Automatic Generation of Artifacts for the Application of Two Web Application Testing Models

**Submitted by:** Sarah Vessels

**Advisor:** Dr. Jane Hayes

Department of Computer Science

University of Kentucky

Lexington, KY 40506

March 2011

**Committee members:**

Dr. Raphael Finkel

Dr. Judy Goldsmith

Dr. Jane Hayes

# Abstract

Web applications are prevalent and it is important that they be of high quality as businesses, schools, and public services rely upon them. Web applications can be tested using testing models designed for web applications. Unique testing models designed for web applications can be beneficial since web applications differ greatly from desktop applications. Web applications are accessed via a browser, and a user can manipulate the web application in ways not possible with desktop applications, such as modifying the URL or using the Back button in the browser. It can take a great deal of time to apply a given web application testing model to a web application. This project seeks to speed up the process of applying two particular web application testing models: the Atomic Section Model and the Qian, Miao, Zeng model. Two generators were written using the Ruby programming language, one for each model. The generators take as input the source code of a Ruby on Rails web application, and the URL to a web application written with any framework, respectively. They produce as output test paths that can be traversed manually to ensure good coverage of the web application, and artifacts that can be further manipulated manually to produce test paths, respectively.

Contents

[Abstract 2](#_Toc288065867)

[Definitions 4](#_Toc288065868)

[Introduction 5](#_Toc288065869)

[Implementing the Qian, Miao, and Zeng Model 6](#_Toc288065870)

[Sample Output from the QMZ Tool 9](#_Toc288065871)

[Implementing the Atomic Section Model 9](#_Toc288065872)

[Sample Output from the ASM Tool 10](#_Toc288065873)

[Conclusions and Future WorkBibliography 10](#_Toc288065874)

[Bibliography 11](#_Toc288065875)

# Definitions

* **CSS** - Cascading Style Sheets; a language used to describe the appearance of a web page
* **CSS** **selector** - a pattern matching rule in CSS that is used to determine which styles apply to which elements in an HTML page
* **ERB** - "a templating system that embeds Ruby into a text document... often used to embed Ruby code in an HTML document, similar to ASP, JSP and PHP" [[[1]](#endnote-2)]; sample file extensions of an ERB file include .rhtml and .html.erb
* **ERB output tag** - found in an ERB file, this tag allows Ruby code to be evaluated and the result to be output when the ERB file is used as a template; the result of the Ruby code within an ERB output tag is displayed as part of the web page when it is included in a Ruby on Rails ERB view; it is started by <%= and ended with %>
* **ERB tag** - found in an ERB file, this tag allows Ruby code to be embedded in an ERB file; the results of the evaluation of the Ruby code will not be included as part of the source of the output file once the ERB is applied as a template; an ERB tag is started by <% and ended with %>
* **form transition** - a form within a web page that has a destination page and a button that, when activated, navigates the browser to the destination page
* **gem** - a Ruby library, often installed through the use of the RubyGems package manager[[2]](#footnote-2)
* **HTML** - HyperText Markup Language; "the predominant markup language for web pages... the basic building-blocks of webpages" [[[3]](#endnote-3)]
* **hyperlink** - a link within a web page that, when followed, will navigate the browser to the target of the hyperlink
* **RHTML** - see ERB
* **Ruby on Rails view** - the Ruby on Rails framework uses the Model View Controller paradigm [[[4]](#endnote-4)], so a Rails view is intended to contain code to be displayed in a web browser, including HTML, Javascript, and CSS
* **s-expression** - symbol expression, a.k.a. sexp; s-expressions are "list-based data structures that represent semi-structured data" [[[5]](#endnote-5)]; the Ruby gem ruby\_parser class RubyParser returns s-expressions when Ruby code is successfully parsed [[[6]](#endnote-6)]
* **URI** - Uniform Resource Identifier; used to identify the location of a web site on the Internet
* **web scraper** - a tool that saves the source of a web page for later processing

# Introduction

Web applications require testing just as any software does, in order to ensure correct functionality, expected appearance and layout, and that correct data is displayed. Testing web applications is different than testing desktop applications because the primary way of using a web application is through another application: a web browser. Because of this method of access, web applications are subject to the user’s manipulation in ways that desktop applications cannot usually be manipulated. A user can manually change the URL in the web browser to point to another page in the web application. Users can also use their browser’s Back and Forward buttons to change pages. Different plugins, browser settings, and browser extensions alter the user experience. All these things can provide the web application with unexpected input, or cause it to perform actions in such a way that the designers did not anticipate. It is for this reason that different testing models exist solely for testing web applications.

