TEST CASE GENERATION USING COMBINATORIAL BASED COVERAGE FOR RICH WEB APPLICATIONS

by

Chad Maughan

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Renee Bryce Major Professor	Dr. Daniel Bryce Committee Member
Dr. Daniel Watson Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2012

ABSTRACT

Test Case Generation Using Combinatorial Based Coverage for Rich Web Applications

by

Chad Maughan, Master of Science

Utah State University, 2012

Major Professor: Dr. Renee Bryce
Department: Computer Science

Web applications are increasingly moving business and processing logic from the server to the browser. Traditional, multiple-page request/response applications are quickly being replaced by single-page applications where complex application logic is downloaded on the initial page load and data is then subsequently fetched asynchronously via the browser's native XMLHttpRequest (XHR) object.

These new generation web applications are called Rich Web Applications (RWA). Frameworks such as the Google Web Toolkit (GWT), and JavaScript model-view-controller (MVC) frameworks such as Backbone.js are facilitating this move. With this migration, testing frameworks need to follow the logic by moving analysis and test generation from the server to the client. One problem hindering the movement of testing in this domain is the adoption of semantic URLs. This paper introduces a novel approach to systematically identify variables in semantic URLs and use them as part of the test generation process.

Using a sample RWA seeded with various JavaScript faults, I demonstrate in this thesis, as an empirical study, that combinatorial testing algorithms and reduction strategies also apply to new RWAs.

(36 pages)

PUBLIC ABSTRACT

CHAD M. MAUGHAN

Rich Web Applications (RWA) that are data driven and feature responsive user interfaces are rapidly growing in popularity. Popular sites such as Twitter, Pandora, and Angry Birds (browser version) are all examples of popular RWAs. These RWAs are more complex and are developed differently than many previous web sites. More of the processing power needed to run these applications is performed on the client machine, not the server. Due to this development strategy, testing tools need new techniques to identify web application variables, capture errors, and identify problems. In this thesis, I introduce novel techniques to identify variables in RWA semantic URLs and automatically generate tests for RWAs using a form of testing called combinatorial testing.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

After the introduction of HTML5 in 2011, and its compelling set of new APIs [**?**], Rich Web Applications (RWA) have rapidly gained in popularity. RWAs differ from the more recent Rich Internet Applications (RIA) in that they don't require proprietary browser plug-ins. They also differ from traditional request/response web applications in that they typically use a single HTML page and update the view on the client-side instead of building many HTML pages on the server-side.

The concept of a "rich" in-browser application experience was introduced years earlier with technologies such as Adobe's Flex, Microsoft's Silverlight, and Oracle's JavaFX [**?**]. Each of these technologies uses proprietary browser add-ons to provide a responsive user interaction and are categorized as Rich Internet Applications (RIA). These proprietary add-ons provide additional functionalities to the web that were missing before HTML5, but also decreased its openness [**?**].

Rich Web Applications (RWA) differ from RIAs in that RWAs focus on creating a specific browser independent application without the reliance on any proprietary, client-side browser plug-ins. Indeed, Ian Hickson, one of the W3Cs HTML5 editors, stated "one of our goals is to move the Web away from proprietary technologies" [**?**].

Additionally, there are key differences between traditional web applications and RWAs [**?**]. First, a traditional web application consists of many static HTML pages or scripts (e.g., JSP or PHP) to render different portions of the site [**?**, **?**]. A user of a traditional web application is required to retrieve a full page for each interaction with the application. Conversely, a RWA typically consists of a single HTML page that acts as an entry point. This single page entry point contains complex application logic implemented in bundled JavaScript or a collection of JavaScript script files. This application logic is downloaded

on the first HTML entry point page load and is then run on the browser. A RWA then updates its views by manipulating the Document Object Model (DOM) using HTML composites downloaded with that first page load. It also retrieves data associated with these components by asynchronously making Ajax requests (i.e., XmlHttpRequest or XHR) to the server. The server then responds with data in a JSON or XML format.

Along with this architecture change, RWAs have adopted a new URL structure that relies heavily on sometimes complex URL fragments to maintain history and state within the application. These URLs are sometimes called semantic or clean URLs. These semantic URLs are much easier for the end-user to understand, describe, and infer site application structure. While easier for user understanding, semantic URLs pose a significantly more difficult challenge in systematically identifying variables for subsequent testing.

As RWAs adopt the HTML5 strategy of moving away from proprietary technologies, there is also a movement to follow the HTTP protocol as it was designed— a "generic, stateless protocol" [**?**]. This leads to many RWAs being stateless. A stateless web application has many benefits, including performance, scalability, and simplicity. Not requiring the expensive overhead of maintaining session information on the server allows for more concurrent users with less infrastructure. It also allows for the removal of an additional layer of complexity in the production environment by not requiring the replication of session information across multiple nodes.

A RWA architecture provides many benefits. From a development standpoint, a RWA allows for a clean separation of concerns. Front-end development and design can be done independent of the back-end services. Back-end developers can then focus exclusively on building a robust API. This API can then be made available to both RWAs and other non-web applications, such as native Android or iOS applications. Perhaps the biggest advantage is that a user is never required to load a new page. Due to this, and the resulting lack of latency, a RWA feels more responsive or "rich."

While end users benefit from the migration of complex application logic from the server to the browser, testing has become more difficult. Exceptions and coding irregularities no

longer occur solely on the server in a convenient, controlled, and centralized location. Errors now occur in a distributed fashion in various types and versions of browsers, computers, and environments. As such, a more structured approach to testing needs to move from the server to the client.

