Applying Mutation Testing to Web Applications

Conference Paper · April 2010

DOI: 10.1109/ICSTW.2010.38 · Source: DBLP

Upsorn Praphamontripong
University of Virginia

Jeff Offutt
George Mason University

Applying Mutation Testing to Web Applications

Upsorn Praphamontripong and Jeff Offutt
Software Engineering
George Mason University, Fairfax, VA, USA
upraphamr@gmu.edu, offutt@gmu.edu

Abstract

As our awareness of the complexities inherent in web applications grows, we find an increasing need for more sophisticated ways to test them. Many web application faults are a result of how web software components interact; sometimes client-server and sometimes server-server. This paper presents a novel solution to the problem of integration testing of web applications by using mutation analysis. New mutation operators are defined, a tool (webMuJava) that implements these operators is presented, and results from a case study applying the tool to test a small web application are presented. The results show that mutation analysis can help create tests that are effective at finding web application faults, as well as indicating several directions for improvement.

1 Introduction

Web applications are user interactive software applications that can be accessed through web browsers. They are typically developed by teams with diverse expertise that integrate diverse frameworks and web components [20]. Web components are software components that can be tested independently and interact with each other to provide services as part of web applications. Web components are written in different languages, including Java Servlets, Java Server Pages (JSPs), JavaScripts, Active Server Pages (ASPs), PHP, and AJAX (Asynchronous JavaScript and XML). Web components may reside in different locations and be integrated dynamically. Web applications are also heterogeneous collections of distributed and dynamically integrated web components. The appearance, user interface and functionality of web applications may vary by users, time, and geography. Web applications are also accessed by massive numbers of users with different hardware and web browsers [12].

Society suffers many losses due to web application failures. In October 2004, PayPal had to waive all customers' transaction fees for an entire day because of a service outage after upgrading the site [10]. The service unavailability may have been due to integration errors [24]. In August 2006, Amazon.com disconnected its web site for two hours because of software problems, losing millions of dollars [28]. In July 2008, Amazon's S3 systems web components hosted storage service failed, causing businesses that rely on this service to lose information and revenue [4]. The impact of faulty web applications may range from inconvenience (i.e., malfunction of the application or users' dissatisfaction), to economic loss (i.e., interruption of business), and to catastrophic loss (i.e., loss of life due to failures of medical web applications). This research is developing new ways to test integration aspects of web software components. Web applications integrate components that are on multiple hardware/software platforms, written in different languages, and do not share the same memory space. Execution is based on requests from clients to multiple components on severs, where each request creates a new thread on an independent object. The software manages state and control flow in novel ways that are unique to web applications. Although powerful, these abilities lead to new kinds of faults that must be tested with new techniques.

Program mutation testing [5, 1] takes a syntactic structure (the program), creates slightly modified versions of the program, then helps the tester find test inputs that will cause the modified programs to behave differently from the original. The modified versions are created with *mutation operators*, which encode rules that help ensure the software is well tested. The modified versions are called *mutants*, and if a test causes the mutant to behave differently from the original version, the test is said to *kill* the *mutant*.

This research presents a collection of new mutation operators specifically designed to test interactions among web components. Several Java-based mutation operators have been proposed in the literature [3, 16, 17, 21]. However, these do not target web-specific issues. Therefore, additional mutation operators to address web-specific faults have the potential to improve our ability to test web applications.

The web mutation operators in this paper are defined on JSPs and Java Servlets. The web mutation operators have been implemented in a tool that is based on muJava [18]. webMuJava automatically generates mutants, accepts tests, and runs the mutants against the tests to report results. Test cases are created manually as sequences of requests and can be written in HttpUnit, HtmlUnit, and Java. Test evaluation is based on responses from the server and checked automatically.

This paper is organized as follows. Section 2 describes atomic section modeling of web applications. Section 3 summarizes mutation testing and extensions of mutation operators that specifically deal with the features of web applications. Section 4 describes a tool and small empirical validation of the mutation operators. Results are discussed in Section 5. Section 6 provides an overview of related research and Section 7 concludes with further research.

2 Modeling Web Applications

Previous mutation systems have used three general types of operators: (1) operators that imitate faults that programmers make, such as replacing one scalar variable with another; (2) operators that force good tests, such as failing only if an expression has the value zero; and (3) operators that imitate uncommon faults, such as changes to a logic predicate that can only be killed by very powerful tests. Thus, we started by identifying potential web application faults.

Several papers have attempted to classify faults in web applications. Guo and Sampath categorized faults based on functionality of the statements where the faults reside; database manipulation, control flow and business logic, form parameter management, web page appearance, and redirection [8]. Marchetto, Ricca and Tonella consider faults related to browser incompatibility, faults related to the needed plugins, faults related to form construction, faults related to database interactions or management, faults related to the web pages integrations, faults due to the unavailable resources, and faults related to user authentication [19].