One such model is the Atomic Section Model (ASM) [[[7]](#endnote-7)] by Jeff Offutt and Ye Wu. The ASM represents all possible user interface screens of a web application, just as control flow graphs represent all possible sequences of software program execution. ASM can be divided into two levels: the Component Interaction Model (CIM) and the Application Transition Graph (ATG) [2]. CIM is designed for the individual components of the web application while ATG is for the web application as a whole, in which the individual components come together and function as a system.

Another web application testing model is the Qian, Miao, and Zeng model (QMZ) [[[8]](#endnote-8)]. Applying the QMZ model involves the creation of a Page Flow Diagram (PFD) [3], which represents “the relationship among pages of a web application” [3]. A Page Test Tree (PTT) [3], based upon the PFD, is also necessary for generating shorter paths than those shown in the PFD. A PTT is a tree-like structure that is a simpler representation than that provided by the PFD of pages in the web application.

As part of this project, I have written tools in Ruby to partially automate the process of applying the ASM and the QMZ models to a web application. Ruby was chosen because it is a language with which I am comfortable, and a good number of libraries exist to do various things that were necessary in this project, like parse HTML or scrape web pages. Due to the nature of the QMZ model, the underlying language of the web application being tested does not matter, so the QMZ tool is platform-agnostic. The ASM works by considering the code of web pages in the application, and so it was necessary to pick a programming language/framework to target when writing a tool for generating ASM artifacts. I chose to target the Ruby on Rails web application framework because it is popular[[9]](#footnote-3) and there are many web applications written with the framework whose source is freely available online, which provided good test input for the ASM tool. Though these tools were been written and tested in OS X 10.5.8 and Windows 7, using Ruby 1.8.7, they should be executable in any operating system for which there is a Ruby interpreter. The different gems my tools use must also be installed.

Note that these tools do not fully automate web application testing, which must still be done manually by a human. The QMZ tool generates test paths to follow (i.e., URIs to access in a particular order) to achieve good coverage of the pages and transitions in a web application, where "good coverage" is defined according to the QMZ model. The ASM tool generates CIMs which can be combined into an ATG and then traversed in various ways to achieve coverage of the web application. Test cases would need to be constructed manually for each model to provide input to forms encountered in the web application, and to describe how each page is expected to appear when it is reached after following a certain path of web pages. Manual testing is required to ensure, upon following these test paths, that the web pages display as expected and that no errors occur. My contribution with this project is to remove some of the tedium of applying these web application testing models, especially for large web applications with many pages.

# Implementing the Qian, Miao, and Zeng Model

To apply the QMZ model to a web application, I had to represent the PFD and PTT for a given site. To construct these, the extent of the web application must be known. My QMZ tool first functions as a web scraper to download pages it determines to be part of a given web site, then extract from those pages the links to other pages within the site. To execute my QMZ tool, the Ruby script scraper.rb is run on the command line with various parameters. A URI to the web application can be passed with the -u option followed by the URI. For example: scraper.rb -u "http://mywebapp.example.com/". The scraper script constructs a new instance of my Page class, which is then used to construct a new instance of my Site class.