The RWA focus on a "rich" user experience (i.e., responsive interfaces and interactive capabilities) [**?**], comes at a price as the "richness" increases the importance and cost of testing. For software testing, much effort has been applied to controlling the cost while still maintaining effective fault discovery. Research into quality and reliability of software development by Wallace and Kuhn has yielded empirical data that "suggests that relatively few parameters are actually involved in triggering failures" [**?**]. Kuhn et al. reviewed 15 years worth of medical device recall data gathered by the U.S. Food and Drug Administration (FDA). Working with 109 cases, they determined that 97% of all reported flaws could be detected by testing all pairs of parameter settings [**?**]. Pairwise testing (i.e., two-way testing) combines all interactions between two parameters. In a separate study, Kuhn et al. also discovered by analyzing the publicly available fault database of the Mozilla Browser, that 76% of faults were found by two-way testing [**?**]. The remaining faults were identified with higher "t-wise" combination of parameters.

Colbourn shows that producing higher "t-wise" variable combinations is NP Hard [**?**], meaning the number of test cases to perform grows exponentially with higher variable interaction. This substantial growth in test cases can make exhaustive or higher strength testing impractical and costly. One strategy to control cost is to reduce the number of tests needed to discover faults. Due to the heavy use of templates (i.e., reusable snippets of layout and logic) in RWAs, they are a prime candidate for reduction. Later, I propose a reduction strategy and provide a comparison on the effectiveness of discovering faults on a sample application.

In addition to generating tests that cover interactions, combinatorial based prioritization may improve testing effectiveness by increasing the rate at which faults are detected. This strategy tests "more important" components early in the testing phase. While outside

the scope of this paper, I do mention some ideas for prioritization that could be explored in future research. Indeed, recent work by Bryce et al. [**?**, **?**, **?**], with two-way inter-window event coverage for event-driven systems, has applicability with RWAs.

Before any combinatorial testing can be performed, a covering array needs to be generated to build the "t-wise" combination of variables. With their single page HTML entry points, and their dynamic, semantic URL fragments, RWAs pose a challenge in identifying variables for the creation of covering arrays. This thesis will also introduce a novel and practical way to identify variables in semantic URL fragments and demonstrate that combinatorial testing, like in other application types, provides benefit in identifying faults in RWAs on a relatively small test suite.

CHAPTER 2

BACKGROUND

This chapter is divided into four sections. First, I provide some background on JavaScript and the characteristics of the language that make it challenging to test. Second, I explain a traditional URL format, describing the relative ease of programmatically discovering variables. Third, I explain the more recent semantic URL format and some of the difficulties introduced with systemic variable identification. Fourth, I introduce and describe an algorithm for statistically analyzing an application's URL structure to identify variables.

2.1 JavaScript Background

Due to its distribution in all browsers, JavaScript is the primary language for Rich Web Applications (RWA). It plays a central role in RWAs by updating styles and modifying application structure markup (i.e., the Document Object Model or DOM). It also is the key data transport mechanism for interacting with the server via it's XmlHttpRequest (XHR) object. As a language, it is dynamic, weakly typed, interpreted, and prototype-based. With it's features such as first-class functions, it supports multiple development paradigms, including: object-oriented, imperative, and functional. JavaScript can either be loaded by the browser as static code or it can be dynamically created at runtime using an "eval( )" function.

Two traits make JavaScript particularly vulnerable to faults. First, as a dynamically typed, interpreted language it does not benefit as much as strongly typed, compiled languages from static code analysis. Lint-like tools exist for JavaScript but they typically focus on syntax. Second, JavaScript is designed to respond to timers and browser events such as clicks, hovering, or key presses. This makes it difficult to test event driven code in a development environment as it requires simulating those events to test.

New JavaScript frameworks such as Backbone.js have helped with the consistency of JavaScript code being developed. They have helped isolate faults to mostly appearance related exceptions. In the next chapter, I describe in detail the types of web application and JavaScript faults in the context of a sample application and how those faults are identified with the developed testing strategies.

2.2 Traditional URL Formats

Rich web applications, along with a migration from server-side to client-side JavaScript based frameworks, have generally adopted a different URL structure from the traditional format defined in RFC 1738 [**?**]. A traditional URL structure defined in the RFC 1738 standard, follows the format:

http://<host>:<port>/<path>?<searchpart>

The <searchpart> of a traditional URL format is composed of a series of key-value pairs that start after the question mark. Keys and values are separated by an equals sign and the key-value pairs are separated by an ampersand. An example of a <searchpart>, more frequently called a query string, following the question mark of a URL is as follows:

key1=value1&key2=value2

Traditional URL structures, as defined in RFC 1738, with their key-value pairs are very easy to programmatically identify and parse. This well structured format allows a testing solution to quickly determine variable names and values for combinatorial testing.

2.3 Semantic URL Formats

Rich web applications have departed from this traditional <searchpart> format in favor of a more user friendly, more permanent format [**?**]. This new format is often referred to as semantic or clean URLs. One goal of semantic URLs is to provide an immediate, human

understandable format of resources. Instead of clearly identifying variables and their values as key-value pairs after a question mark in the <searchpart> section of a URL, rich web applications typically combine their variables and values as part of the <path> portion of the URL. While easier for a human to understand, this structure makes it difficult to programmatically discover variables for combinatorial testing.

