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](#_Toc288227584)

[Definitions 4](#_Toc288227585)

[Introduction 6](#_Toc288227586)

[Implementing the Qian, Miao, and Zeng Model 7](#_Toc288227587)

[Sample Output from the QMZ Tool 10](#_Toc288227588)

[Implementing the Atomic Section Model 12](#_Toc288227589)

[Sample Output from the ASM Tool 17](#_Toc288227590)

[Single File Output 17](#_Toc288227591)

[Rails Application Output 19](#_Toc288227592)

[Future Work 20](#_Toc288227593)

[QMZ Tool 20](#_Toc288227594)

[ASM Tool 20](#_Toc288227595)

[Conclusions 21](#_Toc288227596)

[Bibliography 22](#_Toc288227597)

# Definitions

* **AJAX** -
* **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) [6]. 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) [7], which represents “the relationship among pages of a web application” [7]. A Page Test Tree (PTT) [7], 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[[10]](#footnote-4) 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 [7]. 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 [7] 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" [7].

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

For the web site <http://babyhealthlexington.org/>, the following output was produced:

>ruby scraper.rb -u "http://babyhealthlexington.org/"

Got URI http://babyhealthlexington.org/

Getting pages for site at http://babyhealthlexington.org/...

Got 8 new Pages linked from Page / (13 links)

.Got 2 new Pages linked from Page /wp-admin/ (3 links)

...Got error '405 Method Not Allowed' trying to open URI http://babyhealthlexington.org/wp-comments-post.php, skipping...

Got 3 new Pages linked from Page /2009/01/21/hello-world/ (14 links)

..Got 1 new Page linked from Page /2009/01/21/hello-world/comment-page-1/ (14 links)

.Got 6 new Pages linked from Page /es/2009/01/21/hello-world/comment-page-1/ (14 links)

.............Got 21 pages for site at http://babyhealthlexington.org/

Pages in site rooted at http://babyhealthlexington.org/:

/

/wp-admin/

/2009/01/21/hello-world/

/category/updates/

/es/

/links/

/people/

/mission-statement/

/history/

/wp-login.php

/wp-login.php?action=lostpassword

/2009/01/21/hello-world/trackback/

/2009/01/21/hello-world/comment-page-1/

/es/2009/01/21/hello-world/

/es/2009/01/21/hello-world/comment-page-1/

/es/2009/01/21/hello-world/trackback/

/es/category/updates/

/es/links/

/es/people/

/es/mission-statement/

/es/history/

Getting PFD for site http://babyhealthlexington.org/...

Converting PFD to PTT...

Writing HTML file with test paths to http.babyhealthlexington.org/index.html...

File successfully written

The HTML file of test paths that was created looks like this:

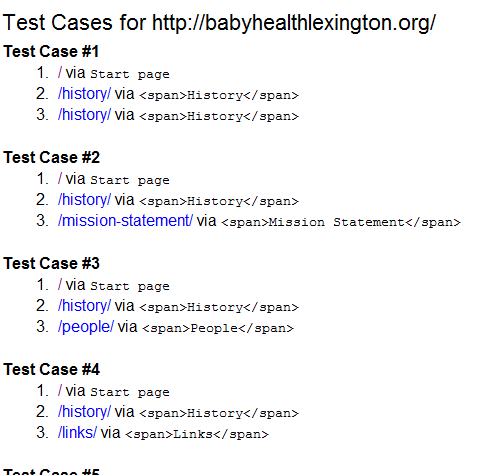


Figure Screenshot of HTML page produced by QMZ tool

In total, 185 test paths were produced for this web site with 21 pages. It is left to the user to determine which test paths to develop into full test cases (deciding input for forms encountered on the test path, which browsers to use, etc.).

# 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. A list of CIMs for the pages in the Ruby on Rails web application is the result of my ASM tool.

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.