These categorizations overlap without being complete or consistent. In the absence of a widely accepted fault model, this research uses a **structural** approach. The mutation operators in this paper are designed to test control and state connections among web software components as defined in the atomic section model [22].

Atomic sections are defined on the presentation layer of server software, the components that produce HTML response pages [22]. An *atomic section* (*ATS*) is a piece of HTML such that if one piece of the HTML is sent to the client, then all the HTML in that ATS is sent. An ATS may be pure static HTML, or empty, or an HTML section created by a program component that contains static structure with content variables.

Atomic sections are combined using algebraic rules to form *component expressions*. Let p and p_i be component expressions. Component expressions are defined as follows. **Basis**: A component expression p itself is an atomic section. **Sequence** ($p \rightarrow p_1 \cdot p_2$): A component expression p is composed of a component expression p_1 followed by a component expression p_2. **Selection** ($p \rightarrow p_1 | p_2$): A component expression p is obtained by selecting either a component expression p_1 or a component expression p_2, but not both. **Iteration** ($p \rightarrow {p_1}^*$): A component expression p consists of an arbitrary length sequence of a component expression p_1. **Aggregation** ($p \rightarrow p_1\{p_2\}$): A component expression p is comprised of a component expression p_1, in which a component expression p_2 is included as part of a component p_1 when p_1 is sent to the client.

We consider five types of transitions among web components. A **Simple Link Transition** is an <A> link in an atomic section that defines a transition from the client to a web component on the server. If there is more than one <A> link in an atomic section, one of several web components can be invoked. A **Form Link Transition** submits an HTML <FORM>, which causes a transition from the client to a web component on the server, along with whatever data is included in the form. A **Component Expression Transition** is when the execution of a web component causes an atomic section (i.e., a component expression) to be generated and returned to the client. A web component can produce several component expressions. An **Operational Transition** is a transition taken by the user or the environment outside of the control of the software. Examples of operational transitions are the user pressing the back button, the forward button, the refresh button, or directly altering the URL in the browser. Reloads from cache are also operational transitions. Finally, a **Redirect Transition** causes the client to regenerate the same request to a different URL.

The fault analysis for this work is based on the potential faults that can occur in the above transitions. These faults are used to design the web mutation operators that are defined in Section 3.

Faults in simple link transitions: The `href` attribute of an <A> tag specifies the destination of a web resource in the form of a Uniform Resource Identifier (URL). A common mistake is a web application developer using an incorrect URL. Another common mistake is that an incorrect destination could lead to a loop.

Faults in form link transitions: The HTML <`form`> tag defines the server-side form handler as a URL in the <`action`> attribute. Four common mistakes in form link transitions are considered. First, faults may be due to an unavailable or improper destination. Faults may occur when the developer uses an incorrect URL, and the handler specified cannot process the request. This is an easy mistake to make, especially when URLs are generated dynamically on the server.

Second, faults may occur when the wrong HTTP transfer mode (GET, POST, etc.) is given. Using the wrong transfer mode can result in different behavior of web applications or reveal confidential information.

Third, faults may occur when necessary information is omitted or inappropriate information is submitted via hidden controls. Hidden form fields allow web applications to place data in HTML that will be submitted in the next request. These elements are not rendered by the browser, but users can see them by viewing the source, and also save and modify the source. If users omit or change hidden input data, the request can be invalid or allow the users to gain unauthorized access.

Fourth, faults may be due to parameter mismatches. For instance, a form and a web component that submits the form may refer to the same argument with different names, or a web component may submit a form with too many arguments. Parameter mismatches can cause a web component to behave unexpectedly or return improper results.

Faults in component expression transitions: The contents of HTML, text, or JSP files can be reused or included into another HTML or JSP file dynamically using Server-Side Include (SSI)–a server-side scripting language. An `include` directive generates an atomic section and returns it to the client. Specifying an `include` directive destination incorrectly may cause an erroneous response.

Faults in operational transitions: The intended transitions of the web application's control may be altered when the user presses the back button, the forward button, or the refresh button They can also be altered when the user directly modifies the URL. Potential faults include when the user accidentally presses the button or when the user intentionally bypasses the input validation. This kind of fault is, in general, due to the user's behavior and hence is out of scope of this project.

Faults in redirect transitions: HTML allows redirect transitions by using an `HTTP-EQUIV` attribute and specifying a forwarded destination using a `URL` attribute of a <`META`> element. Java Servlets implement redirect transitions with the `res.SendRedirect (destination)` command and JSPs use the <`jsp:forward`> command. Faults can occur when the developers specify incorrect destinations. As a result, the request may be processed inaccurately or may not be processed at all.

