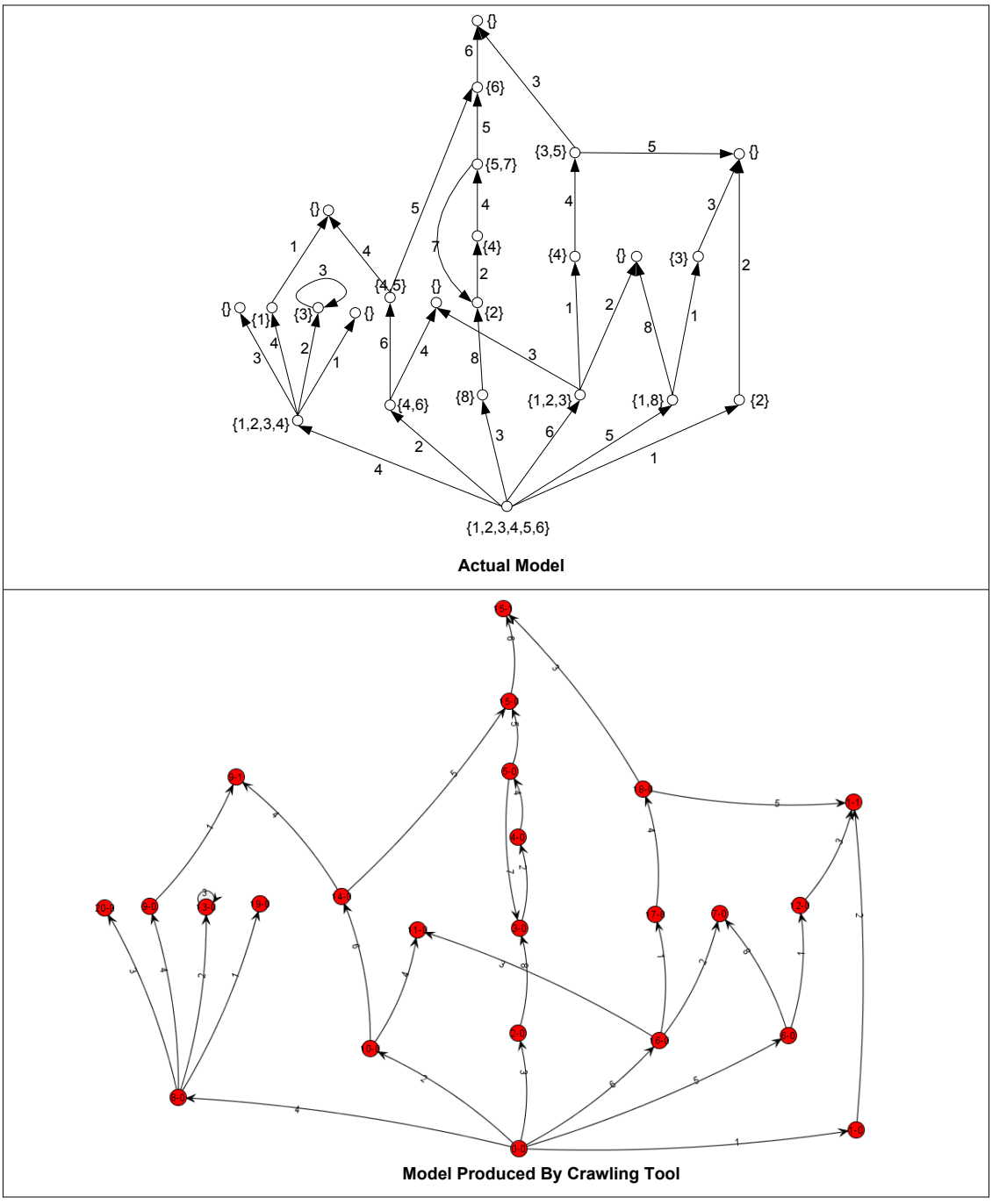

Figure 30: Non-hypercube web application #3 - actual model vs. created model

Non-hypercube Web Application #4: Previous, Next

This application consists of a series of states which are linked by “previous” and “next” buttons.

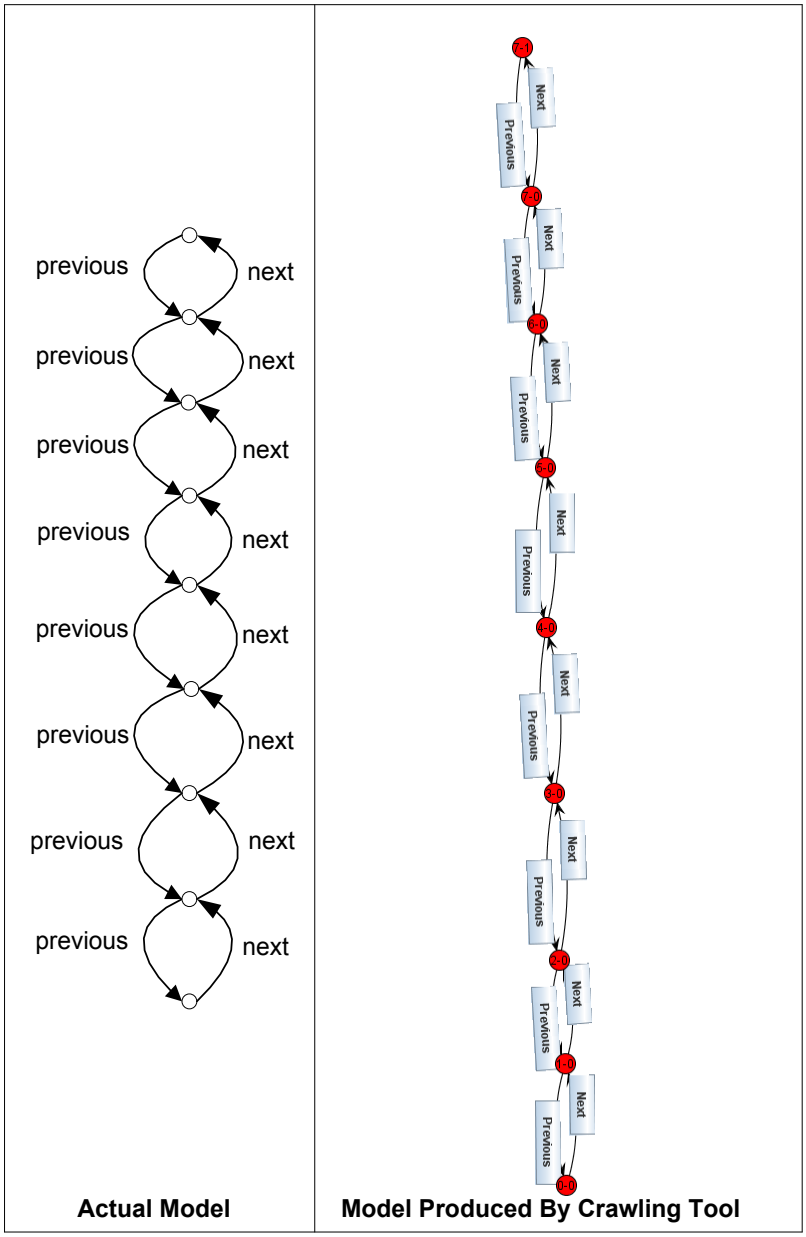

Figure 31: Non-hypercube web application #4: Previous, Next - actual model vs. created model

Non-hypercube Web Application #5: AJAX News

The final application tested is a publicly available test AJAX application [53] developed by [19]. It consists of 8 states, each displaying a different news article. The application consists of "previous" and "next" buttons which allow the user to cycle through the articles. Additionally, the title of each article is listed in every state. Therefore, the user can access any article (state) from any state. The model of this application would therefore be a fully connected graph. Figure 32 illustrates this with the model produced by the prototype crawling tool.

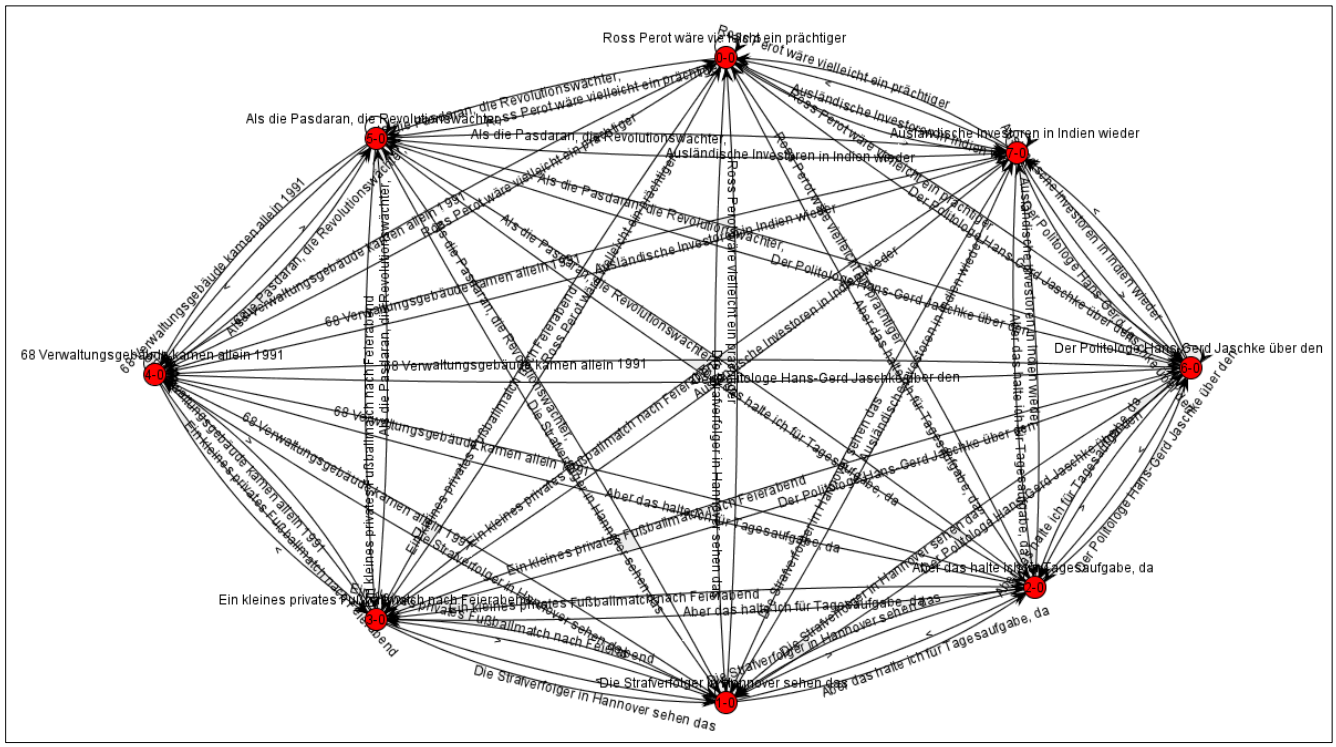

Figure 32: Model of non-hypercube web application #5: AJAX News

7.2.3 Crawling Efficiency

Comparison testing is performed on the applications presented in Section 7.2.2. The event-based crawling strategy presented in this thesis (which uses the **MTC** algorithm to generate the initial set of chains for each hypercube) is compared to a breadth-first crawling strategy and a depth-first crawling strategy. In an effort to ensure that results are not influenced by a specific ordering of the events in each state, the events in each state are randomly ordered for each crawl and each web application is crawled 10 times with each strategy. The results presented for a given strategy's performance for a specific application indicate the average of these 10 crawls. The results are summarized in this section while tables showing the full results can be found in Appendix B: Crawling Strategy Comparisons.