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[[11]](#footnote-5). 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[[12]](#footnote-6) as well as parts of the complex\_html grammar[[13]](#footnote-7) 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[[14]](#footnote-8) 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, and downto. One shortcoming is that my tool cannot detect when iteration will occur within a user-defined method. Consider the following Ruby code:

<%

def looper

1.upto(3) { |i| yield i }

end

looper do |j| %>

This is iteration <%= j %>

<% end %>

Here, the s-expression for the looper do |j| block does not indicate that the call to the looper method iterates. The component expression for this ERB comes back as p1, and not p1\*. It is not sufficient to simply treat all blocks as iterations since, for example, the Rails method form\_tag[[15]](#footnote-9) takes a block but does not iterate.

The two main scripts for the ASM tool are generator.rb and single\_file\_generator.rb. The former expects as input the path to the root directory of a Ruby on Rails application, while the latter expects the path to a single ERB file. They also each take a URI to the root page of the web application, if the application is hosted online somewhere, so that a guess can be made as to the URI to the page, based on known Rails URI patterns.

The single\_file\_generator.rb script first reads the file at the given path and passes it to the Parser class. This class uses the Treetop ERB grammar to attempt to parse the given ERB code. It also performs much of the cleanup necessary to construct a CIM for the file, such as identifying atomic sections and nesting code. Parser constructs a class ERBGrammarParser by making a call to Treetop.load and providing the Treetop ERB grammar file. This ERBGrammarParser class is used to parse the provided ERB code, and it returns the root node of the results of its parse: an instance of my ERBDocument class. ERBDocument inherits from Treetop's Treetop::Runtime::SyntaxNode class, as do the other node types represented in my grammar. Most of the work of my ASM tool is done within these classes. Because Ruby has open classes [[[16]](#endnote-9)], I was also able to add extra methods and properties to the Treetop::Runtime::SyntaxNode for use in inherited classes.

It is necessary to identify blocks of code so as to know when an atomic section is within, for example, an if block. Lines of ERB code must be parsed to see if they form valid Ruby code when concatenated. The RubyParser class will raise an exception when invalid code is passed, so it is possible to successively parse lines of Ruby code found in ERB tags to determine when a full block has been found. Then, any code found between the first line of the block and the last line of the block are considered the children of that block. Consider the following:

<% 1 + 1 %>

<p>Text</p>

<% while condition %>

<%= getValue() %>

<strong>Sample</strong>

<% end %>

In this example, the first ERB tag containing Ruby code 1 + 1 would be valid code by itself. It is a single line so it has no children. In this case, my tool would move on to the next ERB tag, while condition. When this code is parsed, RubyParser would raise an exception because this by itself is not valid Ruby. When the next ERB tag's content is appended, however, the while condition with end are valid Ruby. Then my tool knows that the ERB output tag getValue() and the HTML <strong>Sample</strong> are nested within that while loop.

Some code blocks contain others, such as in the following example:

|  |  |
| --- | --- |
| 1 | <% if outer\_condition %> |
| 2 | <% for i in 1..5 do %> |
| 3 | Test |
| 4 | <% end %> |
| 5 | <% else %> |
| 6 | Value |
| 7 | <% end %> |

Here my tool would have to parse all ERB tags concatenated together before RubyParser would accept the input as valid code. After that, the contents of the code block would be lines 2 through 6, inclusive. That is, the RubyParser does not identify lines 2-4, inclusive as being in the true condition of an if statement, nor does it identify line 6 as being in the false condition of an if statement. This must be handled later by my tool via the split\_branches method.

After code blocks have been found, atomic sections can be identified. This is done by iterating over all the nodes in the ERBDocument and appending content to an instance of the AtomicSection class. When an ERB tag is encountered, or when a call to the Rails method render is encountered within an ERB output tag, a new atomic section is started. A new section must be started for a render because that is an instance of aggregation, where the contents returned by render will be embedded in the page. Atomic sections are retained within a list in ERB nodes such as the ERBDocument and ERBTag classes.

After atomic sections are identified, they are nested within code blocks. An index is used to indicate where in the original ERB document a node was found, and these indices are used to determine the parent of an atomic section. An atomic section's range is the range of indices of its contents. Atomic sections are identified at the top-most level, the ERBDocument, and then moved to be contained in the separate atomic section lists that ERBTag nodes maintain. This is useful later when determining component expressions, to see if an atomic section is within, for example, an ERB tag with a loop. If an atomic section's range is indices 3-5, for example, and an ERBTag has range 2-6, the atomic section will be nested within that ERBTag.