As an example of a semantic URL, imagine a rich web application that provides census information on states, counties, and cities over a certain population of the United States. A semantic URL for this application might look as follows:

http://example.com/#/state/utah/county/cache/city/logan

This example is a well structured semantic URL. Each variable is preceded by a descriptive key (e.g., "state" precedes "ut" and "county" precedes "cache"). Unfortunately, no standards exist for semantic URL structures, and many rich web applications do not always find it practical to follow this industry best practice of associated keys and values for variables. This application could have just as easily been developed with a URL structure:

http://example.com/#/utah/cache/logan

Due to the variety of formats used in semantic URLs and the absence of an industry standard, identifying variables for combinatorial testing can be difficult. In the next section I propose an algorithm for variable identification regardless of the example structures shown above.

2.4 Variable Detection Algorithm for Semantic URLs

In order to address the complexities of identifying variables in semantic URLs, I propose the following algorithm that processes all URLs from a rich web application and identifies the location of variables in a URL structure.

The algorithm processes each URL of the rich web application and then subsequently processes each portion of the URL <path>. It then combines them into a common hierarchical graph to allow analysis of the application structure. The key to identifying the location of variables is by calculating branching complexity as each URL is processed. This branching complexity is measured by creating a graph node at each URL <path> portion (e.g., each part of the URL path separated by a "/" character). As each node is created, the Strahler number is calculated by looking at all the previous child nodes [**?**]. Identifying the degree of branching at each level of the URL path allows for analysis of the branching complexity for each abstract portion of the URLS of the application.

The algorithm is listed in Figure 2.1 followed in the next section by a detailed description.

2.5 Algorithm Description

For brevity, the algorithm, as listed, assumes the possession of a list every possible URL available in the application. The sample rich web application discussed in this section has approximately 200,000 different semantic URLs. A better implementation of the algorithm would be to crawl the site in a depth-first, non-deterministic fashion recording and processing each URL (i.e., applying this algorithm in-line as the site is crawled).

The algorithm proceeds to lob off each outer part of the URL path until the position variable listed on line 103 is less than the fragment identifier character position in the URL.

Starting with the example URL listed previously, the algorithm values are processed in the following list.

1. http://example.com/#/state/ut/county/cache/city/logan.
2. http://example.com/#/state/ut/county/cache/city
3. http://example.com/#/state/ut/county/cache
4. http://example.com/#/state/ut/county
5. http://example.com/#/state/ut/

```
100 foreach(url in urls) {
101     Node previousNode = retrieveOrCreateNode(url);
102     int position = url.length();
103     String shortenedUrl;
104     while(position > fragmentIdentifier) {
105         position = url.lastIndexOf("/");
106         if(position > 0) {
107             shortenedUrl = url.substring(0, position);
108             Node node = retrieveOrCreateNode(shortenedUrl);
109             int branchingComplexity = calculateBranchingComplexity(node);
110             node.setProperty(BRANCHING_COMPLEXITY, branchingComplexity);
111             previousNode.createRelationshipTo(node,
112                     RelationshipTypes.CHILD_OF);
113             previousNode = node;
114         }
115         else {
116             break;
117         }
118     }
119     Node last = retrieveOrCreateNode(shortenedUrl);
120 }
```

Figure 2.1: Algorithm for variable identification of semantic URLs

6. http://example.com/#/state

7. http://example.com/#

As we're focused on the <path> portion of the URL, the algorithm stops processing when the position index of the last instance of the path separator "/" is less than the position index of the fragment identifier (the "#" character). Continuing on line 108, the algorithm then either retrieves (if it already exists) or creates (if the remaining URL has not been processed) a new node that represents that portion of the URL. The algorithm works its way backwards from full URL to just the host to allow for the easy calculation of the Strahler number used to identify branching complexity in line 109. On line 111 the

algorithm creates the edge between the current and the previous node. It then assigns the current node to the previous node so it can continue processing if needed.

An additional benefit of having the application structure in a common graph is it makes it easy to visually identify branching complexity as in Figure 2.2. Dark areas represent extensive branching that is associated with variables in semantic URLs.