The results of these tests present the following statistics:

- How quickly new states are discovered (visited). This is tracked by:
    - The total number of transitions required before discovering each state.
    - The total number of resets required before discovering each state.
- How quickly new transitions are discovered (executed). This is tracked by:
    - The total number of transitions required before discovering each transition.
    - The total number of resets required before discovering each transition.

In testing, a reset is performed by reloading the page at the URL of the base state. However, as discussed in Section 3.2, this method may not be sufficient for many web applications. Therefore, the number of steps required to reset an application may vary.

Given that the strategy is expected to perform at its best in the case of a hypercube application (a web application which follows the structure of a hypercube), the strategies are first compared by crawling hypercube web applications.

4 Dimensional Hypercube Web Application

In Figure 33 and Figure 34 the MTC-based strategy is shown to outperform the breadth-first and depth-first strategies both for transitions required to visit each state of the application and for total transitions required to execute each transition of the application. The MTC-based strategy is able to find all states in an average of 20 transitions whereas this is done in 68 transitions using breadth-first and 47 transitions using depth-first. The MTC-based strategy also requires less resets to find all states (5), compared to breadth-first (28) and depth-first (15).

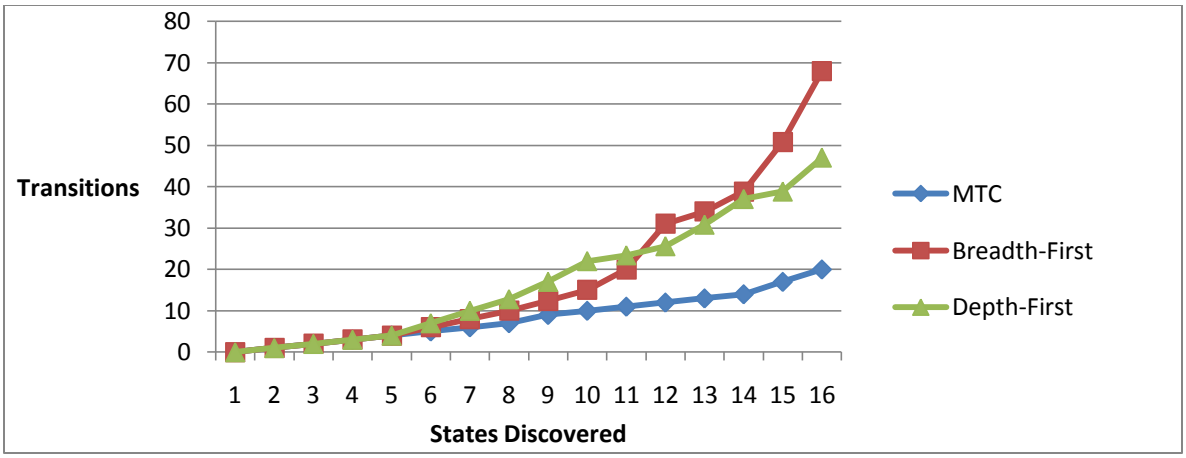

Figure 33: Transitions vs. states discovered (4 dimensional hypercube web application)

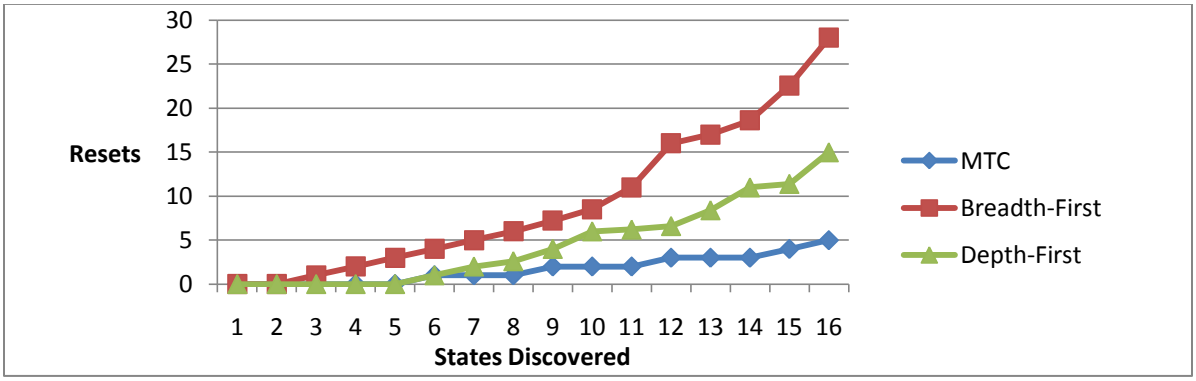

Figure 34: Resets vs. states discovered (4 dimensional hypercube web application)

Figure 35 and Figure 36 show that the MTC-based strategy allows visiting all transitions before the breadth-first and depth-first strategies. It takes 40 transitions and 11 resets while using breadth-first takes 80 transitions and 31 resets and depth-first takes 52 transitions and 17 resets.

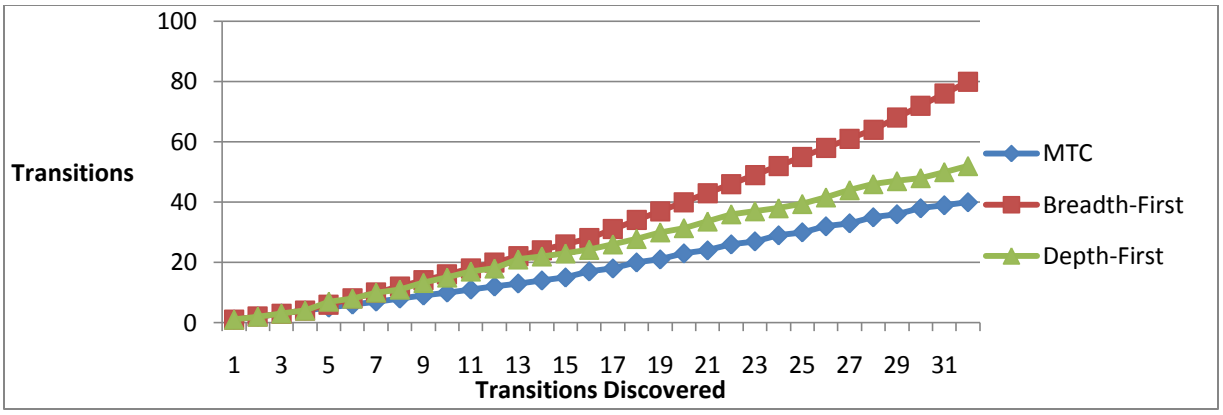

Figure 35: Transitions vs. transitions discovered (4 dimensional hypercube web application)

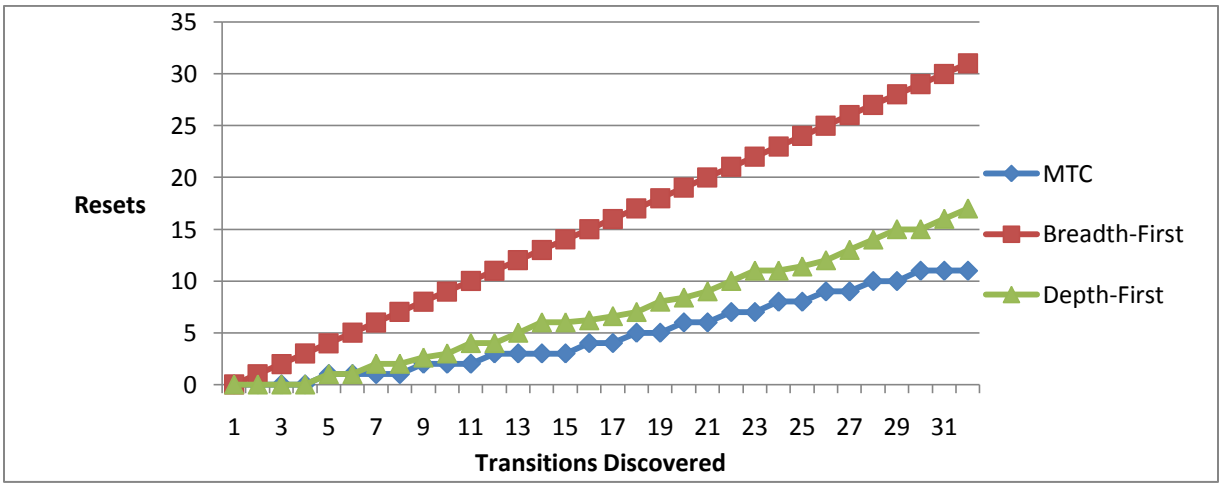

Figure 36: Resets vs. transitions discovered (4 dimensional hypercube web application)

Also, the MTC-based strategy makes it possible to complete the crawl in 40 transitions and 11 resets whereas this takes 80 transitions and 31 resets using breadth-first and 52 transitions and 17 resets using depth-first.

Testing is also completed on hypercube applications of 3, 5 and 6 dimensions. In each case, the results mirror those for the 4 dimensional hypercube web application. The MTC-based crawl outperforms the depth-first and breath-first crawls by a significant margin with the disparity in performance increasing as the number of states increase.

Non-hypercube Web Application #1

The strategy is tested using a non-hypercube web application. As Figure 37 and Figure 38 show, the MTC-based strategy allows finding all states in fewer transitions and resets (9.8 and 2.4 respectively) than breadth-first (20.1 transitions and 9.7 resets) and depth-first (12 transitions and 3.7 resets) crawls.