Once atomic sections have been nested, if-else branches can be split in ERBTag nodes. This is necessary for identifying selections in component expressions. For each node that looks like a selection (for example, its s-expression has an if or a case statement), the split\_branch method is called on it. This method determines which of the node's contents belongs in which branch by comparing s-expressions. **FINISH THIS**

After branches have been split, transitions can be identified. This is done using the same code that the QMZ model uses, i.e., with Nokogiri. There are different node classes in the ASM model that can have transitions, such as an ERBTag with a form\_tag call or an HTMLOpenTag with a hyperlink. There are several modules in my ASM tool that contain code to be shared between classes, including instance methods and class methods. This uses the mixin feature[[17]](#footnote-10) of Ruby to share members without multiple inheritance (Ruby uses single inheritance). Since multiple classes need access to the HTML parsing methods that use Nokogiri, these are placed in a SharedHTMLParsing module. They could not be placed in a base class because the node classes in the ASM tool must inherit from Treetop::Runtime::SyntaxNode.

Node classes that can possibly have transitions have a get\_local\_transitions method that differs by class. Each implementation takes a source parameter, though, intended to contain the URI of the page where the hyperlink or form transition is located. It is the job of the get\_local\_transitions method to identify any transitions within its own code, which the Treetop::Runtime::SyntaxNode retains, and return Transition instances to represent these. The Transition class has three children: FormTransition, RedirectTransition, and LinkTransition. These three classes correspond to the form link transition, redirect transition, and simple link transition in the Atomic Section Model. A RedirectTransition is created when, for example, an ERBTag has a call to the Rails method redirect\_to, which would cause the browser to redirect to another page.

The get\_local\_transitions method defined in ERBTag looks for form and redirect transitions by looking at the s-expression of the code of that ERBTag. It does not need to look for link transitions because even if a Rails method such as link\_to were used in an ERB tag to produce a hyperlink, the result of that code would not be included in the final web page because it is in an ERB tag and not an ERB output tag.

To find local form transitions in an ERBTag, the names of known <form>-producing methods are iterated over, and the s-expression is checked for each to see if a call is made to one of them. One improvement that could be made in future work is to move this list of methods to an external file that could be read at runtime. This would allow greater flexibility in the event that Rails changes or adds methods that affect transition generation. If a call is detected within the s-expression, the arguments to the call are extracted from the s-expression and used to determine the destination of the transition. For a form transition, the destination is the contents of the action attribute in HTML. The s-expression is checked for calls to the Rails method url\_for, as well as the assignment of values to keys controller and action in a hash. Once the destination of the form transition has been determined, a new instance of the FormTransition class can be constructed.

Other transitions are identified differently based on the node class in which they are identified. For example, the HTMLOpenTag class uses Nokogiri to parse its HTML source to find link and form transitions. The ERBOutputTag class checks the s-expression for its Ruby source code to find calls to Rails methods such as link\_to and link\_to\_unless\_current.

Once transitions have been identified, a new instance of the ComponentInteractionModel class can be constructed with the atomic sections, component expression, and list of transitions for that ERB file. The final produce of my ASM tool is a single CIM or a list of CIMs, depending on whether the single\_file\_generator.rb or the generator.rb script was used.

# Sample Output from the ASM Tool

## Single File Output

Consider the following ERB:

<div id="game\_area">

<% #Write out a hidden AJAX (remote) form to submit cards.

form\_remote\_tag(:url => url\_for(:action => "handle\_card\_submit"), :html => {:id => 'cardform'}) do %>

<%= hidden\_field\_tag 'form', 'card\_submission' %>

<% #Write out the cards for the user to see, along with hidden checkboxes.

@cards.each\_with\_index do |card, count| %>

<%= display\_card(card, session[:user][:color\_blind]) %>

<%= check\_box\_tag('cards[]', card.id, false, :id => "card#{card.id}", :class => 'display\_none') %>

<%

end

end %>

</div><!--game\_area-->

This ERB will produce an HTML form tag that has an AJAX call to the handle\_card\_submit action (meaning the handle\_card\_submit() method in the current Rails controller class, which equates to the handle\_card\_submit/ page). Within that form, there is a hidden form tag as well as a loop over the @cards array. Within the loop, there are two ERB output tags. Wrapped around the entire form is an HTML div tag. Using the single\_file\_generator.rb script, here is the output:

>ruby single\_file\_generator.rb app\vie

ws\test\cards.html.erb "http://example.com"

Parsing ERB file app\views\test\cards.html.erb...