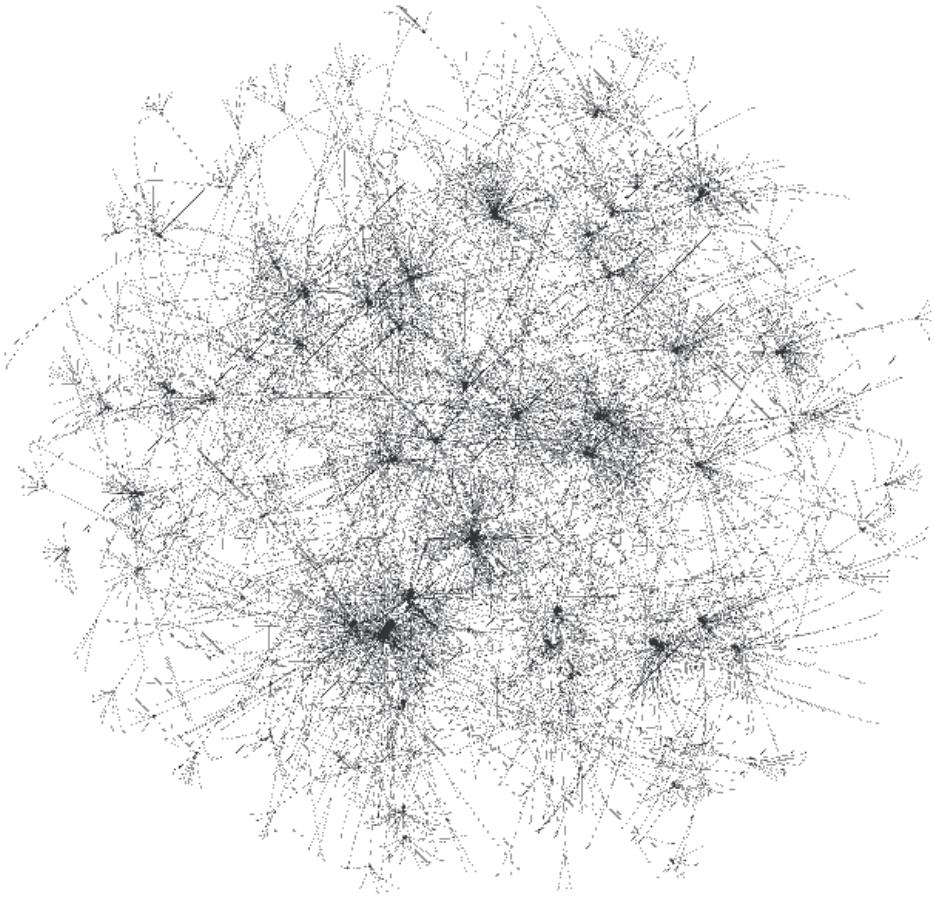

Figure 2.2: A visual representation of branching complexity in an application structure.

CHAPTER 3

EXPERIMENTS

3.1 Research Questions

There are four main questions of focus that the following experiments attempt to answer:

1. What is the impact on the size of the test suite using abstract URLs instead of an exhaustive enumeration of every variable combination?

2. What is the fault finding effectiveness of testing with abstract URLs versus exhaustive testing?

3. What is the impact on the size of the test suite using combinatorial coverage compared to single coverage?

4. What is the fault finding effectiveness of the combinatorial coverage compared with single coverage?

3.2 Sample Application

All experiments were conducted on a sample Rich Web Application written by the author and is available at `http://chadmaughan.com/thesis`. The application provides United States census information for 1990, 2000, and preliminary numbers from 2005 for each state, county, and city with a population over 25,000 residents. The application is designed for mobile devices with larger touch points. The sample application uses the following technologies:

1. Backbone.js: A client-side, JavaScript MVC framework that gives structure to rich web applications. Backbone.js allows you to bind custom events to models, and route URL fragments to JavaScript functions.

2. jQuery Mobile: A unified, HTML5-based cross-platform user interface system for mobile devices. It focuses on semantic markup that is easily themeable.

3. Spring MVC: A module of the popular Spring Framework for Java that allows for easy implementation of server-side controllers for building REST APIs.

4. Apache Derby DB: An open source, embedded relational database implemented in Java.

Source code for both the application and the fault finding, testing code is also available at `http://code.chadmaughan.com/thesis`. Metrics about the sample application are included in Table 3.1. A sample screenshot of the application is provided in Figure 3.1.

Table 3.1: Sample Rich Web Application Information.

Number of Application States	199,484
Number of Files	328
JavaScript Lines of Code	605
Java Number of Classes	21
Java Lines of Code	1,091
Seeded Faults	73

3.3 JavaScript Faults

The sample application is seeded with 73 faults. A detailed list of those exceptions and the categories they belong to are listed in Table 3.2.

On fault categories, Sampath et al. described five web application fault categories [**?**], of which all are used in the sample application.

1. Data store faults: Faults in the application code that manipulates data in any kind of data store. This category of faults also applies to data that is incorrectly persisted