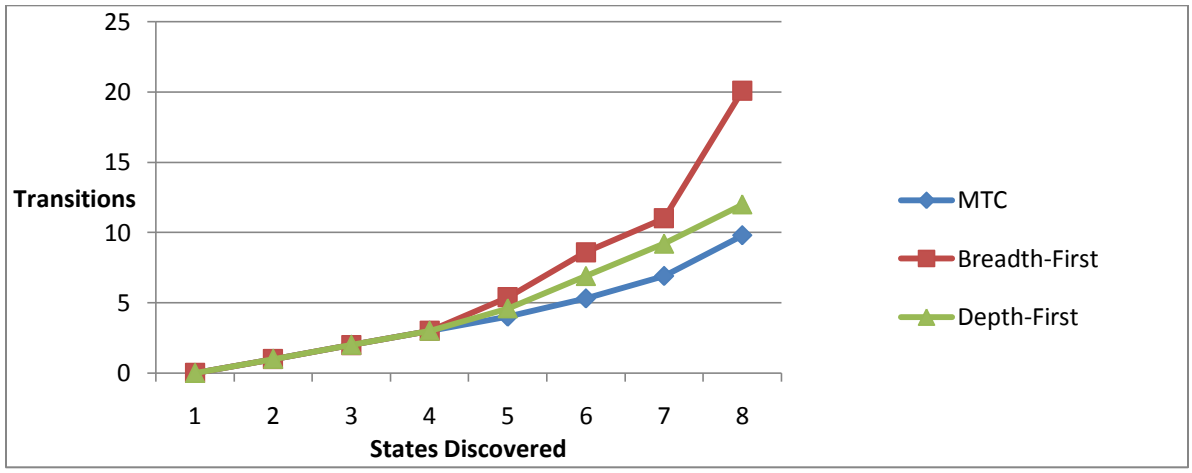

Figure 37: Transitions vs. states discovered (Non-hypercube Web application #1)

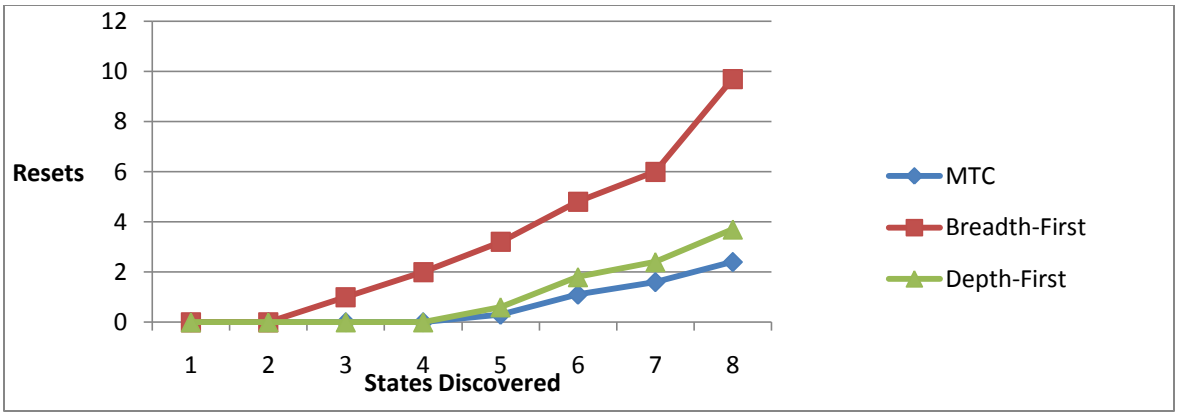

Figure 38: Resets vs. states discovered (Non-hypercube Web application #1)

The MTC-based strategy is also able to discover more transitions in less time. Figure 39 and Figure 40 show this faster rate of transition discovery.

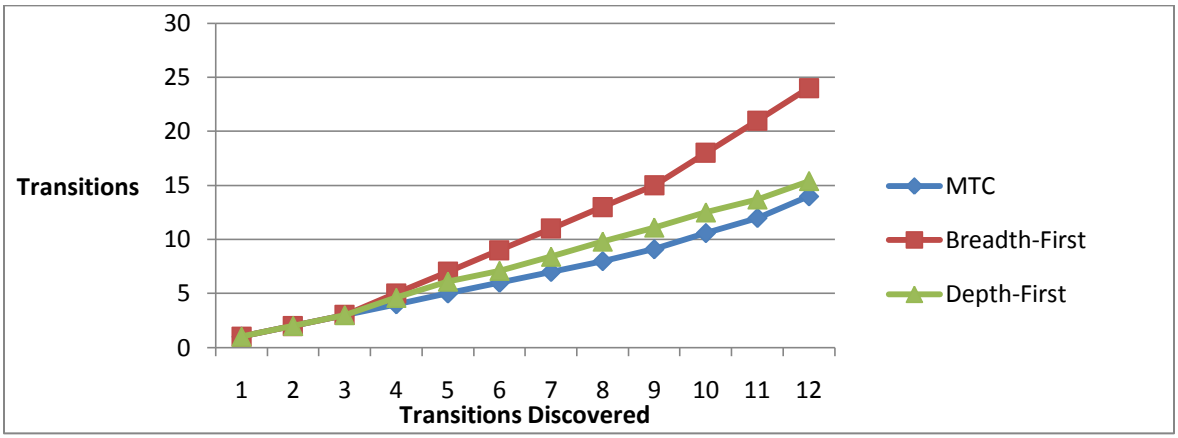

Figure 39: Transitions vs. transitions discovered (Non-hypercube web application #1)

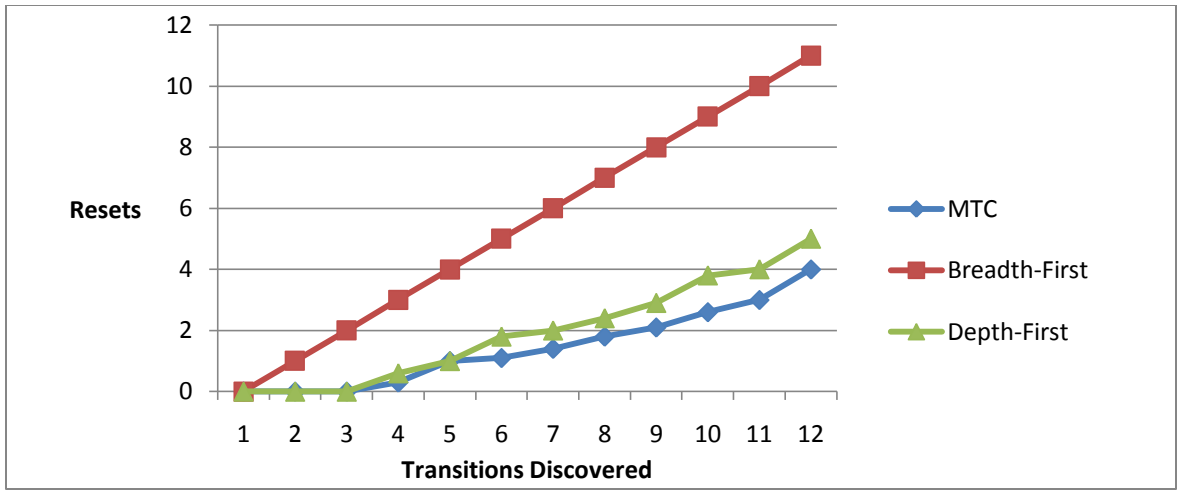

Figure 40: Resets vs. transitions discovered (Non-hypercube web application #1)

Non-hypercube Web Application #2

In this case, the breadth-first crawl again exhibits the worst performance, requiring 35.2 transitions and 13.3 resets to discover all 13 states of the application. It takes the MTC-based crawl 24 transitions and 7.1 resets to accomplish the same task. The depth-first crawl is able to discover all states in roughly the same number of states and transitions, taking 24.6 transitions and 6.7 resets. However, as Figure 41 and Figure 42 show, the MTC-based crawl discovers states at a faster rate for a signification portion of the crawl (this can be observed by looking at the data for states 8 through 12) before slowing down to find the last state in about the same number of transitions as the depth-first crawl.

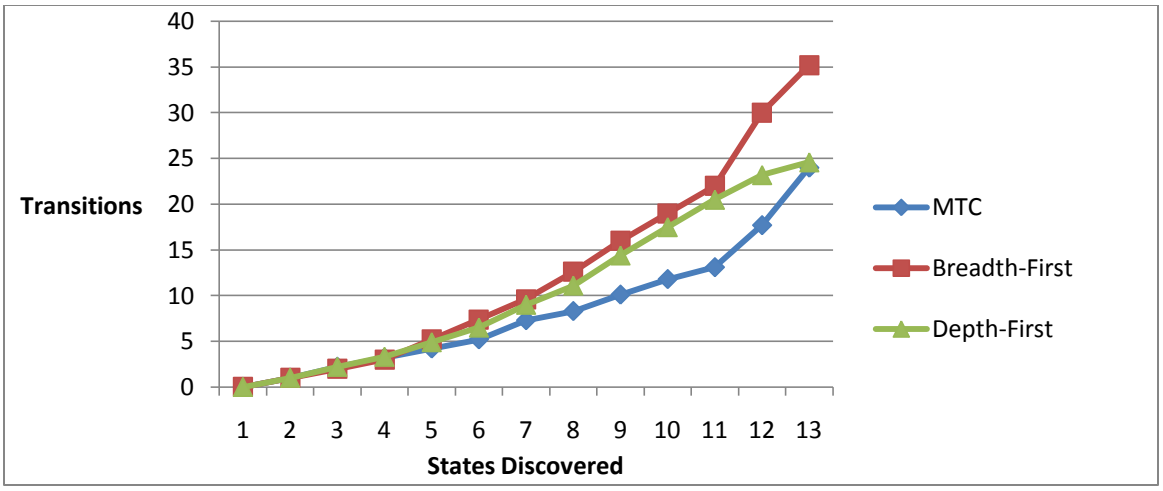

Figure 41: Transitions vs. states discovered (Non-hypercube web application #2)

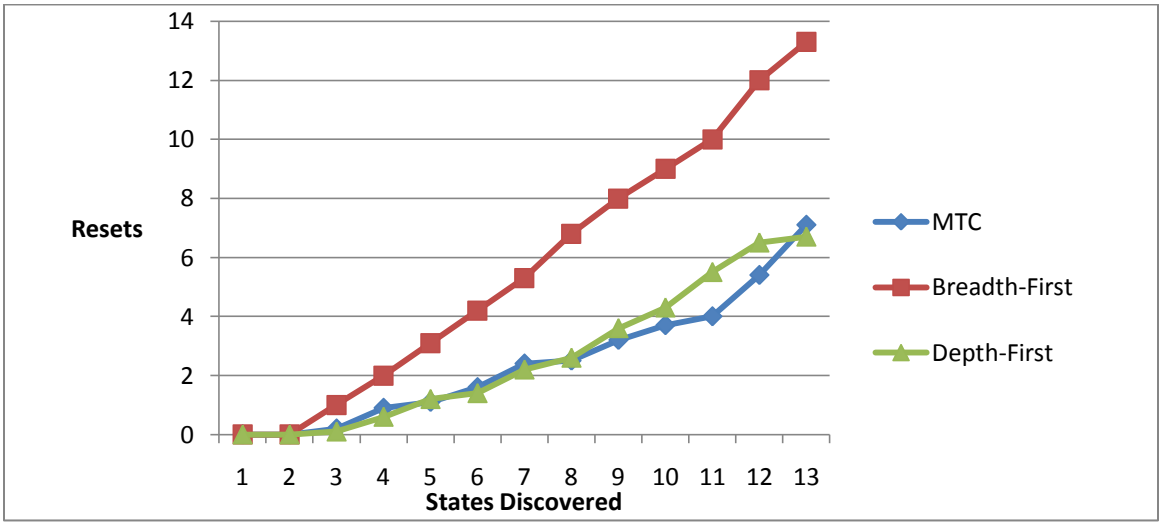

Figure 42: Resets vs. states discovered (Non-hypercube web application #2)

Figure 43 and Figure 44 show that the discovery of transitions follows the same trend seen in Figure 41 and Figure 42. The breadth-first crawl requires the most transitions (38) and resets (14). The MTC-based crawl and the depth-first crawl have a similar rate of transition discovery

in terms of resets. However, in terms of transitions, the MTC-based crawl once again discovers many at a faster rate than the depth-first crawl before that rate decreases resulting in both discovering the 15th (final) transition in a similar number of transitions (26.4 for the MTC-based crawl and 25.4 for the depth-first crawl).