3 Mutation Testing for Web Applications

The effectiveness of mutation testing depends largely on the mutation operators [16]. The muJava system has traditional mutation operators that modify individual statements, and object-oriented (OO) class level operators that modify OO language features.

This paper presents new source-code, first-order, mutation operators for web software components. They test the connections defined in Section 2 by modifying transitions and atomic sections. We define 11 web application mutation operators, grouped into mutation operators that modify HTML and mutation operators that modify JSP.

By convention, the mutation operator names start with "W", indicating mutation operators dealing with web-specific features, and end with a "D" or "R", indicating whether the operators delete or replace something. The middle character specifies what language element is changed.

3.1 Mutation Operators for HTML

Simple link replacement (WLR): The WLR operator replaces a destination of a simple link transition specified in an <A> tag with another destination in the same domain of the targeted web application. This alteration mimics a common mistake that web application developers make. The changes cause references to incorrect or nonexistent URLs, and may also lead to dead code. This mutant can be killed by a test case that causes the reference to the URL to be incorrect.

Simple link deletion (WLD): The WLD operator removes a destination of a simple link transition specified in an <A> tag. This modification breaks the original control flow.

Form link replacement (WFR): The WFR operator changes a destination of a form link transition to another destination in the same domain of the targeted web applications. Similar to WLR, the changes cause references to an incorrect or nonexistent URL. Furthermore, the mutated destination may not be able to process the request. This mutant can be killed by a test case that causes the reference to the URL to result in different behavior.

Transfer mode replacement (WTR): The WTR operator replaces all `GET` requests with `POST` requests and all `POST` requests with `GET` requests. The WTR operator guides testers to generate test inputs to ensure that transfer modes are specified appropriately.

Hidden form field replacement (WHR): The WHR operator alters `value` attributes of an `<input>` tag of type `hidden` with another value; for instance, `null`, a space, an empty string, a zero, and a negative integer (such as -1).

Hidden form field deletion (WHD): The WHD operator removes an entire block of an `<input>` tag of type `hidden`. WHD mutants guide testers to generate test inputs to ensure that data submitted to the server are properly handled.

Server-side-include replacement (WIR): The WIR operator changes `file` attributes of `include` directives to another destination in the same domain of the targeted web application. The variations distort part of the original presentation of the web page.

Server-side-include deletion (WID): The WID operator removes an entire `include` directive from the HTML file. Thus, a portion of the original presentation of the web page is distorted. This mutant can only be killed by a test case that shows the presentation of the web page to be incorrect.

3.2 Mutation Operators for JSP

Redirect transition replacement (WRR): The WRR changes a forwarded destination of a redirect transition specified in `<jsp:forward>` to another destination; an incorrect destination or a URL in the same domain application. The change causes references to an incorrect or nonexistent URL. Additionally, to help testers ensure that possible dead code or an infinite loop is handled properly, a destination may be replaced by the targeted web application itself. This mutant can only be killed by a test case that shows the reference to the URL is incorrect.

Redirect transition deletion (WRD): The WRD operator removes an entire redirection, as specified in a `<jsp:forward>` tag. This operator helps testers ensure that control flow in the web application is implemented correctly.

Get session replacement (WGR): A new connection to the web server is opened every time a client retrieves a web page. Since the server does not automatically maintain information about the client, the user session must be properly stored and tracked by server software. To support session tracking, web software development frameworks provide in-memory objects and methods to access those objects. This is called the *session object* in J2EE. When a reference to a session object is requested, if the session object does not exist it can be created (parameter `true`, or if it does not exist it will not be created (parameter `false`). The WGR operator changes `true` to `false` and `false` to `true`.

4 Case Study

An important goal of this project is to evaluate the use of mutation to test web applications. The web mutation operators in Section 3 were implemented in a tool, webMuJava, based on muJava [18]. For this case study, we hand-seeded faults into a moderate size web application, generated tests to kill the mutants, then evaluated the tests on their ability to find the seeded faults.

4.1 Subject Web Application

For an initial trial, we used a web application that is small enough for reasonable hand analysis, yet large and complex enough to include a variety of interactions among web components. The mutation operators are based on atomic sections and transitions. The paper that defined the atomic sections [22] used a web application called the "Small Text Information System (STIS)," so this study also used STIS. STIS consists of 18 Java Server Pages (JSPs) and 5 Java bean classes and stores information in a MySQL database. Only the JSPs were mutated, and the study excluded two JSPs that provide administration features, leaving a total of 16.