A Page requires a URI either as a Ruby String or an instance of the URI class. Upon creating a new Page, the Page class attempts to download the given URI; if it fails, an ArgumentError is thrown. If the URI is successfully downloaded, the HTML of the page is scraped for links. To the QMZ model, links come in the form of hyperlinks and form transitions. To identify these in a given chunk of HTML code, the Ruby gem Nokogiri [] was used. Nokogiri is an HTML, XML, SAX, and Reader parser, though only its HTML-parsing abilities were used in my project. The Nokogiri::HTML module is passed a chunk of HTML that it parses. The result is a Nokogiri::HTML::Document that can be traversed using CSS selectors such as 'a' or 'form'. Using the css method on a Nokogiri document, a list of matching elements from the HTML can be extracted. For example, Nokogiri::HTML(anHTMLString).css('form') would return a list of all the form elements in the given HTML string, case insensitive. The css method can be called on each of these elements in turn to find other elements contained within them, such as buttons in a form. Finally, the values of particular attributes can be extracted from HTML elements through Nokogiri, so the target of a form--its action attribute--can be extracted. A Page finds all forms in the HTML that it scraped from the given URI, then narrows that list down to forms that have a target and have some type of button the user can activate to submit the form. It further narrows down this list of form elements to those whose targets point to another page in the same site that is being scraped. This is determined by comparing the URI host of the target with the Page URI; relative links (links with a URI that does not specify a host) are always included.

Hyperlinks are similarly extracted from HTML by the Page class using Nokogiri. Hyperlinks in HTML are written using the a tag, so Nokogiri's css method is called with the CSS selector parameter 'a'. The resulting list of hyperlinks is pared down to just those that have a target--the href attribute--that is within the same site that is being scraped.

For both form transitions and hyperlinks, a description of the link is retained for later use. For form transitions, the description is the value of the submit button, or the source of the image if it is an image submit button. For hyperlinks, the description is the link text. These descriptions are retained so that when test paths are presented to the user, the user knows how to access a particular link from a page. This is helpful because there may be multiple ways to access a given page, for example if a single form had a submit button with the value "Confirm" and another submit button with the value "Reject". These two buttons would be treated as separate transitions from the source page to the target of the form, thus test paths would be generated to traverse both of them. The server might do different things depending on which button was used, so it is good to test both possibilities.

When a new instance of my Site class is created, the root/home page of the site is passed as a parameter. This is represented by an instance of the Page class, which stores information about the page's URI and the links found within the page's source code. From that home page, the Site class will create other Page instances to represent the other pages in the web application. If two separate pages in a site have a link to the same target page, only one copy of the target page will be represented as a Page. Likewise if two pages have different links that resolve to the same target page, such as when an HTTP redirect is performed by the server.

Two different URIs can point to the same page without a redirect being involved; for example, http://google.com and http://google.com/#comments. The latter URI points to the same page, but to a different marker in that page. To determine if two different URIs point to the same page in order to avoid scraping the same page multiple times, my tool considers the scheme (e.g., http), host (e.g., google.com), and request portion of the URI. The request portion will contain parameters such as ?query=test, which could affect which page is displayed, but not #comments.

If pages link to images, PDF files, or other non-HTML files, these will be skipped. If any link within a page produces a 404 File Not Found or similar error, or is a link to a non-HTML page, that link will be retained in a blacklist of links that should not be traversed. This blacklist helps speed up the process of scraping a web application by not repeatedly attempting to scrape a page that failed previously. The Site.get\_pages method is recursively called to traverse the web application through its form transitions and hyperlinks. The QMZ model knows only about the parts of the web application that are accessible via these links. If there were some page that would be accessible by directly entering a URI in a browser, but that was not linked to by any other page in the site, that page would be skipped by the QMZ model and my tool.

Once a Site has been constructed, the get\_pfd method is called to construct a PFD that represents the web application. This method returns an instance of my PFD class, which contains pages and links, represented by the Page and Link classes, respectively. The QMZ model describes a PFD as containing pages, links, and flows. Flows are modeled in my tool via the Link class, which contains the source URI, the target URI, and the target Page. A list of Link instances is constructed in the get\_pfd method by iterating over each Page in the Site, and then iterating in a nested loop over all the link URIs within that Page. A target page is found by extracting the Page with the same URI parts (scheme, host, and request) as the current link URI. A list of Link instances is maintained and used, along with a list of Page instances and the root URI, to create a new PFD instance.