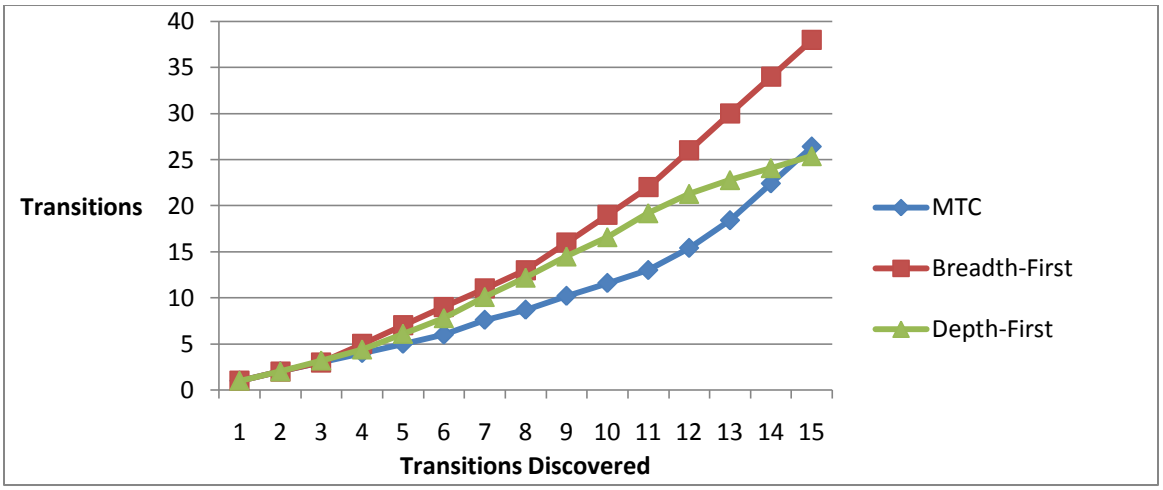

Figure 43: Transitions vs. transitions discovered (Non-hypercube web application #2)

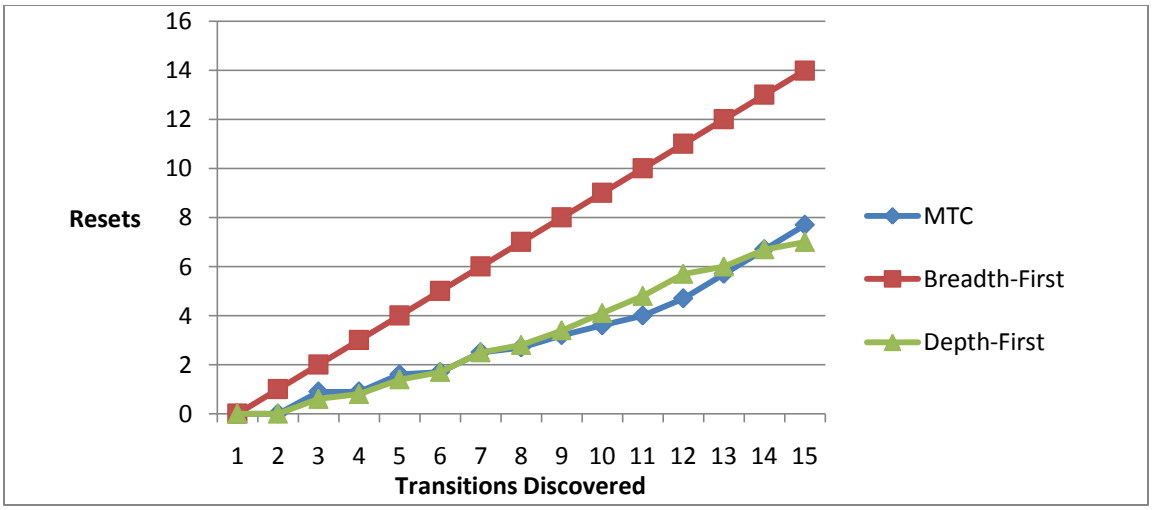

Figure 44: Resets vs. transitions discovered (Non-hypercube web application #2)

Non-hypercube Web Application #3

The results from testing this application (shown in Figure 45) reveal that the MTC-based crawl discovers states at a slightly faster rate (in terms of transitions required) than the depth-first crawl. The depth-first crawl and breadth-first crawl discover states at the same rate (in terms of transitions required) for most of the crawl before the rate of discovery by the breadth-first state increases.

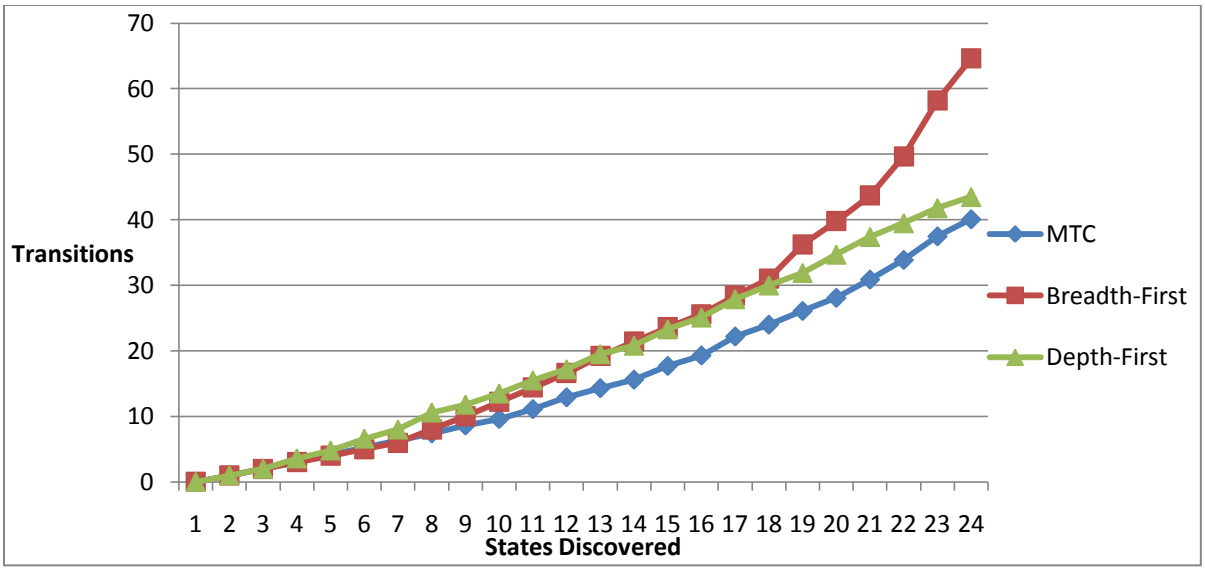

Figure 45: Transitions vs. states discovered (Non-hypercube web application #3)

Figure 46 shows that the number of resets required to discover all states is greatest for the breadth-first crawl (27.9) whereas the MTC-based crawl and the depth-first crawl require a similar number of resets for each state discovered. The total number of resets required to discover all states is 15.2 for the MTC-based crawl and 13.8 for the depth-first crawl.

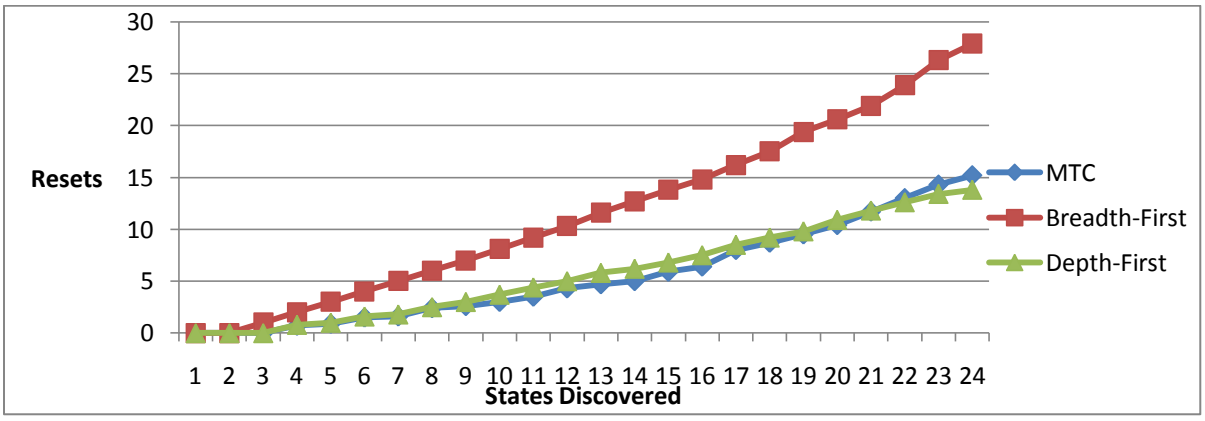

Figure 46: Resets vs. states discovered (Non-hypercube web application #3)

Figure 47 shows similar results to Figure 46. The number of transitions and resets required to discover each transition is almost the same for depth-first and MTC-based crawling. Again, the breadth-first crawl requires significantly more transitions for almost each transition discovered. In total it takes 79 transitions for the breadth-first crawl compared to 52.2 and 48 transitions for the MTC and depth-first crawl respectively. The results shown in Figure 48 indicate that the number of resets required to discover each transition are similar to the number of transitions required.