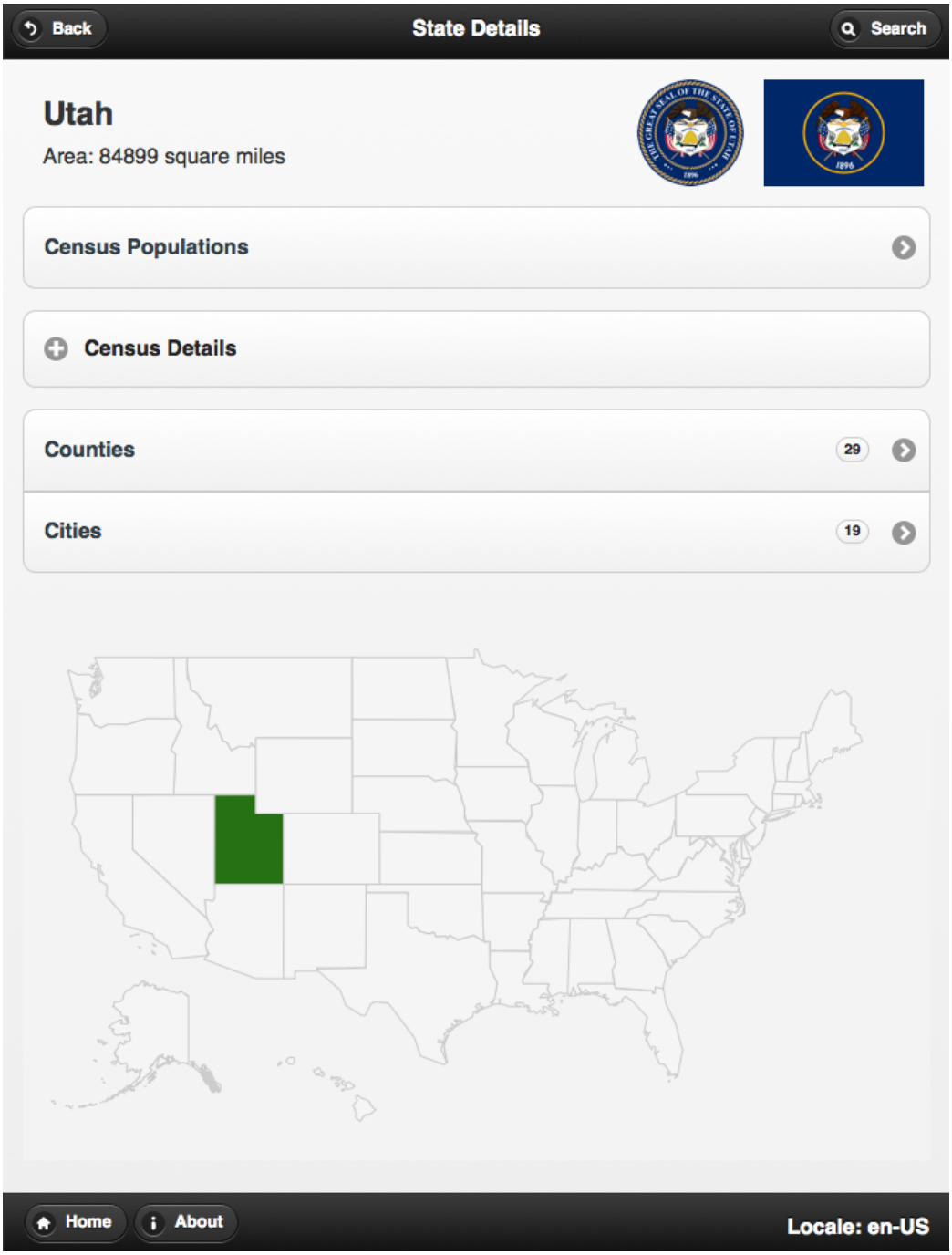

Figure 3.1: Sample rich web application screenshot.

in the data store. There are a number of seeded data store faults in the sample application.

2. Logic faults: Faults in the application code that implements business logic and control flow. An example of this is an error in the page transition with jQuery Mobile.

3. Form faults: Faults in the application code that controls, modifies and displays name-value pairs in forms. The sample application does not submit any data to the server but does use dynamically updated forms to direct to different application states.

4. Appearance Faults: faults in the application code that controls the way in which a web page is displayed. With the use of modern JavaScript based templating solutions, such as Mustache.js and Underscore.js, appearance faults typically cause a template to not be rendered. The sample application demonstrates this type of error when trying to load the search page.

5. Link faults: Faults in the application code that changes the page pointed to by an URL.

Guo and Sampath [**?**] add a sixth category, compatibility faults or "faults in application code that ensures that the web application complies with different browsers, versions of browsers and other client environments." With the rapid introduction of new HTML5 APIs (i.e., Websockets and Webstorage) and the varying speed of adoption among browser creators, compatibility faults will play an increasingly important role in client-side testing.

Guo also expands the logic faults category to include seven sub-categories. The sub-categories of logic faults are:

1. Browser interaction faults: Faults in the application code that control the web browser, such as code that disables the "back" button on the browser, or code that is affected by user-defined browser settings, such as disabled cookies. Browser manipulation is discouraged as it alters expected application behavior. Other browser settings, such as disabling JavaScript, would render the application useless. As such, the sample application does not use this fault sub-category.

2. Session faults: Faults in the application code that deal with maintaining state of application or other session-based operations such as using sessions to save and data entered into a form and display the data after the sessions has been validated. As most rich web applications are stateless to avoid the overhead of session management and allow for greater scalability, the sample application does not use this sub-category.

3. Paging faults: Faults in the application code that deals with paging when displaying large amounts of data on the screen. While applicable to rich web applications, the sample application does not use this sub-category.

4. Server-side parsing faults: Faults in the application code that deal with server-side parsing of HTML, XML, and JavaScript tags. This sub-category does not apply to rich web applications as HTML used in a RWA is typically a minimal page used only as an entry point. "client-side parsing faults" are more applicable to rich web applications. The sample application has a number JavaScript syntax related faults. These errors are described below in Table 3.2.

5. Encoding/decoding faults: Faults in the application code that encodes or decodes characters for transmission, storage, and display

6. Locale faults: Faults that exist in application code that sets or gets locale-specific information, such as date format or language. Not used in the sample application.

7. Other: Other logic faults that do not belong to any of the above sub categories.

More recently, in addition to Sampath and Guo, Ocariza et al. [**?**] defines five categories specifically for JavaScript exceptions. He found that JavaScript exceptions tend to be much more defined than other applications and fall into well-defined categories. In fact, 94% of all errors studied from the Alexa top 100 list fall into five categories, namely:

1. Permission Denied: Faults in the application code that attempt to access JavaScript components from another domain violating the same-origin policy. The sample application has a seeded fault where it tries to load recent search data from Twitter and violates the same-origin policy.

2. Null Exception faults: Faults in the application code where a property or method is accessed via a null object. The sample application attempts to add a CSS class name to an element that is null.

3. Undefined Symbol faults: Faults in the application code where a function or variable is accessed that has not been previously defined.

4. Syntax Error faults: Faults in the application code where interpreted code, such as in an eval() function, has the wrong syntax.

5. Miscellaneous faults: Faults in the application code that apply specifically to a single site.

These five categories align more closely with the JavaScript Error object and its six other core errors: EvalError (Syntax Error faults), RangeError, ReferenceError (Undefined Symbol faults), SyntaxError (Syntax Error faults), TypeError, and URIError [**?**].

3.4 Testing Technologies

In addition to the sample Rich Web Application, the code used to identify the seeded faults rely on some key technologies. These technologies are listed below:

1. Neo4j: A powerful graph database with a rich API that was used to both systematically identify variables in semantic URLs and store a full navigation graph of the rich web application for test traversal retrieval.