The PFD is immediately converted into a PTT, so as to remove cycles that may exist among pages in the PFD [3]. This is done via the Site.pfd2ptt class method, which takes in an instance of the PFD class and returns a new instance. A PTT consists of pages and links, like a PFD, and so the same class (PFD) can be used to represent them both. My pfd2ptt method works by making a deep copy of the provided PFD and then applying the PFD2PTT algorithm [3] from the QMZ model to that deep copy. Deep copies are made of Page instances, as per the algorithm, and an is\_copy flag is set on those pages for use later during traversal. The deep copy is thus modified to remove cycles and now "is a tree expressing the successive relationship among pages" [3].

Once the PTT is acquired, scraper.rb calls the get\_test\_paths method on the PFD instance, which returns an array of test paths. The method works by doing a preorder traversal on the PTT, calling the private PFD method preorder. The preorder method takes in a Page; an integer level; an array of test paths, each of which is itself an array; and a description of the link followed to get to that page. An instance of the LinkText class is used to pair a URI with a description of how the URI was reached; each test path is an array of LinkText instances. The first LinkText in any test path always represents the root page of the site, with the description "Start page". A new LinkText is constructed with its URI set to the given Page's URI, and its description set to the given description. The choice of where to put the LinkText is between 1) the end of the last test path in the array of test paths, or 2) the end of a new test path that begins the same way as the last test path in the array of test paths. If the number of LinkTexts in that last test path is less than or equal to the current level, then the LinkText is appended to the end of that test path. Otherwise, a new test path is created by duplicating a range of the last test path, from the first index up to the levelth index, and then the new LinkText is appended to the end. That new test path is appended to the end of the array of test paths. If the given Page does not have its is\_copy flag set, the page's links are iterated over and preorder is recursively called for each one, with the parameters being the target page of the Link, the current level plus one, the array of test paths as it is now, and the Link description. Because of this recursive call, the array of test paths will be modified to either have a new test path appended to the end, or to have the final test path get a new LinkText. Finally, the array of test paths is returned from preorder.

The result of the preorder method is an array of arrays of LinkText instances, which is returned by PFD.get\_test\_paths. These represent the test paths a user should take, with each starting at the root page of the site and ending on one of the pages in the site. The test paths differ by both the pages they traverse as well as the transitions used to travel from one page to another. Once scraper.rb gets the test paths, it then creates an HTML page to display them. It styles the page using CSS and stores both the HTML page and the CSS file in a new directory whose name is based off the root page URI of the site. If the root URI of the site were http://google.com/, then the directory containing the test paths would be called http.google.com.

# Sample Output from the QMZ Tool

# Implementing the Atomic Section Model

To implement the Atomic Section Model, I had to make a choice about the target language/framework. Once the decision was made to target Ruby on Rails, it was necessary to identify atomic sections within the ERB files that make up the output of a Rails web application. After atomic sections have been identified, component expressions must be identified. Transitions can also be identified after atomic sections have been found. The combination of atomic sections, transitions, and component expressions forms a Component Interaction Model for the page.

Component expressions represent the structure of a page in the Atomic Section Model. They list atomic sections in the order in which they appear in the page, and indicate whether the sections are in a sequence, selection, iteration, or aggregation. A sequence of sections p1 and p2 is represented in the output of my tool by p1.p2, and it indicates that atomic section p2 follows p1 in the page. A selection of sections p1 and p2 is represented by p1|p2, and indicates that either p1 or p2 will be displayed when the page is rendered in the browser, but not both. An iteration of section p1 is represented by p1\*, and indicates that section p1 will be displayed consecutively an arbitrary number of times. Finally, an aggregation of sections p1 and p2 is represented by p1{p2}, and indicates that p2 is included as part of p1.

Once component expressions have been constructed, transitions from one page to another via hyperlinks and forms are identified. The atomic sections, component expressions, and transitions together make up the Component Interaction Model for a page, and a list of CIMs is the output of my ASM tool.

The most difficult hurdle to overcome in the design of the ASM tool was how to generate a component expression for a given page. The tool can handle a combination of HTML, Ruby, and JavaScript in an ERB file, but it requires the Ruby to be valid. This seems a reasonable requirement since if a page in the web application being tested has invalid Ruby, the web application will generate an error when that invalid Ruby code is reached, and so my tool is still fulfilling its purpose of helping identify errors in the application.