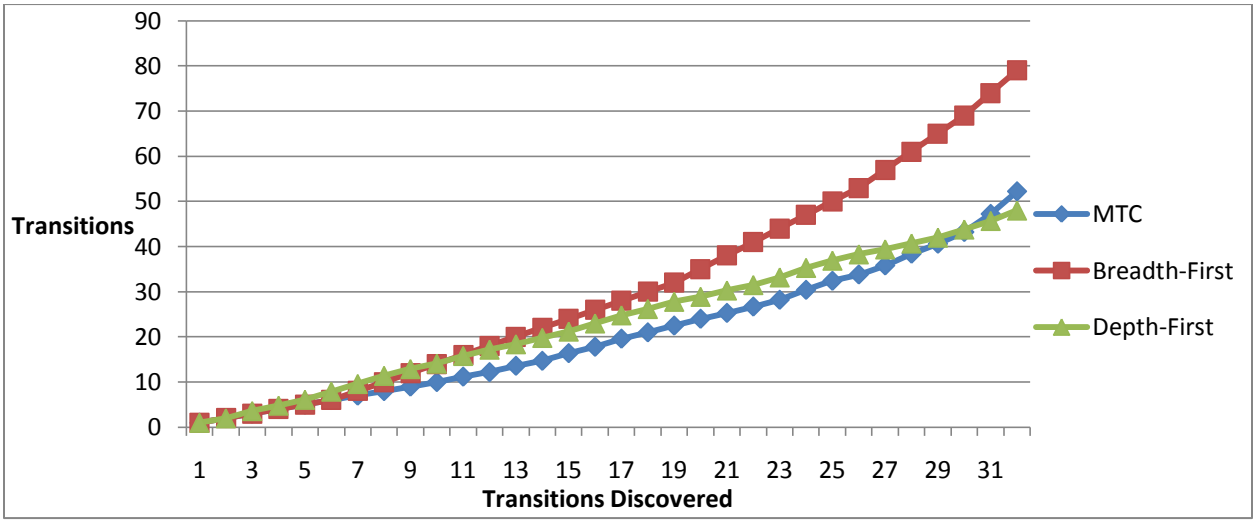

Figure 47: Transitions vs. transitions discovered (Non-hypercube web application #3)

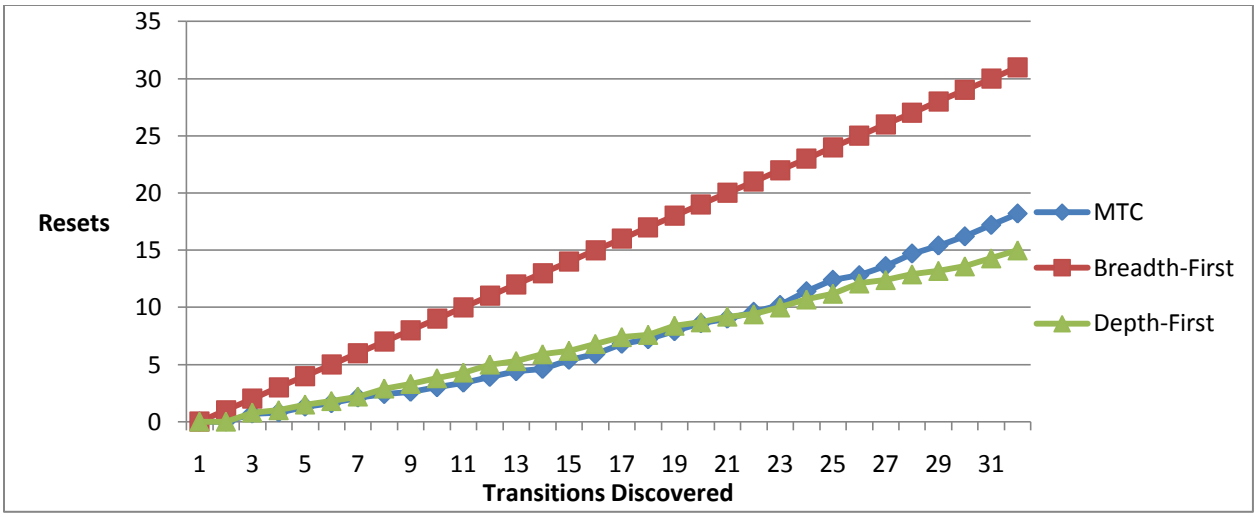

Figure 48: Resets vs. transitions discovered (Non-hypercube web application #3)

Non-hypercube Web Application #4: Previous, Next

For this application, the MTC-based crawl and the depth-first crawl display identical performance. Again, the breadth-first crawl performs significantly worse than the other two. These observations are true across all metrics – the number of transitions and resets required to discover each state, and the number of transitions and resets required to discover each transition. These results are illustrated in Figure 49, Figure 50, Figure 51, and Figure 52 .

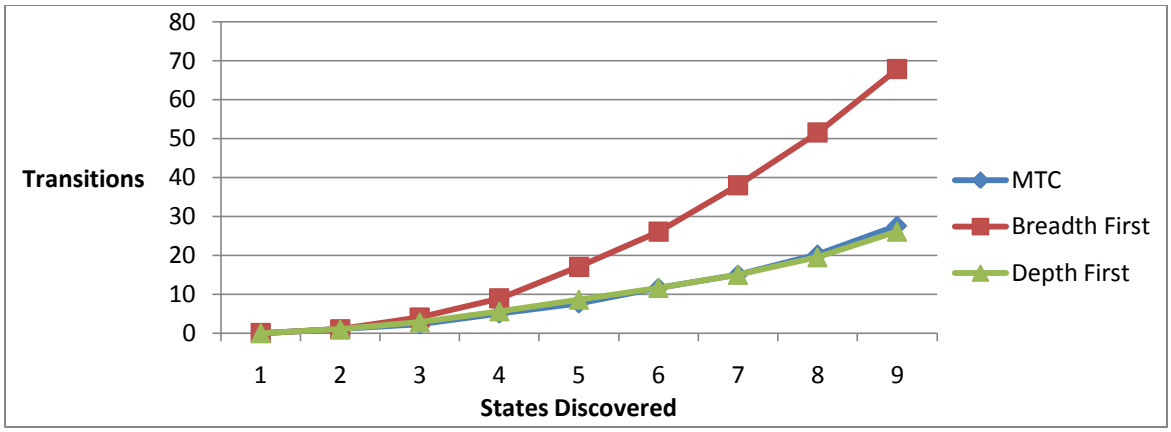

Figure 49: Transitions vs. transitions discovered (Non-hypercube web application #4: Previous, Next)

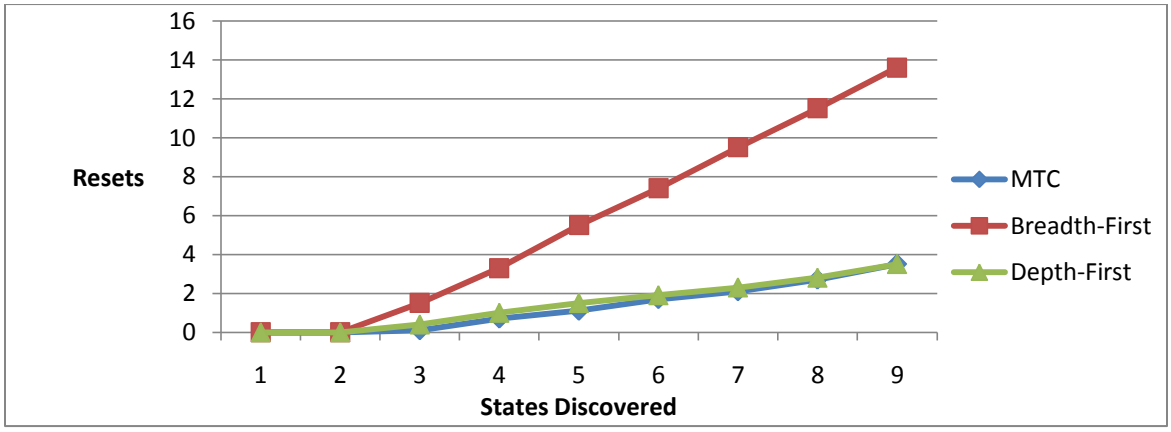

Figure 50: Resets vs. states discovered (Non-hypercube web application #4: Previous, Next)

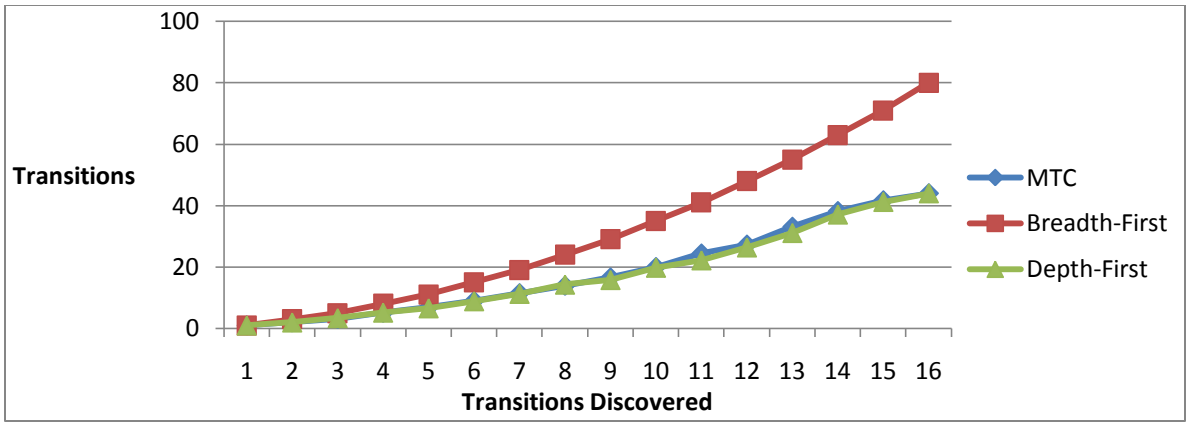

Figure 51: Transitions vs. transitions discovered (Non-hypercube web application #4: Previous, Next)

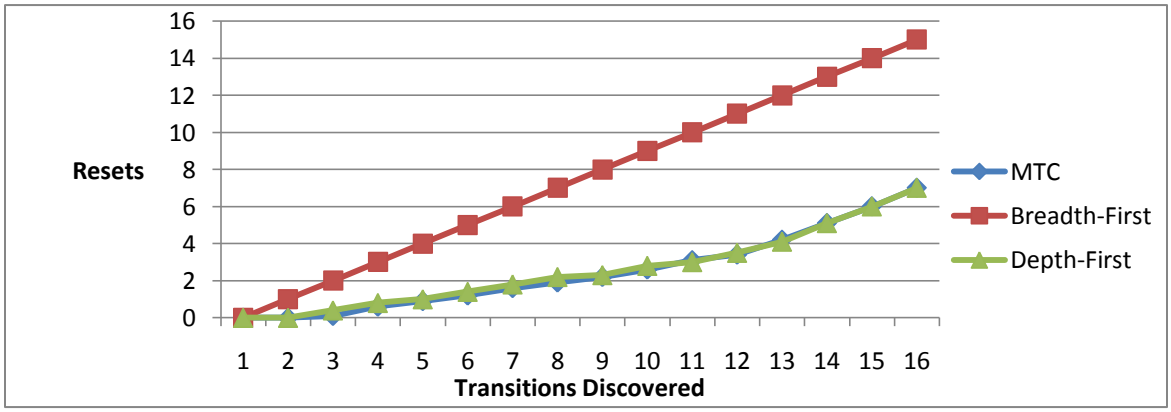

Figure 52: Resets vs. transitions discovered (Non-hypercube web application #4: Previous, Next)

Non-hypercube Web Application #5: AJAX News

In the final test, the events are not randomized. This is because we want to investigate the performance of the strategies in this specific scenario, where the application is only one level deep but is a complete graph. We are particularly interested in how quickly states are discovered in this case. Figure 53 shows that a breadth-first crawl requires the least transitions (in terms of state discovery) in this case, taking 8 transitions to reach all states. The depth-first crawl is the least efficient requiring 147 transitions. The MTC-based crawl requires 37 transitions to complete the same task. This is not as efficient as the breadth-first crawl but much better than the depth-first one.