2. BrowserMob: An embedded proxy software that allows the interception of HTTP requests from the client and HTTP responses from the server. This allows for the

Table 3.2: Sample Rich Web Application Seeded Errors.

Location	Description	Category	Count
index.html:39	console.log(variable) - variable is not defined.	ReferenceError, Undefined	1
main.js:54	SearchPageView template is not included as a source.	ReferenceError, Appearance	1
geochart:438	County Map doesn't exist on County Details page. "Container is not defined" error.	Appearance	1 (3,144)
main.js:49	Syntax Error on eval() - eval("window.print(;")	SyntaxError	1
main.js:37	Missing script file - notthere.js"	Link, Other, Misc	1
jquery-1.7.1.min.js:4	Multiple words in the name (9 states plus Washington DC) doesn't have a flag to display		10
jquery-1.7.1.min.js:4	Multiple words in the name (9 states plus Washington DC) doesn't have a seal to display	Appearance	10
CityController.java:39	Cities with periods in their name/code, return HTTP 500 (i.e., St. George, UT).	Data Store	13
index.html:87	Type Error 'bad type' for all Washington State cities	TypeError, Misc	37
twitter.js:4	City Twitter Search feed	Permission	1 (1,267)
census-table-view.js:17	Adds a CSS class to a null element	Null Exception, TypeError	1

identification of network errors and for the modification of the server responses to assist in identifying client side JavaScript errors.

3. Selenium WebDriver: Allows for the crawling and interaction with the application in the various browsers "as the user would," giving a more accurate result while testing.

3.5 Navigation Graph

A navigation graph is a visual representation of an application [**?**]. A navigation graph of a rich web application is a visual representation of how different states of the web application are related to one another. A traditional web application navigation graph typically would have a single node for each HTML page or a single node for each rendering of a

server-side script. An edge on a traditional web application represents an HTML anchor tag. As rich web applications typically have a single or very limited number of HTML pages acting as entry points, a node in the navigation graph of a rich web application represents a single state of the application. An edge in a navigation graph for a rich web application represents a transition from one state to another. Like in a traditional web application, this is also typically done through an HTML anchor tag.

Building a navigation graph of the rich web applications has many benefits, including understanding the application as a whole, maintenance over the development cycle, and test case generation and the subsequent testing of those tests [**?**]. To build the graph, and expand it as the application increases in size, a rich web application can be crawled using the Selenium WebDriver in a simple depth first traversal. Figure 3.2 demonstrates the crawling algorithm using Selenium WebDriver.

Also of benefit is the ability to identify key, holistic characteristics about the application as the navigation graph is built. For example, while this experiment does not focus on test prioritization, systematically creating a navigation graph with Selenium WebDriver allows you to easily record and track key information about state transitions in the application. As an example, one may wish to prioritize testing based on the location of links or elements that trigger a change in state. Combining the exact top and left pixel location of that element allows you to prioritize links that are more in the key navigation sections. As another prioritization strategy with a navigation graph, one could also easily prioritize based on states of the application that are most transitioned.

3.6 Abstract URLs

One problem with the depth-first traversal of a rich web application is the time it takes to complete a full application crawl. For example, the sample census application discussed previously contains nearly 200,000 different states. Assuming an average page load of around 500 milliseconds, it would still require more than 27 hours to perform a full regression test. While this amount of time to perform an exhaustive regression test is possible, it may not always be practical.

```
100 public static void main(String[] args) {
101     driver = new FirefoxDriver();
102     new Crawl("http://example.com");
103 }
104
105 public Crawl(String startingUrl) throws Exception {
106     String newAddress = null;
107     while ((newAddress = queue.poll()) != null)
108         processPage(newAddress);
109 }
110
111 private void processPage(String sourceHref) throws Exception {
112     driver.get(sourceHref);
113     List<WebElement> links = driver.findElements(By.tagName("a"));
114     Node sourceNode = retrieveOrCreateNode(sourceHref);
115     String targetHref;
116
117     for (int i = 0; i < links.size(); i++) {
118         WebElement w = links.get(i);
119         targetHref = w.getAttribute("href");
120         if (isValidUrl(targetHref)) {
121             if(w.isDisplayed()) {
122                 Node targetNode = retrieveOrCreateNode(targetHref);
123                 if(targetNode == null)
124                     targetNode = nodeMapper.mapNode(targetHref, w);
125                 sourceNode.createRelationshipTo(targetNode,
126                         RelationshipTypes.LINKS_TO);
127                 if (!processed.contains(targetHref))
128                     queue.add(targetHref);
129             }
130         }
131     }
132 }
```

Figure 3.2: Depth first crawling of a rich web application

Using variables previously identified through the algorithm described in the background chapter, application states can be stored in the navigation graph as "abstract URLs" [**?**], or URLs with variable names instead of each possible value. Similar to semantic URL formats discussed earlier, an example of an abstract semantic URL stored in a navigation graph would look as follows:

http://example.com/#/state/{var1}/county/{var2}/city/{var3}

Storing the abstract URL instead of every URL variable combination significantly reduces the number of tests required. Indeed, it is ideal as most rich web application states reuse small HTML templates for the displaying of particular data points. This means that client-side errors would typically manifest themselves for every variance of a variable. One disadvantage of using abstract URLs is that data specific errors may be missed. For example, the sample application has a fault where any city with a period in it's name (i.e., "St. George, UT") doesn't render correctly. Unless the random value for the abstract URL variable selected contained a period, this data related fault would be missed. While not discussed in detail in this paper, due to the available resource of the full application structure (from the systemic variable identification), one possibility is to use a sample size confidence level to adequately test a certain size of the data available.