It is customary to give a measure of the size of software artifacts that are used in empirical studies. However, size is problematic with scripted page modules such as JSPs, which contain a mix of HTML, Java statements and directives. We found two online tools that measure the size of JSPs, Code Counter Pro[1] and Practiline Source Code Line Counter[2]. These tools count carriage returns, count HTML comments, do not count multi-line HTML statements as one line, count blank lines, and do not consistently eliminate Java comments. Most importantly, they do not separate HTML from Java statements. Since webMuJava mutates Java statements in JSP, the number of Java statements seems to be the most important measure. Our solution for this study was to count by hand, and count Java statements, HTML statements, comments and blank lines separately. These numbers are shown in Table 1. We believe the number of Java lines is the most relevant for this project.

4.2 Empirical Conduct

The case study had four steps. First, we used webMuJava to create mutants. Second, we generated tests by hand and automated them. Third, we seeded faults into a subject program,. Fourth, we ran the tests on the hand-seeded faults.

We extended muJava by implementing the mutation operators defined in Section 3. Accordingly, the architecture of webMuJava is similar to the architecture of muJava with additional web-specific mutation operators. MuJava supports only `.class` files, so additional modules to parse and analyze JSP and HTML were implemented in webMuJava.

To ensure that the mutation operators use proper destinations that are in the same domain of the targeted web application, only one web application was tested at a time. URLs appearing in all selected files were parsed and extracted. The frequency of each URL's use was analyzed. This information was recorded and later used by the operators to mutate the destination of a link or a form. For example, the WLR operator replaces the destination of a simple link transition with the most frequently used URL. If the URL is exactly the file under test itself, the second most frequently used URL is used. If there are multiple URLs with the same frequency, the first URL is used.

Only one URL was used for each destination replacement. For example, the WLR operator alters an original destination to another destination only once. The reason why the WLR operator does not replace the original destination with all possible URLs is that the mutants created are trivial; incorrect URLs will always result in different behavior of the web application. Hence, a selective approach can be used to reduce the number of mutants.

Figure 1 shows the webMuJava screen for generating mutants. The testers can select multiple files to test. Then, the testers can select the web-specific

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

[2] http://sourcecount.com/features.htm

Table 1. Size of the JSP files (in LOC)

JSP file	Java lines	HTML lines	Java/HTML ratio	Comment lines	Blank lines	Total lines
about	0	97	0.00	8	19	124
browse	62	83	0.75	52	41	238
categories	34	49	0.69	37	21	141
category_edit	14	37	0.38	22	13	86
index	0	31	0.00	13	7	51
login	19	32	0.59	22	23	96
logout	10	21	0.48	13	9	53
navigation_bar	3	25	0.12	13	9	50
page_footer	2	4	0.50	6	3	15
page_header	9	7	1.29	9	8	33
record_add	4	45	0.09	22	15	86
record_delete	3	5	0.60	8	4	20
record_edit	36	55	0.65	30	25	146
record_insert	12	46	0.26	23	15	96
record_search	7	41	0.17	14	11	73
update_search	9	3	3.00	6	3	21
Total	**224**	**581**	**0.39**	**298**	**226**	**1329**

mutation operators (or traditional mutants, or class mutants). Once the tester clicks the `Generate` button, webMuJava automatically generates mutants. These mutants can be viewed in the `Web Mutants Viewer` panel. By specifying a target file and a test case, webMuJava automatically executes all mutants. Test evaluation is based on responses from the server and is done automatically.

Test cases were generated by hand as sequences of HTTP requests and were automated in Java and HtmlUnit. HtmlUnit is a test framework that allows the testers to simulate the behavior of the web applications to be tested; for instance, invoking HTML pages, filling out forms, and clicking links.