Source file: app\views\test\cards.html.erb

0: Atomic Section #1 (indices 0..0):

0: div id => "game\_area"

1: #Write out a hidden AJAX (remote) form to submit cards.

2-9: form\_remote\_tag(:url => url\_for(:action => "handle\_card\_submit"), :html

=> {:id => 'cardform'}) do

Content and sections:

3: Atomic Section #2 (indices 3..3):

3: <%= hidden\_field\_tag 'form', 'card\_submission'

4: #Write out the cards for the user to see, along with hidden checkboxes.

5-8: @cards.each\_with\_index do |card, count|

Content and sections:

6: Atomic Section #3 (indices 6..7):

6: <%= display\_card(card, session[:user][:color\_blind])

7: <%= check\_box\_tag('cards[]', card.id, false, :id => "card#{card.id}

", :class => 'display\_none')

Close:

8: end

Close:

9: end

10: Atomic Section #4 (indices 10..11):

10: /div

11: <!--game\_area-->

Component Interaction Model

Start page: app\views\test\cards.html.erb

Start URL: http://example.com/test/cards

Component expression: p1.p2.p3\*.p4

Transitions:

Form Transition

</test/cards> --> </test/handle\_card\_submit>

Underlying code: form\_remote\_tag(:url => url\_for(:ac

tion => "handle\_card\_submit"), :html => {:id => 'cardform'}) do

The single\_file\_generator.rb script shows its parse of the input ERB, which is useful for debugging. Even though the opening div, the results of the form\_remote\_tag call, and the hidden\_field\_tag call will all be displayed together in the resulting HTML when this page is viewed in the browser, the tool does not know that, and so treats the form\_remote\_tag ERB tag as a separator between two atomic sections. It would be possible to maintain a list of Rails methods that do not result in conditional or iterative output, such as form\_remote\_tag, and thus modify the tool to realize that a form\_remote\_tag ERB tag does not immediately signify the start of a new atomic section, but that was outside the scope of this project.

The ERB code above also shows another intricacy of the Ruby on Rails framework: some ERB tags, when interpreted by Rails, can produce output. Namely, the form\_remote\_tag method produces an HTML form tag with different attributes depending on the parameters[[18]](#footnote-11). My tool treats all ERB tags as non-output producing, since generally this is the case, though it does scan ERB tags for form transitions, as evidenced in the output above that shows a single form transition was found in the ERB.

## Rails Application Output

The following is the result produced by running the generator.rb script against the root directory of a Ruby on Rails application:

**FINISH THIS**

# Future Work

## QMZ Tool

The QMZ tool can be expanded to take credentials and other parameters for forms, so that when it encounters a form in a page it has scraped, it can populate the form and submit it. This would allow the tool to traverse the web site as much as possible when unauthenticated, for example, and then traverse the web site again after it has submitted a login form and authenticated. The PTT could be modified such that a node indicates whether any form on the page was submitted first, so that one branch of the PTT might be the unauthenticated traversal of the web site, and another branch might be the authenticated traversal. This would greatly increase the usefulness of the QMZ tool since so many web applications require some form of authentication to access the full contents. If the particular forms and their desired input were known in advance, these could be provided to the QMZ tool initially, and it could submit the appropriate parameters to the specified forms. It might also be possible to make the QMZ tool interactive such that if it found a form within a scraped page, it would prompt the user to get parameters. Form submission could be handled by the Ruby gem Mechanize[[19]](#footnote-12) or something similar. The Mechanize gem can also maintain state via cookies, which many web sites use to differentiate an authenticated user from an unauthenticated user.