One of the experiments completed was calculating and comparing the amount of time required to crawl a full rich web application, a reduced number of states based on a sample size, and a fully reduced test suite testing only abstract URLs one time. Results of this experiment are discussed in the following chapter.

3.7 Capturing Errors

Another difficulty encountered with testing rich web applications is the ability to identify errors that occur in the browser. JavaScript errors are difficult to systematically intercept and report. This section introduces a novel approach for identifying two different

types of exceptions during testing, namely network related and JavaScript browser related.

Some network related exceptions are errors that are simple to identify on the server-side by examining the log files. BrowserMob was used as an embedded proxy to intercept all communications between the browser and the server. This allows for the logging of network related exceptions, as well as response interception and subsequent modification for JavaScript exception reporting.

JavaScript exceptions in modern browsers typically manifest themselves via the browser console object. The console is typically not visible to the end user and needs to be enabled via menu options or browser plug-ins such as Firebug for Firefox. A sample of thrown exception is shown in Figure 3.3.

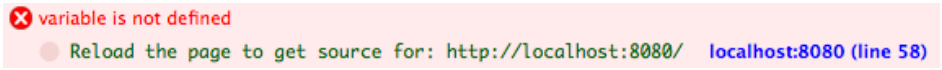

Figure 3.3: An example JavaScript error in the Firebug console.

Unfortunately, once these exceptions are thrown, there is no way to retrieve them from the browser. A simple way to capture these is to introduce a small snippet of JavaScript code in every HTML entry point that has a collection for inserting exceptions. This requires that testing code be added to the deployable deliverable, considered by many to be bad practice. As an alternative, this experiment introduces a novel approach to keep testing scaffolding out of the deployable code base. BrowserMob is used in to intercept only full HTML page responses from the server to the browser, and alter it injecting a custom browser Console object at the beginning of the HTML <script> tag. This custom Console object stores any exception for later retrieval and reporting. Figure 3.4 shows the custom JavaScript injected into each response.

3.8 Combinatorial Testing

The preparation done with the creation of both the exhaustive and the reduced navigation graph has prepared for the generation of combinatorial test cases. The sample

```
100 <script type="text/javascript">
101     window.jsErrors = [];
102     window.onerror = function(errorMessage) {
103         window.jsErrors[window.jsErrors.length] = errorMessage;
104     }
105 </script>
```

Figure 3.4: Browser exception catching

application has some variables, namely state, county, and city, that are hierarchical in nature. Czerwonka [**?**], while describing the capabilities of the Microsoft PICT combinatorial test generation tool, explains that these hierarchical variables are treated as a "sub-model" and pairwise combined first to represent a single variable that is then used in the generation of the combinatorial test cases. For example, in the sample application on the home page, the State of Utah, Cache County, and Logan City are combined together to form a single variable that is then used in building the combinatorial tests. Additionally, a great majority of exceptions that occur in rich web applications result from the different browsers and environments. Tatsumi [**?**] made the distinction of input and environment variables. The input variables discovered via the process introduced previously can then be combined with provided environment variables to also identify "compatibility faults." I discuss in the concluding chapter future work that can add distributed computing capabilities enabling browsers in different client environments to perform the same tests.

CHAPTER 4

RESULTS

Each section of this chapter focuses on the results of one of the four research questions discussed in the experiments chapter. First, an abstract URL test suite is compared to the full exhaustive test suite in both size and effectiveness. Second, a more effective combinatorial coverage based test suite is compared to the same exhaustive test suite.

4.1 Size Impact of Abstract URL Test Suite

Using the algorithm described in Figure 2.1, one is able to identify variables in semantic URLs. These variables are then used to build abstract URLs that represent a single application state for each template or combinations of templates in the sample application. The abstract URL test suite does not represent every possible data combination represented in the sample application. The exhaustive count represents every application state as a combination of data plus templates. The hypothesis of this experiment was that the abstract URL test suite would find less faults than the exhaustive test suite but would miss most, if not all, data related faults.

For this relatively small application, one can quickly understand the value of identifying abstract URLs. Indeed, assuming an average page load latency of around 500 milliseconds, running the exhaustive test suite would require more than 27 hours to perform a full regression test.

One option for future work would be to increase the number of abstract URL tests from a single random data point for each application state to a statistically relevant sample of random data points for each application state and compare the effectiveness to the exhaustive test suite. Table 4.1 compares the sizes of the two strategies.

Table 4.1: Size of Test Suites by Strategy.

Strategy	Number of Tests
Exhaustive	199,484
Abstract URLs	27

4.2 Effectiveness of Abstract URL Test Suite

The exhaustive test suite yields significantly better results in fault detection compared to the abstract URL test suite but also took the longest amount of time. In real-world scenarios, it may not be practical to let a full regression test of nearly 200,000 application states run for more than a day before deploying.

Table 4.2: Effectiveness of Single Coverage With Abstract URL Reduction.

Strategy	Number Faults Identified	Percent Faults Identified
Exhaustive	69	89.6%
Abstract URL	15	19.5%

4.3 Size Impact of Combinatorial Coverage

The test generation code, when discovering a <form> element on any snippet of HTML code, analyzes it looking for related or hierarchical variables to apply combinatorial algorithms to test generation. Not all of the application states in the sample application have multiple variable combinations as an option. Some pages, such as the application "about" page, have no variables. As the combinatorial test generation algorithms do not apply to them, I have combined both generated combinatorial tests and the remaining "uncombined" states (i.e., about page) to provide a more comprehensive test suite to better compare with the exhaustive, single coverage testing strategy. The single, exhaustive coverage and combinatorial testing coverage had the following number of tests in their respective test suites.