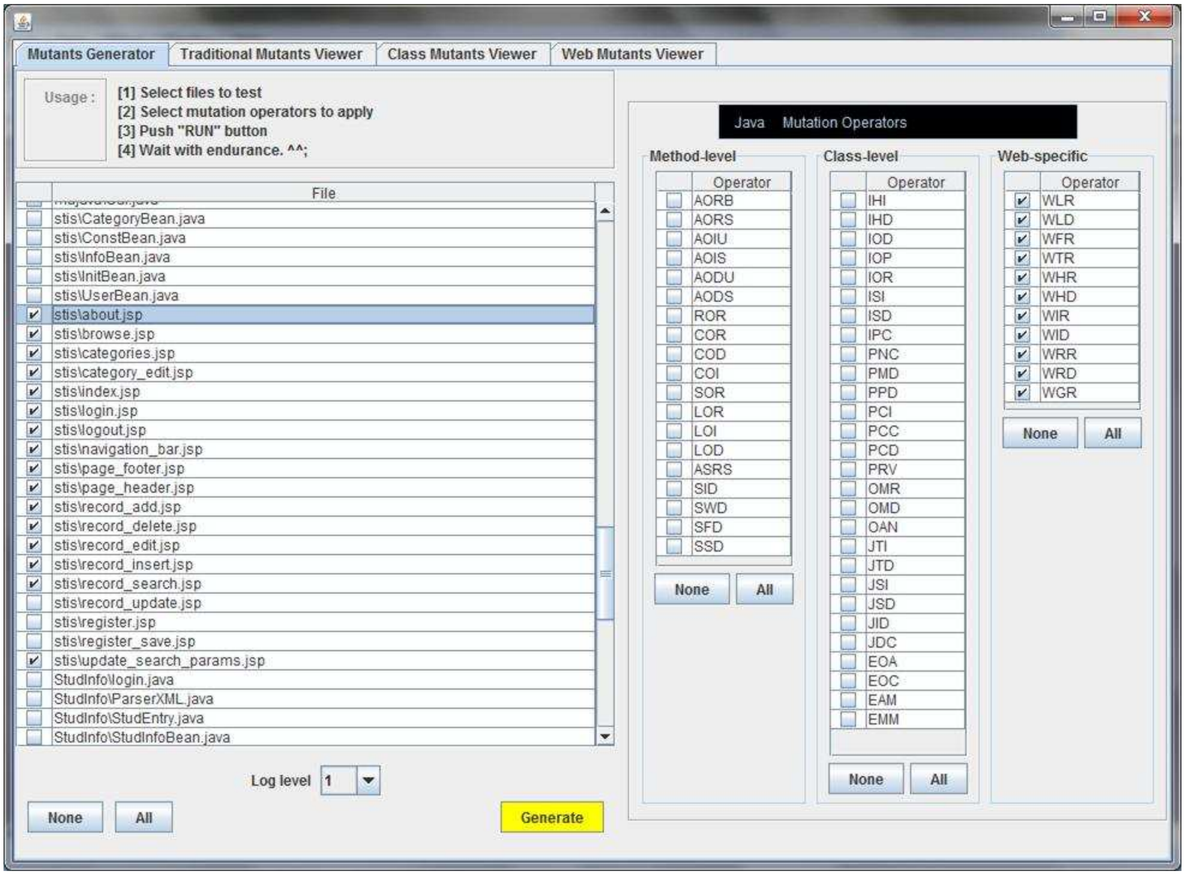
Each mutated file was stored in a directory indexed by its name, its mutation operator, and its mutant number. webMuJava then successively replaces the original file with each mutant. Hence, invocations of the original web application remain the same.

To analyze the potential of our web-specific mutation operators, tests were generated to cover as many mutants as possible and were later used to execute hand-seeded faults. webMuJava compares the appearances of the web page created by the original program to the web page that was created by the mutant. The comparison excludes a destination value of an `<A>` tag or in a form link transition since the transition has not been triggered. The percentage of seeded faults detected indicates how well web-specific mutation operators can be used to generate or improve tests.

Faults were seeded into the subject by hand. To avoid bias, we chose someone who had no prior knowledge of web mutation testing and little knowledge of traditional mutation testing, but who had experience building web applications. We asked the faults to be in the Java statements part of the JSPs, rather than the HTML, but did not give any more specific guidance. The fault seeder created 167 faults in the 16 JSPs. Upon analysis, we found that 20 were functionally equivalent to the original JSPs, so they were excluded, leaving 147.

4.3 Threats to Validity

One potential threat to **internal validity** is that we used hand-seeded faults. One person inserted faults manually based on his experience as a web developer. This person was not the tester and had no knowledge of the test method, however there is no guarantee that they represent real web application faults. For **external validity**, one potential threat is due to the limitation in application domain and representativeness of our subject. Replication of this study on other web applications is needed. A **construct validity** threat is the use of webMuJava, which was assumed to generate mutants and mark them as being killed correctly.



[1] Select files to test
[2] Select mutation operators to apply
[3] Push "RUN" button
[4] Wait with endurance. ^^;

☐ stis\CategoryBean.java
☐ stis\ConstBean.java
☐ stis\InfoBean.java
☐ stis\InitBean.java
☐ stis\UserBean.java
☑ stis\about.jsp
☑ stis\browse.jsp
☑ stis\categories.jsp
☑ stis\category_edit.jsp
☑ stis\index.jsp
☑ stis\login.jsp
☑ stis\logout.jsp
☑ stis\navigation_bar.jsp
☑ stis\page_footer.jsp
☑ stis\page_header.jsp
☑ stis\record_add.jsp
☑ stis\record_delete.jsp
☑ stis\record_edit.jsp
☑ stis\record_insert.jsp
☑ stis\record_search.jsp
☐ stis\record_update.jsp
☐ stis\register.jsp
☐ stis\register_save.jsp
☑ stis\update_search_params.jsp
☐ StudInfo\login.java
☐ StudInfo\ParserXML.java
☐ StudInfo\StudEntry.java
☐ StudInfo\StudInfoBean.java