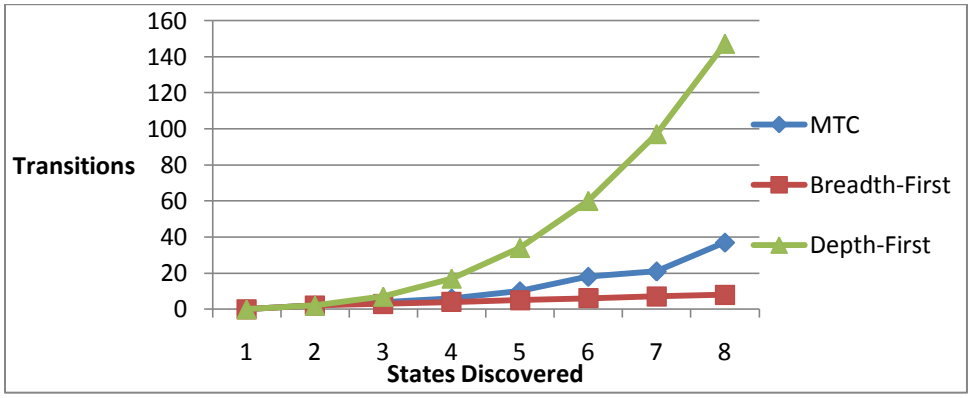

Figure 53: Transitions vs. states discovered (Non-hypercube web application #5: AJAX News)

In terms of resets, Figure 54 shows that the MTC-based crawl requires the least resets (3) in order to find all states. The breadth-first crawl performs well also, requiring 7 resets while the depth-first crawl requires 28 resets.

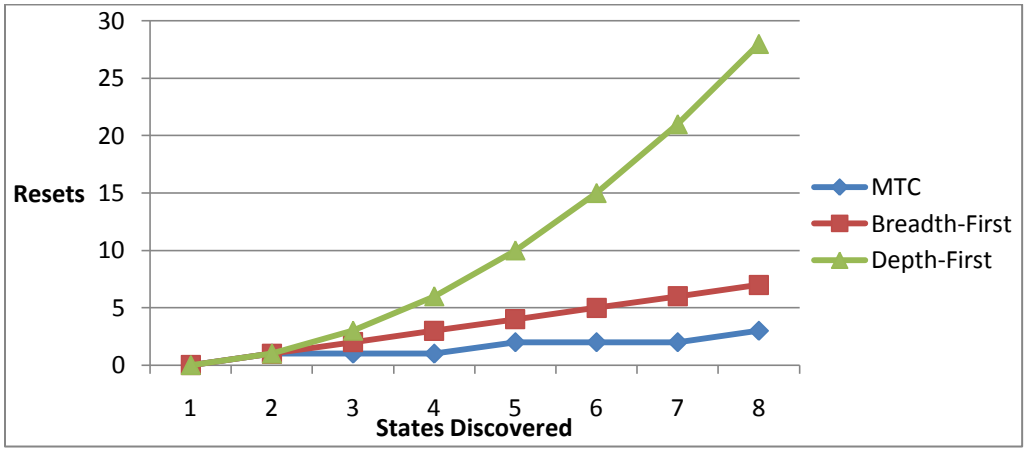

Figure 54: Resets vs. states discovered (Non-hypercube web application #5: AJAX News)

7.3 Evaluation

The results obtained show that while the "load, reload" technique is unable to handle all cases in which there is irrelevant and frequently changing content on the page, it is very useful since it is able to identify this irrelevant content in many cases, negating the need for further analysis of the pages in order to determine state equivalence.

The results also show that the crawling strategy is able to crawl and create the correct models for various applications including those that follow a hypercube structure and those that do not. However, at present all results are based on test AJAX applications. This is due to the current limitations of both the strategy and the prototype tool. With regard to the strategy, it is very difficult to find "real" applications which have 16 or less events enabled in all states. Additionally, the portion of the prototype tool which is responsible for imitating the browser is

inconsistent in its ability to process pages from many web applications thus preventing testing with “real” web applications.

In terms of performance, the strategy has proven to be very efficient for crawling hypercube web applications. In addition, it also exhibited favourable performance when compared to depth-first and breadth-first crawling strategies for non-hypercube web applications. In some tests, the MTC-based strategy was the most efficient. In others the MTC-based strategy and the depth-first strategy had comparable results whereas the breadth-first crawl was least efficient. Even in these cases, however, the MTC-based strategy often had a faster rate of state and transition discovery for a significant portion of the crawl when compared with depth-first crawling. This is very important because as discussed in Section 3.9, there may not always be enough time to cover all states in the application. Therefore, it is desirable to discover as many as possible in the least number of transitions and resets possible. The MTC-based strategy shows significant potential in this regard.

Finally, the last test web application (AJAX News) illustrates another advantage of the MTC-based strategy. While the breadth-first crawl is most efficient in this case (in terms of transitions), the MTC-based strategy performs well especially compared with the depth-first crawl. This is because it also prioritizes covering the breadth of the application as quickly as possible. Therefore, it is less likely to spend significant time in a section of the application where there are less states to be discovered.

Overall, the results show that the MTC-based strategy was the most consistent since it performed well in all scenarios. This reflects the characteristics of the strategy. Like a depth-first crawl, it aims to maximize chain length in order to minimize resets. Additionally, as discussed above, it shares the strength of a breadth-first crawl in that it aims to cover the breadth of the application as quickly as possible. Also, the results confirm that the order of event execution dictated by the strategy accomplishes the goal of discovering states at a faster rate. Again, these results are based on test web applications so additional testing with "real" applications will be required to determine if these results hold more generally.

8 Conclusion and Future Work

Modern applications bring an increased level of responsiveness to the web. They provide a better experience for millions of users who rely on web applications for information, productivity, and entertainment, among other things. However, such applications present many problems for current crawlers because of the differences in the way that they communicate with the server in order to retrieve additional content. This work addresses the problem of crawling AJAX applications. We lay a foundation for this research and the work to follow by identifying the challenges which will need to be solved. We also make several important contributions towards solutions for these challenges and produce a prototype tool which implements many of these ideas.

8.1 Summary of Contributions

The main contributions of this thesis are the following:

- **A compiled list of challenges that will need to be addressed over time in order to produce a crawling tool that is able to crawl rich internet applications**: After studying the work which has been done in this area of crawling (AJAX-based applications in particular) and having done analysis of the problems which need to be solved in order to produce a tool for crawling rich internet applications, we have compiled a list which represents the known challenges related to achieving this goal.

- **An initial strategy for crawling rich internet applications conforming to the structure of a hypercube:** This strategy enables crawling "hypercube" web applications in a minimum number of paths, transitions, and resets. It combines the use of a Minimum Chain Decomposition algorithm with a Minimum Transition Coverage algorithm in order to produce a series of chains which efficiently cover all states in the hypercube followed by all transitions in the hypercube.
- **A technique for modifying the initial strategy:** We will very rarely encounter web applications which are in the form of a perfect hypercube. Therefore, we need a method of adapting the MTC-based strategy in order to be able to crawl any web application. We detail the different scenarios in which we may arrive at a state which contradicts the initial model. We provide an algorithm to adapt the strategy based on the deviation(s) that are encountered.
- **A method of determining whether to execute events or follow URLs when crawling web applications. Also, a method of determining *which* events to execute:** Once multiple URLs have been found or at least one deviation has occurred, there is a need to determine the next URL to follow or hypercube group, hypercube, and chain to crawl. We provide one way of doing so.
- **A complete strategy for crawling web applications:** We provide a complete strategy which allows for crawling web applications which consist of both asynchronous and synchronous requests to the server.
- **The identification of a class of states in AJAX-based applications that we call intermediate states:** We define intermediate states, which may occur between the time an AJAX call is made and the time that the resulting callback method is

executed. We discuss the importance of the intermediate state in building a complete model and its potential security implications.

- **A description of some factors that should be taken into account when determining state equivalence:** We point out that the criteria for evaluating equivalence when the goal is reaching as many states as possible is different from the criteria from evaluating equivalence based on the application of the crawl. We also show that the criteria for determining state equivalence based on the application of the crawl differs depending on the purpose of crawling.
- **A technique for automatically excluding the irrelevant portions of a DOM when computing state equivalence:** We provide a technique which identifies and ignores changing portions of a page which are irrelevant to state equivalence.
- **An initial prototype crawler:** We produced tool that utilizes the event-based crawling strategy as well as "load, reload". It is also able to produce a graph of the model that is uncovered.

8.2 Future Work

Though this work shows promise, we are still in an early stage of development. Given that the overall goal of the research is to improve the crawling of rich internet applications, there are several areas that would benefit from future work.

We have discussed some of the limitations of the current version of the event-based crawling strategy. This includes the current requirement that the strategy generates all chains for a given hypercube before actually doing the crawl. This is memory-intensive and results in some significant limitations. Given that the number of states of a hypercube increases exponentially as the number of dimensions increase, having to generate all chains at the beginning means that we are limited in the number of events that are encountered. For example, the tool is currently only able to generate a strategy for a hypercube of a maximum of 16 dimensions. In order to overcome this limitation, we are working on a version that is capable of generating the strategy on a chain by chain basis. This would be a big advantage because for example, when we encounter a base state with 20 enabled events, instead of requiring the time and memory to generate 1,847,560 chains up front, we want to generate the strategy as we crawl the application. In the end this will allow us to do more crawling earlier giving more immediate results in terms of building the model. In addition, going chain by chain means that there will be no need to replace multiple chains at a time because instead of building many chains, detecting a deviation and having to dispose of many chains and create replacements, we will be able to utilize any feedback from the current chain going forward as we build new ones. Furthermore, even though the base state may have 20 enabled events, the web application may only consist of (for example) 21 states. In the current implement, the tool is unable to crawl such a small site because it fails to move beyond the strategy generation step. The described new version would solve this problem.

It would also be useful to investigate the extent to which existing web applications follow a hypercube structure. This could give some additional insight into just how effective the strategy may be, on average, when used to crawl a larger set of applications.

Another area of limitation of the prototype crawling tool is in the determination of state equivalence. Thus far, we have done some initial work in this area. This includes the identification of a need for both a crawling-based equivalence function and an application-based equivalence function, where the latter would vary based on the application of the crawl. We have also presented the "load, reload" (see Section 4.2) technique which helps to automatically identify and ignore irrelevant and frequently changing portions of the DOM when determining equivalence. However, there is much work to be done on this complex topic, particularly in modern applications where the URL is no longer extremely valuable in uniquely identifying a state. Progress in the area will also make it possible to overcome some of the other challenges, such as state explosion and infinite runs.