Table 4.3: Size of Test Suites by Strategy.

Strategy	Number of Tests
Exhaustive, Single Coverage	199,484
2-way Combinatorial Coverage	25,560 (13,387 combinatorial + 12,173 remaining)

```
100 <fieldset data-name="unit" data-role="controlgroup">
101     <select data-hierarchical="unit" name="state" id="state">
102         <option>Select State</option>
103     </select>
104     <select data-hierarchical="unit" name="county" id="county">
105         <option value="">Select County</option>
106     </select>
107     <select data-hierarchical="unit" name="city" id="city">
108         <option value="">Select City</option>
109     </select>
110 </fieldset>
```

Figure 4.1: Hierarchical variable HTML form example

Inside an encountered <form> element, the combinatorial test generation will search for any input element or select element as well as a "data-hierarchical" attribute that indicates these options should be pre-combined before applied as a option in a variable in the combinatorial test generation. An example <form> element can be found in Figure 4.1. The "data-hierarchical" attribute tells the test generation code to combine them as a sub-model before applying it to a combinatorial algorithm. A sample of the pre-combined sub-model data would look like is found in Table 4.4.

With the unit variable type pre-combined in the sample application, combinatorial test generation can then be applied according to Table 4.4.

Table 4.4: Sample Combinatorial Tests.

Unit	Year	Data Point
/state/alabama	1990	Population
/state/alabama/county/autauga	2000	Density
/state/alabama/county/lee/city/auburn-city	2005	Ranking
/state/alabama/county/tuscaloosa/city/tuscaloosa-city		
...		
Remaining combined units		

4.4 Effectiveness of Combinatorial Coverage

The single, exhaustive coverage testing yielded a slightly better fault detection but required a full order of magnitude more tests to achieve that result, 199,484 compared to 25,560. Assuming an average latency of 500 milliseconds would require only 3.5 hours to execute all test cases in the combinatorial coverage test suite, or approximately 23.5 hours less time required to test the exhaustive test suite on the sample application.

Table 4.5: Effectiveness of Combinatorial Coverage.

Strategy	Number Faults Identified	Percent Faults Identified
Exhaustive, Single Coverage	69	89.6%
2-way, Combinatorial Coverage	68	88.3%

CHAPTER 5

CONCLUSIONS

Combinatorial testing is a powerful tool in identifying software faults. Indeed, Kuhn, et al. showed in a study of a NASA Distributed Database that 93% of all faults were identified by 2-way combinations, and 98% by 3-way combinations. Across industries, "the detection rate curves for the other applications studied are similar, reaching 100% detection with 4 to 6-way interactions" [**?**].

Rich web applications continue to grow with the introduction and rapid development of new HTML5 features and APIs, powerful JavaScript based frameworks, and increasingly more powerful client machines. This thesis demonstrates that combinatorial testing can play an important role in the testing of rich web applications and that more future work is needed.

While there is not a specific standard for semantic URLs compared to the traditional URL format defined in RFC 1378, this paper has shown a novel way to identify variables by employing graph theory and branching complexity analysis. This approach worked with the sample application but needs additional work to be universally applicable to all semantic URL formats. For instance, an application that has a small number of variables that equal the branching complexity of the URL structure could result in false positives with variable identification.

This research also showed that using abstract URLs to generate test cases was an effective and inexpensive way to discover all types of faults except data faults in the sample application and particularly well suited for template heavy client-side applications. While it didn't find as many errors as the exhaustive or combinatorial approaches, with only 27 tests (0.0001% of the exhaustive tests run) it found 19.5%. Using abstract URLs may be a good strategy when time is extremely limited. A point of future work that may possibly yield

better results with abstract URLs would be to increase the number of random variables used from a single variable to a statistically significant percentage of total data points available.

Also demonstrated in this paper was a convenient way to capture JavaScript exceptions by intercepting HTTP requests and responses via an embedded proxy server, and then injecting a JavaScript array in the <HEAD> section of each HTML page to capture any thrown exceptions.

Future work may extend testing to distributed machines and different client environments. For example, the environment variables shown in Table 5.1 could be combined with input variables to help catch compatibility faults, an ongoing concern with the various browser creators adoption rate of new HTML5 features.

Table 5.1: Environment Variables.

OS	Browser
Linux	Chrome
OSX	Firefox
Windows	IE
	Safari

While I attempted to make the sample application as "real world" as possible, additional work is needed to make the testing tool better and more practical for rich web applications in real world scenarios. An initial effort was made to implement ideas in Microsoft's PICT tool [**?**], such as making a better distinction between preparation and test generation, pre-combining related, hierarchical fields, and providing test generation guidelines by employing the new HTML5 data-* attributes. As an example of future enhancements to the testing software, one could create a new "data-exclude-test" to keep a particular variable from being combined for test case generation. Also, guidelines on associated form variables with attributes in the ¡fieldset¿ tags would allow for better control of test case generation with variables. Integrating in with build tools would also be beneficial to keep those guideline attributes from being deployed to production. More future work that would be beneficial would be better testing of all JavaScript event types and application state

transitions that are not associated with an update to the URL fragment (i.e., an event triggered by the clicking of a non anchor tag that updates a value in an existing portion of the Document Object Model (DOM) structure).

Despite much future work left to be researched, this thesis demonstrates that test case generation using combinatorial coverage strategies in rich web applications, as in other application types, provides much benefit in identifying faults and should be explored further.