Method-level

	Operator
☐	AORB
☐	AORS
☐	AOIU
☐	AOIS
☐	AODU
☐	AODS
☐	ROR
☐	COR
☐	COD
☐	COI
☐	SOR
☐	LOR
☐	LOI
☐	LOD
☐	ASRS
☐	SID
☐	SWD
☐	SFD
☐	SSD

Class-level

	Operator
☐	IHI
☐	IHD
☐	IOD
☐	IOP
☐	IOR
☐	ISI
☐	ISD
☐	IPC
☐	PNC
☐	PMD
☐	PPD
☐	PCI
☐	PCC
☐	PCD
☐	PRV
☐	OMR
☐	OMD
☐	OAN
☐	JTI
☐	JTD
☐	JSI
☐	JSD
☐	JID
☐	JDC
☐	EOA
☐	EOC
☐	EAM
☐	EMM

Web-specific

	Operator
☑	WLR
☑	WLD
☑	WFR
☑	WTR
☑	WHR
☑	WHD
☑	WIR
☑	WID
☑	WRR
☑	WRD
☑	WGR

Figure 1. WebMuJava - Screen for generating mutants

5 Results

Table 2 summarizes the results. Manual analysis revealed that 12 of the 41 live mutants were equivalent, leaving 29 non-equivalent undetected mutants. All equivalent mutants were of type WHR and involved changes of values of non-keys of records to be updated to or deleted from the database. For example, when a record is being removed, only a record's name (a primary key) is taken into account. Thus, replacing a value of a parameter `rec_category` from an empty string (`""`) to `null` as `<input type="hidden" name="rec_category" value="">` in browse.jsp has no effect. The WHR operator also generated approximately a third of the mutants, by far the most. WHR changes `value` attributes of hidden inputs to `null`, a space, an empty string, a zero, and a negative integer. Approximately a third of the WHR mutants were not detected. Most live WHR mutants involve changes of values of a hidden input dealing with sorting records; for instance, replacing a value of a parameter `rec_sort` in browse.jsp from `<input type="hidden" name="rec_sort" value="">` to `<input type="hidden" name="rec_sort" value="0">` or to `<input type="hidden" name="rec_sort" value="-1">`.

Table 3 shows the results from running the tests on the hand-seeded faults. The number of hand-seeded faults varies among JSP files (ranging from 0 to 29) and are not equally distributed. Note that record_delete.jsp had no faults.

The faults that were not found fall into several categories. Several relate to changes in scope settings of `jsp:useBean`s. For example, in category_edit.jsp, a `scope` attribute is altered from `session` to `page` as `<jsp:useBean id="iconst" scope="page" class="stis.ConstBean">`.

Several others are due to the changes between an

Table 2. Summary of mutants and tests

JSP file	Mutants											Total	Tests	Live	Killed	Score
	WLR	WLD	WFR	WTR	WHR	WHD	WIR	WID	WRR	WRD	WGR					
about	6	6	0	0	0	0	2	2	0	0	0	16	7	0	16	1.00
browse	5	5	3	4	25	5	4	4	0	0	0	55	13	14	41	0.75
categories	4	4	0	3	20	4	2	2	0	0	0	39	11	6	33	0.85
category_edit	1	1	0	1	5	1	1	1	1	1	0	13	6	0	13	1.00
index	1	1	1	1	0	0	2	2	0	0	0	8	4	0	8	1.00
login	4	4	0	0	0	0	3	3	1	1	1	17	7	0	17	1.00
logout	2	2	0	0	0	0	1	1	0	0	1	7	3	2	5	0.71
navigation_bar	5	5	0	0	0	0	0	0	0	0	0	10	5	0	10	1.00
page_footer	1	1	0	0	0	0	1	1	0	0	0	4	2	0	4	1.00
page_header	0	0	0	0	0	0	1	1	0	0	1	3	2	1	2	0.67
record_add	1	1	1	1	0	0	2	2	0	0	0	8	4	0	8	1.00
record_delete	0	0	0	0	0	0	1	1	1	1	0	4	2	0	4	1.00
record_edit	1	1	0	1	10	2	2	2	1	1	0	21	6	6	15	0.71
record_insert	1	1	0	1	0	0	2	2	1	1	0	9	4	0	9	1.00
record_search	0	0	1	1	0	0	0	0	0	0	0	2	2	0	2	1.00
update_search	0	0	0	0	0	0	0	0	1	1	1	3	2	0	3	1.00
Total	**32**	**32**	**6**	**13**	**60**	**12**	**24**	**24**	**6**	**6**	**4**	**219**	**80**	**29**	**190**	
Live	0	0	0	0	23	3	1	1	0	0	1	**29**				
Killed	32	32	6	13	37	9	23	23	6	6	3	**190**				
Score	1.00	1.00	1.00	1.00	0.62	0.75	0.96	0.96	1.00	1.00	0.75	**0.87**				