Additionally, while the equivalence function should limit the number of states by grouping them into subsets, there may still be a large number of states to crawl and process. However, these states may not be equally important in terms of the purpose of the crawl. There could be a way to determine which states are more important and to try to reach these states first. This would help ensure that the most important states are discovered when time is limited and the crawl cannot be completed. Also, states with minimal importance may not need to be visited.

Another item that could be covered in future work is the implementation of fine-grained control of events. As we discussed, this capability is important for being able to generate a truly complete model since it will allow intermediate states to be captured as well. Additionally, there are other challenges described in Chapter 3 which need to be addressed. These include data input values (Section 3.7) and server states (Section 3.8).

Finally, AJAX-based web applications have been the focus for the initial prototype crawling tool and support for additional technologies should be ensured going forward.

References

[1] J. Garrett, "Ajax: A New Approach to Web Applications," Adaptive path, February 2005, http://www.adaptivepath.com/publications/essays/archives/000385.php.

[2] Adobe Flash. http://www.adobe.com/products/flashplayer/.

[3] Microsoft Silverlight. http://www.microsoft.com/silverlight/.

[4] Adobe Flex. http://www.adobe.com/products/flex/.

[5] K. Benjamin, G. v. Bochmann, G. V. Jourdan, and I. V. Onut, "Some Modeling Challenges When Testing Rich Internet Applications for Security," In 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, Paris, 2010, pp.403-409.

[6] URI. W3C: World Wide Web Consortium, Available at http://www.w3.org/Addressing/.

[7] HTML. W3C: World Wide Web Consortium, Available at http://www.w3.org/MarkUp/.

[8] JavaScript. W3C: World Wide Web Consortium, Available at http://www.w3.org/TR/REC-html40/interact/scripts.html.

[9] DOM. W3C: World Wide Web Consortium, Available at http://www.w3.org/DOM/.

[10] IBM. http://www.ibm.com/.

[11] Rational AppScan. http://www.ibm.com/software/awdtools/appscan/.

[12] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," Seventh International World-Wide Web Conference (WWW 1998), April 14-18, 1998, Brisbane, Australia. Google.

[13] Google Search. http://www.google.com

[14] Yahoo! Search. http://www.yahoo.com

[15] Bing. http://www.bing.com

[16] A. Z. Broder , M. Najork , J. L. Wiener, "Efficient URL Caching for World Wide Web Crawling," Proc. of the 12th International Conference on World Wide Web, May 20-24, 2003, Budapest, Hungary .

[17] ETH Zurich, Eidenossische Technische Hochschule Zurich. http://www.ethz.ch/index_EN.

[18] R. Matter, "AJAX Crawl: Making AJAX Applications Searchable," Master Thesis, ETH, Zurich, 2008. Doi: 10.3929/etz-a-005665330.

[19] G. Frey, "Indexing AJAX Web Applications," Master Thesis, ETH Zurich, 2007.

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

[21] CrawlJax. http://crawljax.com/.

[22] A. Mesbah and A. v. Deursen, "Exposing the Hidden Web Induced by AJAX," TUD-SERG Technical Report Series, TUD-SERG-2008-001. 2008.

[23] D, Roest, A. Mesbah, A. v. Deursen, "Regression Testing Ajax Applications: Coping with Dynamism," ICST, 2010 Third International Conference on Software Testing, Verification and Validation, 2010, pp.127-136.

[24] C, Bezemer, A. Mesbah, and A. v. Deursen, "Automated Security Testing of Web Widget Interactions," Foundations of Software Engineering Symposium (FSE), ACM, 2009, pp. 81–90.

[25] A. Mesbah, E. Bozdag, and A. v. Deursen, "Crawling AJAX by Inferring User Inferface State Changes," Proc. 8th Int. Conf. Web Engineering, ICW08 (2008), pp. 122-134.

[26] A. Marchetto, P.Tonella, and F. Ricca, "State-based Testing of AJAX Web Applications," Proc. 1st IEEE Intl. Conf. on Software Testing Verification and Validation (ICST '08), IEEE Computer Society, 2008.

[27] Z. Bar-Yossef, I. Keidar, and U. Schonfeld, "Do not Crawl in the Dust: Different URLs with Similar Text," In WWW, 2007.

[28] G. S. Manku, A. Jain, and A. D. Sarma, "Detecting Near-duplicates for Web Crawling," Proc. 16th WWW, Banff, Alberta, Canada, May 2007, pp. 141-150.

[29] Cobra: Java HTML Renderer and Parser. http://lobobrowser.org/cobra.jsp.

[30] Rhino: JavaScript for Java. http://www.mozilla.org/rhino/.

[31] Mozilla XUL Runner. https://developer.mozilla.org/en/XULRunner

[32] WebClient. http://www.mozilla.org/projects/blackwood/webclient/.

[33] S. Shah, "Crawling AJAX-driven Web 2.0 Applications," http://www.infosecwriters.com/text_resources/pdf/Crawling_AJAX_SShah.pdf.

[34] Watir: Web Application Testing in Ruby. http://watir.com/.

[35] Microsoft Internet Explorer. http://www.microsoft.com/windows/internet-explorer/default.aspx.

[36] rbNarcissus. http://code.google.com/p/rbnarcissus/.

[37] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-Based Testing in Practice," Proc. ICSE'99, May 1999, Los Angeles, California, pp. 285-294.

[38] M. Fantinato and M. Jino, “Applying Extended Finite State Machines in Software Testing of Interactive Systems”, Lecture Notes in Computer Science 2844 (2003), pp. 34–45.

[39] R. Hierons, “Canonical Finite State Machines for Distributed Systems,” Theoretical Computer Science, vol. 411 (2010), pp. 566–580.

[40] C. Bourhfir, R. Dssouli, and E. M. Aboulhamid, “Automatic Test Generation for EFSM-Based Systems,” Techical Report IRO 1043, University of Montreal. Available at: www.Umontreal.Ca/Labs/Teleinfo/Publistindex.html (1996).

[41] K. Derderian, R. Hierons, M. Harman, and Q. Guo, “Estimating the Feasibility of Transition Paths in Extended Finite State Machines,” 2009, http://bura.brunel.ac.uk/bitstream/2438/4077/1/Fulltext.pdf.

[42] D. Lee and M. Yannakakis, “Principles and Methods of Testing Finite State Machines - a Survey,” Proc. IEEE, vol. 84, Lecture Notes in Computer Science, 1996, pp. 1090-1123.

[43] E. F. Moore, ‘‘Gedanken-experiments on Sequential Machines,’’ Automata Studies, Annals of Mathematics Studies, Princeton University Press, no. 34, pp. 129-153, 1956.

[44] D. Lee and K. Sabnani, ‘‘Reverse-engineering of Communication Protocols,’’ Proc. ICNP, pp. 208-216, 1993.

[45] I. Anderson, “Combinatorics of Finite Sets,” Oxford Univ. Press, London, 1987.

[46] R. P. Dilworth, "A Decomposition Theorem for Partially Ordered Sets", Annals of Mathematics, vol. 51, 1951, pp. 161–166.

[47] N. G. de Bruijn, C. Tengbergen, and D. Kruyswijk, “On the Set of Divisors of a Number,” Niew Arch. Wisk 23 (1951), 191-194.

[48] T. Hsu, M. Logan, S. Shahriari and C. Towse, “Partitioning the Boolean Lattice into Chains of Large Minimum Size”, Journal of Combinatorial Theory, Vol. 97(1), January 2002, pp. 62-84.

[49] Eclipse. http://www.eclipse.org/.

[50] HtmlUnit. http://htmlunit.sourceforge.net/.

[51] XmlUnit. http://xmlunit.sourceforge.net/.

[52] Jung: Java Universal Network/Graph Framework. http://jung.sourceforge.net/.

[53] AJAX News Application. http://www.giannyfrey.com/ajax/news.html.

[54] P. Boldi, B. Codenotti B, M. Santini, and S. Vigna, “Ubicrawler: A Scalable Fully Distributed Web Crawler,” Proc. Aus Web02. The 8th Australian World Wide Web Conference, 2002.

[55] A. Mesbah, “Analysis and Testing of AJAX-based Single-page Web Applications,” PhD Thesis, Delft University of Technology, The Netherlands. 2009.

[56] Algorithms and Theory of Computation Handbook, CRC Press LLC, 1999, "Levenshtein distance", in Dictionary of Algorithms and Data Structures, Paul E. Black, ed., U.S. National Institute of Standards and Technology, 14 August 2008, Available at http://www.itl.nist.gov/div897/sqg/dads/HTML/Levenshtein.html.

Appendix A: Web Applications for Testing "Load, Reload"

30 Test Web Applications For Testing "Load, Reload" (Section 7.1):

[1] http://www.netflix.com

[2] http://www.facebook.com

[3] http://www.wachovia.com

[4] http://www.youtube.com

[5] http://www.logicbuy.com

[6] http://www.wikipedia.org

[7] http://www.amazon.com

[8] http://www.ebay.com

[9] http://www.live.com

[10] http://www.engadget.com

[11] http://www.craigslist.org

[12] http://www.msn.com

[13] http://www.apple.com

[14] http://www.bing.com

[15] http://www.google.com

[16] http://www.foursquare.com

[17] http://www.vark.com

[18] http://www.ikea.com

[19] http://www.www.un.org

[20] http://www.gmail.com

[21] http://www.godaddy.com

[22] http://www.bananarepublic.com

[23] http://www.onelook.com

[24] http://www.bankofamerica.com

[25] http://www.kayak.com

[26] http://www.kbb.com

[27] http://ssrg.site.uottawa.ca

[28] http://www.reuters.com

[29] http://www.newegg.com

[30] http://www.rapidshare.com

Appendix B: Crawling Strategy Comparisons

4 Dimensional Hypercube Web Application

States	Transitions per state discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	1	1	1
3	2	2	2
4	3	3	3
5	4	4	4
6	5	7	6
7	6	10	8
8	7	12.8	10
9	9	17	12.4
10	10	22	15
11	11	23.4	20
12	12	25.6	31
13	13	30.8	34
14	14	37	38.8
15	17	38.8	50.8
16	20	47	68

Table 2: Transitions vs. states discovered (4 dimensional hypercube web application)

States	Resets per state discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	0	0	0
3	0	0	1
4	0	0	2
5	0	0	3
6	1	1	4
7	1	2	5
8	1	2.6	6
9	2	4	7.2
10	2	6	8.5
11	2	6.2	11
12	3	6.6	16
13	3	8.4	17
14	3	11	18.6
15	4	11.4	22.6
16	5	15	28

Table 3: Resets vs. states discovered (4 dimensional hypercube web application)