The tool must be able to identify where an ERB tag or an ERB output tag occurs in the file, and treat those differently than HTML or JavaScript. To know where these ERB tags occur, a custom grammar was written for ERB files using the Treetop gem[[10]](#footnote-4). Treetop is described as "a language for describing languages" on its web site; it allows grammars to be written using Ruby code. My ERB grammar was built using parts of the RHTML grammar found in the Erector framework[[11]](#footnote-5) as well as parts of the complex\_html grammar[[12]](#footnote-6) provided as a sample for Treetop use.

Based on the code within an ERB tag, the component expression for the page may change. Consider the following ERB:

Some text

<% if condition %>

<p>Some optional text</p>

<% end %>

<strong>Final text</strong>

In this example, the atomic sections would be as follows:

|  |  |
| --- | --- |
| Atomic Section | Contents |
| p1 | Some text |
| p2 | <p>Some optional text</p> |
| p3 | <strong>Final text</strong> |

The two ERB tags do not belong to any atomic section because they will not be displayed when this Rails page is rendered in the browser. Section p2 will only show up some of the time, since it is within the body of a Ruby if expression. We must therefore treat p2 differently than if it were within a Ruby while loop, or within a Rails form\_tag block. The component expression for the above ERB would be p1.(p2|NULL).p3, meaning section p1 will appear in the browser, followed by either p2 or nothing at all, followed by p3.

So to determine component expressions, Ruby code must be parsed in addition to the HTML, XML, JavaScript, etc. that can appear in an ERB file. In order to parse Ruby code, the ruby\_parser gem[[13]](#footnote-7) was used. This gem allows strings of Ruby code to be parsed, and returns an s-expression when the provided Ruby code is valid. The s-expression is then analyzed to determine if the ERB indicates its contained atomic sections will be selected, iterated, or aggregated. Knowledge of both Ruby and the Ruby on Rails framework was necessary to identify when iteration would occur, for example. Built-in Ruby loop constructs such as for and while had to be identified, as well as methods such as each, each\_with\_index, downto

# Sample Output from the ASM Tool

# Conclusions and Future WorkBibliography

1. Wikipedia contributors, "ERuby," *Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/w/index.php?title=ERuby&oldid=417473164 (accessed March 17, 2011). [↑](#endnote-ref-2)
2. <http://rubygems.org/> [↑](#footnote-ref-2)
3. Wikipedia contributors, "HTML," *Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/w/index.php?title=HTML&oldid=419043476 (accessed March 17, 2011). [↑](#endnote-ref-3)
4. Wikipedia contributors, "Ruby on Rails," *Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/w/index.php?title=Ruby\_on\_Rails&oldid=418130358 (accessed March 16, 2011). [↑](#endnote-ref-4)
5. Wikipedia contributors, "S-expression," *Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/w/index.php?title=S-expression&oldid=418259285 (accessed March 17, 2011). [↑](#endnote-ref-5)
6. Ryan Davis, "parsetree's ruby\_parser-2.0.6 Documentation," *parse\_tree & ruby\_parser*, http://parsetree.rubyforge.org/ruby\_parser/ (accessed March 17, 2011). [↑](#endnote-ref-6)
7. Offutt, Jeff and Wu, Ye, “Modeling Presentation Layers of Web Applications for Testing”, Springer’s Software and Modeling, DOI: 10.1007/s10270-009-0125-4. [↑](#endnote-ref-7)
8. Qian, Miao, and Zeng, “A Practical Web Testing Model for Web Application Testing”, Third International IEEE Conference on Signal-Image Technologies and Internet-Based System, DOI 10.1109/SITIS.2007.16. [↑](#endnote-ref-8)
9. <http://rubyonrails.org/applications> [↑](#footnote-ref-3)
10. <http://treetop.rubyforge.org/> [↑](#footnote-ref-4)
11. <http://erector.rubyforge.org/> [↑](#footnote-ref-5)
12. <http://github.com/threedaymonk/treetop-example/blob/master/complex_html.treetop> [↑](#footnote-ref-6)
13. <http://rubyforge.org/projects/parsetree/> [↑](#footnote-ref-7)