`equals` method and an equal sign (==). An example in login.jsp is:

```
if (request.getParameter ("userid").equals("") ||
request.getParameter ("password").equals(""))
```

The statement was altered to:

```
if (request.getParameter ("userid")==("") ||
request.getParameter ("password").equals(""))
```

and

```
if (request.getParameter ("userid").equals("") ||
request.getParameter ("password")==(""))
```

However, the input validation blocked empty and null strings, so this fault was masked and no test case can cause it to result in a failure.

It may be safe to ignore the masked faults. In the current software configuration, they cannot result in failure. However, it may be reasonable to advocate more robust testing where we explicitly test for masked faults, but that is beyond the scope of this research. Not detecting the scope change faults is more serious, and indicates a “hole” in our mutation operators. We plan to design and implement a “scope change” operator in the future.

6 Related Work

Most research in testing web applications has focused on client-side validation and static server-side validation of links [11]. These are all static validation and measurement tools, none of which support functional testing or black box testing of programs deployed on the web. Some recent research has looked into testing software as statically determined [13], but not the problem of distributed integration, as this paper does. Liu, Kung, Hsia and Hsu [14] test definition-use pairs among client and server web pages and software components. The focus was on data interactions rather than control flow. Ricca and Tonella [25] proposed an analysis model and corresponding testing strategies for *static* web page analysis. Di Lucca and Di Penta [15] proposed testing sequences through a web application that incorporate operational transitions, specifically focusing on the back and forward button transitions. Andrews et al. [2] introduced a system-level testing technique for web applications. Elbaum, Karre and Rothermel [7] proposed a method to use what they called “user session data” to generate test values for web applications. This method extracts data values

Table 3. Seeded faults detected

JSP file	♯ Faults seeded	♯ Tests	Found	Ratio
about	4	7	4	1.00
browse	20	13	16	0.80
categories	26	11	21	0.81
category_edit	17	6	14	0.82
index	4	4	3	0.75
login	19	7	11	0.58
logout	3	3	2	0.67
navigation_bar	2	5	2	1.00
page_footer	2	2	2	1.00
page_header	5	2	5	1.00
record_add	9	4	9	1.00
record_delete	0	n/a	n/a	n/a
record_edit	21	6	14	0.67
record_insert	9	4	9	1.00
record_search	3	2	3	1.00
update_search	3	2	3	1.00
Total	**147**	**80**	**118**	**80%**

from previous user inputs. We are investigating the use of a technique called *bypass testing* [23] for input data validation. Smith and Williams [26] applied mutation to a healthcare web applications at the unit level, using the Jumble mutation tool. They used only traditional (statement-level) mutation operators, and did not test the interactions among web components.

Dobolyi and Weimer [6] presented a model to distinguish differences between pairs of XML/HTML documents, with the goal of supporting regression testing and reducing testing effort. Dobolyi and Weimer represented the documents using a tree structure and compared them based on semantic information. Mutation operators were used to train a comparator to recognizes potential errors. Several situations were not considered to be errors. For example, the number of element inversions do not indicate serious errors (e.g., <b><u>The</u></b> vs. <u><b>The</b></u>). Reordering child nodes of a tree does not indicate errors since the semantics of the web page are not changed. Indeed, we found that altering attribute values can introduce an error. For instance, changing a value of a `value` attribute of a hidden form field can result in submitting inappropriate data to the system.

Because web components interact through interfaces and declarations of invocations are not explicit, it is possible that parameters submitted through the requests may not match. As a result, the application may exit abnormally, return inappropriate results, or fail at runtime. Halfond and Orso [9] presented a static analysis technique to identify parameter mismatches between two web components. In the future, we plan to incorporate this idea into a mutation operator.

7 Conclusions and Future Work

This paper presents a new approach to testing web applications by applying mutation analysis to the connections among web application software components. Eleven web mutation operators are defined and have been implemented in a tool called webMuJava. Results from a case study are presented that show tests created by applying mutation analysis to web applications can help find faults. The results have demonstrated the need for an additional "scope change" mutation operator, which we are already implementing in webMuJava. Because web applications commonly interact with databases, we also plan to implement SQL mutation operators [27]. We also plan to perform controlled experiments using larger, more complex, and industrial web applications to further evaluate this approach. When webMuJava is stable, we will release it to the community as an experimental resource.

Acknowledgement: We thank Jae Hyuk Kwak for hand-seeding faults into our subject JSPs.

References

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2008. ISBN 0-52188-038-1.