States	Transitions per transition discovered		
	MTC	**Depth-First**	**Breadth-First**
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	7	6
6	6	8	8
7	7	10	10
8	8	11	12
9	9	13.2	14
10	10	15	16
11	11	17	18
12	12	18	20
13	13	21	22
14	14	22	24
15	15	23	26
16	17	24.2	28
17	18	26	31
18	20	27.8	34
19	21	30	37
20	23	31.4	40
21	24	33.6	43
22	26	36	46
23	27	37	49
24	29	38	52
25	30	39.4	55
26	32	41.6	58
27	33	44	61
28	35	46	64
29	36	47	68
30	38	48	72
31	39	50	76
32	40	52	80

Table 4: Transitions vs. transitions discovered (4 dimensional hypercube web application)

States	Resets per transition discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	0	0	1
3	0	0	2
4	0	0	3
5	1	1	4
6	1	1	5
7	1	2	6
8	1	2	7
9	2	2.6	8
10	2	3	9
11	2	4	10
12	3	4	11
13	3	5	12
14	3	6	13
15	3	6	14
16	4	6.2	15
17	4	6.6	16
18	5	7	17
19	5	8	18
20	6	8.4	19
21	6	9	20
22	7	10	21
23	7	11	22
24	8	11	23
25	8	11.4	24
26	9	12	25
27	9	13	26
28	10	14	27
29	10	15	28
30	11	15	29
31	11	16	30
32	11	17	31

Table 5: Resets vs. transitions discovered (4 dimensional hypercube web application)

Non-hypercube Web Application #1

States	Transitions per state discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	1	1	1
3	2	2	2
4	3	3	3
5	4	4.6	5.4
6	5.3	6.9	8.6
7	6.9	9.2	11
8	9.8	12	20.1

Table 6: Transitions vs. states discovered (Non-hypercube web application #1)

States	Resets per state discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	0	0	0
3	0	0	1
4	0	0	2
5	0.3	0.6	3.2
6	1.1	1.8	4.8
7	1.6	2.4	6
8	2.4	3.7	9.7

Table 7: Resets vs. states discovered (4 dimensional hypercube web application)

States	Transitions per transition discovered		
	MTC	**Depth-First**	**Breadth-First**
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4.6	5
5	5	6.1	7
6	6	7.1	9
7	7	8.4	11
8	8	9.8	13
9	9.1	11.1	15
10	10.6	12.5	18
11	12	13.7	21
12	14	15.4	24

Table 8: Transitions vs. transitions discovered (4 dimensional hypercube web application)

States	Resets per transition discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	0	0	1
3	0	0	2
4	0.3	0.6	3
5	1	1	4
6	1.1	1.8	5
7	1.4	2	6
8	1.8	2.4	7
9	2.1	2.9	8
10	2.6	3.8	9
11	3	4	10
12	4	5	11

Table 9: Resets vs. transitions discovered (4 dimensional hypercube web application)

Non-hypercube Web Application #2

States	Transitions per state discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	1	1	1
3	2.2	2.2	2
4	3.2	3.3	3
5	4.2	4.9	5.2
6	5.2	6.5	7.4
7	7.3	9	9.6
8	8.3	11.1	12.6
9	10.1	14.4	16
10	11.8	17.5	19
11	13.1	20.5	22
12	17.7	23.2	30
13	24	24.6	35.2

Table 10: Transitions vs. states discovered (Non-hypercube web application #2)

States	Resets per state discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	0	0	0
3	0.2	0.1	1
4	0.9	0.6	2
5	1.1	1.2	3.1
6	1.6	1.4	4.2
7	2.4	2.2	5.3
8	2.5	2.6	6.8
9	3.2	3.6	8
10	3.7	4.3	9
11	4	5.5	10
12	5.4	6.5	12
13	7.1	6.7	13.3

Table 11: Resets vs. states discovered (Non-hypercube web application #2)

States	Transitions per transition discovered		
	MTC	**Depth-First**	**Breadth-First**
1	1	1	1
2	2	2	2
3	3	3.2	3
4	4	4.4	5
5	5	6.1	7
6	6	7.8	9
7	7.6	10.1	11
8	8.7	12.2	13
9	10.2	14.5	16
10	11.6	16.6	19
11	13	19.2	22
12	15.4	21.3	26
13	18.4	22.8	30
14	22.4	24.1	34
15	26.4	25.4	38

Table 12: Transitions vs. transitions discovered (Non-hypercube web application #2)

States	Resets per transition discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	0	0	1
3	0.9	0.6	2
4	0.9	0.8	3
5	1.6	1.4	4
6	1.7	1.7	5
7	2.5	2.5	6
8	2.7	2.8	7
9	3.2	3.4	8
10	3.6	4.1	9
11	4	4.8	10
12	4.7	5.7	11
13	5.7	6	12
14	6.7	6.7	13
15	7.7	7	14

Table 13: Resets vs. transitions discovered (Non-hypercube web application #2)

Non-hypercube Web Application #3

States	Transitions per state discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	1	1	1
3	2	2	2
4	3	3.6	3
5	4.1	4.8	4
6	5.3	6.6	5
7	6.3	8	6
8	7.4	10.6	8
9	8.6	11.8	10
10	9.6	13.5	12.2
11	11.1	15.5	14.4
12	12.9	17.2	16.6
13	14.3	19.5	19.2
14	15.6	20.8	21.4
15	17.7	23.3	23.6
16	19.3	25.1	25.6
17	22.2	27.9	28.4
18	24	30	31
19	26.1	31.9	36.2
20	28.1	34.7	39.8
21	30.9	37.4	43.7
22	33.9	39.5	49.7
23	37.5	41.8	58.2
24	40.1	43.5	64.6

Table 14: Transitions vs. states discovered (Non-hypercube web application #3)

States	Resets per state discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	0	0	0
3	0	0	1
4	0.7	0.8	2
5	0.9	1	3
6	1.5	1.6	4
7	1.6	1.8	5
8	2.4	2.5	6
9	2.6	3	7
10	3	3.7	8.1
11	3.5	4.4	9.2
12	4.3	5	10.3
13	4.7	5.8	11.6
14	5	6.2	12.7
15	5.9	6.8	13.8
16	6.4	7.5	14.8
17	8	8.5	16.2
18	8.7	9.2	17.5
19	9.5	9.8	19.4
20	10.4	10.9	20.6
21	11.7	11.8	21.9
22	13	12.6	23.9
23	14.3	13.4	26.3
24	15.2	13.8	27.9

Table 15: Resets vs. states discovered (Non-hypercube web application #3)

States	Transitions per transition discovered		
	MTC	**Depth-First**	**Breadth-First**
1	1	1	1
2	2	2	2
3	3	3.6	3
4	4	4.8	4
5	5	6.1	5
6	6	7.9	6
7	7	9.6	8
8	8	11.4	10
9	9	12.9	12
10	10	14.1	14
11	11.2	15.8	16
12	12.2	17.2	18
13	13.6	18.4	20
14	14.7	19.8	22
15	16.4	21.2	24
16	17.8	23	26
17	19.6	24.8	28
18	21	26.2	30
19	22.5	27.8	32
20	24	28.9	35
21	25.3	30.3	38
22	26.7	31.5	41
23	28.2	33.2	44
24	30.4	35.3	47
25	32.4	36.9	50
26	33.8	38.3	53
27	35.8	39.4	57
28	38.4	40.7	61
29	40.6	42	65
30	43.2	43.8	69
31	47.2	45.7	74
32	52.2	48	79

Table 16: Transitions vs. transitions discovered (Non-hypercube web application #3)

States	Resets per transition discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	0	0	1
3	0.7	0.8	2
4	0.8	1	3
5	1.3	1.5	4
6	1.6	1.8	5
7	2.1	2.2	6
8	2.4	2.9	7
9	2.6	3.3	8
10	3	3.8	9
11	3.4	4.3	10
12	3.9	5	11
13	4.4	5.3	12
14	4.6	5.9	13
15	5.4	6.2	14
16	5.9	6.8	15
17	6.8	7.4	16
18	7.2	7.6	17
19	7.9	8.4	18
20	8.6	8.7	19
21	9	9.2	20
22	9.6	9.4	21
23	10.2	10	22
24	11.4	10.7	23
25	12.4	11.2	24
26	12.8	12.1	25
27	13.6	12.4	26
28	14.7	12.9	27
29	15.4	13.2	28
30	16.2	13.6	29
31	17.2	14.3	30
32	18.2	15	31

Table 17: Resets vs. transitions discovered (Non-hypercube web application #3)

Non-hypercube Web Application #4: Previous, Next

States	Transitions per state discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	1	1	1
3	2.2	2.8	4
4	5	5.6	8.9
5	7.6	8.6	17
6	11.6	11.6	26
7	15	15	38
8	20.2	19.5	51.5
9	27.6	26.1	67.8

Table 18: Transitions vs. states discovered (Non-hypercube web application #4: Previous, Next)

States	Resets per state discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	0	0	0
3	0.1	0.4	1.5
4	0.7	1	3.3
5	1.1	1.5	5.5
6	1.7	1.9	7.4
7	2.1	2.3	9.5
8	2.7	2.8	11.5
9	3.5	3.5	13.6

Table 19: Resets vs. states discovered (Non-hypercube web application #4: Previous, Next)

States	Transitions per transition discovered		
	MTC	**Depth-First**	**Breadth-First**
1	1	1	1
2	2	2	3
3	3.1	3.4	5
4	5.1	5.2	8
5	6.9	6.6	11
6	9	8.9	15
7	11.5	11.3	19
8	13.9	14.3	24
9	16.6	15.8	29
10	20	19.8	35
11	24.4	22.2	41
12	27.3	26.4	48
13	33.1	31.1	55
14	38.2	37.1	63
15	41.7	41.2	71
16	44	44	80

Table 20: Transitions vs. transitions discovered (Non-hypercube web application #4: Previous, Next)

States	Resets per transition discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	0	0	1
3	0.1	0.4	2
4	0.6	0.8	3
5	0.9	1	4
6	1.2	1.4	5
7	1.6	1.8	6
8	1.9	2.2	7
9	2.2	2.3	8
10	2.6	2.8	9
11	3.1	3	10
12	3.4	3.5	11
13	4.2	4.1	12
14	5.1	5.1	13
15	6	6	14
16	7	7	15

Table 21: Resets vs. transitions discovered (Non-hypercube web application #4: Previous, Next)

Non-hypercube Web Application #5: AJAX NEWS

States	Transitions per state discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	2	2	2
3	4	7	3
4	6	17	4
5	10	34	5
6	18	60	6
7	21	97	7
8	37	147	8

Table 22: Transitions vs. states discovered (Non-hypercube web application #5: AJAX News)

States	Transitions per state discovered		
	MTC	**Depth-First**	**Breadth-First**
1	0	0	0
2	1	1	1
3	1	3	2
4	1	6	3
5	2	10	4
6	2	15	5
7	2	21	6
8	3	28	7

Table 23: Resets vs. states discovered (Non-hypercube web application #5: AJAX News)