The QMZ might also be modifiable to automate part of the testing process. Using a Ruby gem like Selenium[[20]](#footnote-13), it would be possible to take screenshots of web pages. Once the PTT is constructed, test paths could be followed and a screenshot could be taken of each successive page in the path. These screenshots could then be presented to the user in the HTML page of test paths, alongside the associated URI in the test path. It would be necessary to take a different screenshot for each URI even if the same URI appeared in several test paths, since the page might change depending on the referring page, or on the transition used to reach the page. While having a screenshot alone might not be sufficient for all test cases, it would be helpful when the test purpose is to ensure correct appearance and content.

The combination of screenshot and form support in the QMZ tool could be very helpful in terms of automating test execution. A user would still be necessary to manually examine each screenshot, and to provide form parameters. The tool would go through the steps of entering parameters into forms, traveling to each page in the test path, and taking screenshots along the way of the results of following each transition.

## ASM Tool

The ASM tool can be modified to combine a collection of CIMs into an Application Transition Graph for the entire web application. Once the ATG is constructed, different coverage methods could be applied to generate test paths that traverse the entire web application, such as prime path coverage [6] or invalid path coverage [6]. These test paths could be presented in the same way as the QMZ model's test paths, and could even be merged and displayed in a single page.

Further work could be done to the ASM tool to make it more flexible to changes in the Ruby on Rails framework, such as pulling lists of key methods out of node classes and into external files. A YAML file would be easily parsable in Ruby and could contain method names know for generating link, form, and redirect transitions. The different ERB node classes could refer to this YAML file when identifying transitions.

A combination of the QMZ and ASM tools would be helpful. Once an ATG is constructed in the ASM tool and test paths are found through some form of path coverage, a separate tool could be used to combine test paths from the QMZ and ASM tools. This new tool could actually follow the test paths, using gems like the aforementioned Selenium and Mechanize to take screenshots and submit forms. An HTML page of results could be presented to the user, showing the test paths that were followed and screenshots of each page in each test path. The parameters submitted to each form could also be displayed. It might be useful to allow multiple sets of parameters to be used for a single form, in order to simulate different test cases.

# Conclusions

Through the QMZ tool, it is possible to find paths a user might take through your application via the transitions you have provided. Through the ASM tool, it is possible to generate artifacts which can be used to find paths that show engage every possible chunk of code that could be output to the user in their browser. The combination of these two tools can provide a developer with a good overview of what to test. Previously, applying these models involved much manual generation of artifacts. For the manual application of the QMZ model, links and forms needed to be manually found in each web page within the web application. Then, a tree had to be constructed of these transition elements, and then traced to find possible test paths. To manually generate CIMs for the Atomic Section Model for a Ruby on Rails application, each view file in the source code had to be analyzed manually to identify atomic sections. Once atomic sections were labeled, one would have to manually determine the component expression of that page, as well as a list of transitions found within the page.

My contribution is these tools that remove much of the manual work from finding good test paths for a web application. These tools will be released under an open source license to the public, so others may use them freely to help test their code. It is desirable to have correctly functioning web applications since accomplishing even daily tasks is depending more and more on the Internet.

# Bibliography

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://nokogiri.org/> [↑](#footnote-ref-4)
11. <http://treetop.rubyforge.org/> [↑](#footnote-ref-5)
12. <http://erector.rubyforge.org/> [↑](#footnote-ref-6)
13. <http://github.com/threedaymonk/treetop-example/blob/master/complex_html.treetop> [↑](#footnote-ref-7)
14. <http://rubyforge.org/projects/parsetree/> [↑](#footnote-ref-8)
15. <http://api.rubyonrails.org/classes/ActionView/Helpers/FormTagHelper.html#method-i-form_tag> [↑](#footnote-ref-9)
16. Ruby contributors, "Ruby from Other Languages", *Ruby A Programmer's Best Friend,* http://www.ruby-lang.org/en/documentation/ruby-from-other-languages/ (accessed March 17, 2011). [↑](#endnote-ref-9)
17. <http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut_modules.html> [↑](#footnote-ref-10)
18. <http://apidock.com/rails/ActionView/Helpers/PrototypeHelper/form_remote_tag> [↑](#footnote-ref-11)
19. <http://mechanize.rubyforge.org/mechanize/> [↑](#footnote-ref-12)
20. <http://selenium.rubyforge.org/> [↑](#footnote-ref-13)