[2] A. A. Andrews, J. Offutt, and R. T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4(3):326–345, August 2005.

[3] P. Chevalley and P. Thévenod-Fosse. A mutation analysis tool for Java programs. *Journal on Software Tools for Technology Transfer (STTT)*, September 2002.

[4] H. Dahdah. Amazon s3 systems failure downs web 2.0 sites, July 2008. Online: http://www.computerworld.com.au/article/253840, last access September 2009.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[6] K. Dobolyi and W. Weimer. Harnessing web-based application similarities to aid in regression testing. In

20th IEEE International Symposium on Software Reliability Engineering. IEEE, November 2009.

[7] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, March 2005.

[8] Y. Guo and S. Sampath. Web application fault classification - an exploratory study. In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 303–305, New York, NY, USA, 2008. ACM.

[9] W. G. J. Halfond and A. Orso. Automated identification of parameter mismatches in web applications. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 181–191, New York, NY, USA, 2008. ACM.

[10] M. Hicks. Paypal says sorry by waiving fees for a day, October 2004. Online: http://www.eweek.com/c/a/Web-Services-Web-20-and-SOA/PayPal-Says-Sorry-by-Waiving-Fees-for-a-Day/, last access November 2008.

[11] R. Hower. Web site test tools and site management tools, 2002. Online: http://www.softwareqatest.com/qatweb1.html, last access September 2009.

[12] C. Kallepalli and J. Tian. Measuring and modeling usage and reliability for statistical web testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, November 2001.

[13] C. H. Liu, D. Kung, P. Hsia, and C. T. Hsu. Structural testing of Web applications. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 84–96, San Jose CA, October 2000. IEEE Computer Society Press.

[14] C. H. Liu, D. Kung, P. Hsia, and C. T. Hsu. An object-based data flow testing approach for web applications. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):157–179, 2001.

[15] G. D. Lucca and M. D. Penta. Considering browser interaction in web application testing. In *5th International Workshop on Web Site Evolution (WSE 2003)*, pages 74–84, Amsterdam, The Netherlands, September 2003. IEEE Computer Society.

[16] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for Java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 352–363, Annapolis MD, November 2002. IEEE Computer Society Press.

[17] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava : An automated class mutation system. *Wiley's Software Testing, Verification, and Reliability*, 15(2):97–133, June 2005.

[18] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. muJava home page. Online, 2005. http://cs.gmu.edu/∼offutt/mujava/, http://salmosa.kaist.ac.kr/LAB/MuJava/, last access December 2008.

[19] A. Marchetto, F. Ricca, and P. Tonella. Empirical validation of a web fault taxonomy and its usage for fault seeding. In *WSE '07: Proceedings of the 2007 9th IEEE International Workshop on Web Site Evolution*, pages 31–38, Washington, DC, USA, 2007. IEEE Computer Society.

[20] J. Offutt. Quality attributes of Web software applications. *IEEE Software: Special Issue on Software Engineering of Internet Software*, 19(2):25–32, 2002.

[21] J. Offutt, Y.-S. Ma, and Y.-R. Kwon. The class-level mutants of muJava. In *Workshop on Automation of Software Test (AST 2006)*, pages 78–84, Shanghai, China, May 2006.

[22] J. Offutt and Y. Wu. Modeling presentation layers of web applications for testing. *Software and Systems Modeling*, published online July 2009. DOI: 10.1007/s10270-009-0125-4.

[23] J. Offutt, Y. Wu, X. Du, and H. Huang. Bypass testing of Web applications. In *15th International Symposium on Software Reliability Engineering*, pages 187–197, Saint-Malo, Bretagne, France, November 2004. IEEE Computer Society Press.

[24] S. Pertet and P. Narasimhan. Causes of failure in web applications. Technical Report CMU-PDL-05-109, Parallel Data Laboratory, Carnegie Mellon University, December 2005. Online: http://www.pdl.cmu.edu/PDL-FTP/stray/CMU-PDL-05-109.pdf, last access November 2008.

[25] F. Ricca and P. Tonella. Analysis and testing of Web applications. In *23rd International Conference on Software Engineering (ICSE '01)*, pages 25–34, Toronto, CA, May 2001.

[26] B. H. Smith and L. Williams. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software*, 82(11):1819–1832, 2009.

[27] J. Tuya, M. J. Suarez-Cabal, and C. de la Riva. Sqlmutation: a tool to generate mutants of sql database queries. In *Second Workshop on Mutation Analysis (Mutation 2006)*, Raleigh, NC, November 2006.

[28] T. R. Weiss. Two-hour outage sidelines amazon.com, August 2006. Online: http://www.computerworld.com/action/article.do?-command=viewArticleBasic&articleId=9002687, last access November 2008.