TESTING WEB APPLICATIONS WITH MUTATION ANALYSIS

by

Upsorn Praphamontripong
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_____ Dr. Jeff Offutt, Dissertation Director

_____ Dr. Paul Ammann, Committee Member

_____ Dr. Huzefa Rangwala, Committee Member

_____ Dr. Rajesh Ganesan, Committee Member

_____ Dr. Stephen Nash, Senior Associate Dean

_____ Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Spring Semester 2017
George Mason University
Fairfax, VA

Testing Web Applications with Mutation Analysis

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Upsorn Praphamontripong
Master of Science
Central Michigan University, 2004
Bachelor of Science
Thammasat University, 1997

Director: Dr. Jeff Offutt, Professor
Department of Computer Science

Spring Semester 2017
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to my parents, Jate and Khemsiri, who always work hard to provide me with the finest education; my sister, Prachayani, and my brother, Ularn, who constantly cheer me up and give me warmhearted support; my advisor and mentor, Dr. Jeff Offutt, whose encouragement and guidance have made it possible for me to successfully accomplish this research; and most important of all, my children, Palawudth and Adithya, and my husband, Somsak, whose love, patience, and sacrifice are only to see me doing what I am most passionate about.

Acknowledgments

This great milestone of mine would not have been possible if Dr. Jeff Offutt did not let me interview him prior to my joining George Mason University. Throughout this long journey, he has always been there for me with tremendous encouragement and understanding, insightful guidance, consistent caring and patience and even admonishment sometimes. He has always been supportive and especially guiding me throughout the darkest moment. He always believes in me. He told me *"You can do more than you think!"* He has challenged me in each of every possible way a great mentor and dissertation chairperson would do. His feedback, both positive and negative, have helped me to improve the quality of my research and groomed me as a researcher. I could not have professionally grown this much without his constructive and invaluable guidance. If it were not for the opportunity to work with him under the *Self-Paced Learning Increases Retention and Capacity (SPARC)* project and the opportunity he gave me to teach a *Design and Implementation of Software for the Web (SWE 432)* course, I would have never discovered my true passion and pursuit of academia career, nor could I have succeeded this far. *"Thank you so much, Dr. Offutt. You are always my role model and my best mentor!"*

I would also like to express my gratitude to Dr. Paul Ammann, my committee member, for his constant generosity and encouragement. Not only has he provided me valuable suggestions on my research and academia career, but he has always been supportive, especially when I started teaching for the first time. My genuine appreciation goes to Dr. Huzefa Rangwala for stepping up voluntarily to serve as my committee member at the time when I needed it the most. I would also like to extend my sincere gratitude to Dr. Rajesh Ganesan, my committee member who has given me constructive suggestions and alternative views from a system engineer's perspective.

Furthermore, my sincere thanks goes to Dr. Kinga Dobolyi, my wonderful colleague and mentor. I am greatly indebted for her inspiration, advice, and mentorship. I am truly thankful that we shared the *"up and down"* moments and the frequent *"fifteen-hours a day of Marmoset preparation"* together. I wish to thank Dr. Marcio E. Delamaro as well for his insights on my Ph.D. symposium and his enthusiastic assistance with my presentation. I also wish to thank Alastair Neil for his support and troubleshooting Tomcat and all technical difficulties I have had for these past years. I would especially like to thank Lin Deng for helping me set up the experiment and for his creative ideas on *"killing mutants."*

At the risk of the list getting too long, I must acknowledge and express my gratitude to fellow students, colleagues and friends – Sunitha Thammala, JingJing Gu, Nan Li, Vinicius H. S. Durelli, Ehsan Kouroshfar, Nariman Mirzaei, Garrett Kent Kaminski, Vasileios Papadimitriou, Bob Kurtz Jr., and the experiment participants including Han Tsung Liu, Jae Hyuk Kwak, Maha Al-Freih, Norah Alobaidan, Noor Bajunaid, Scott Brown, Colin Buckley, Dr. Nida Gökçe, Santos Jha, Pranab Khanal, Kiranmai Kovuru, Alexander Marcus, Benjamin McWhorter, Mayank Mehta, Shanthi Ramachandran, Victor Shen, Sunny Singh, Dana Mun Turner, Lyla Wade, Wade Ward, and Hozaifah Zafar.

Table of Contents

List of Tables

List of Figures

Abstract

TESTING WEB APPLICATIONS WITH MUTATION ANALYSIS

Upsorn Praphamontripong, Ph.D.

George Mason University, 2017

Dissertation Director: Dr. Jeff Offutt

Web application software uses new technologies that have novel methods for integration and state maintenance that amount to new control flow mechanisms and new variables scoping. While modern web development technologies enhance the capabilities of web applications, they introduce challenges that current testing techniques do not adequately test for. Testing individual web software component in isolation cannot detect interaction faults, which occur in communication among web software components. Improperly implementing and testing the communications among web software components is a major source of faults.

This research presents a novel solution to the problem of integration testing of web applications by using mutation analysis, *web mutation testing*. New mutation operators are defined and a tool, *webMuJava*, that implements these operators is presented. A series of four experimental studies was conducted using 15 web mutation operators. The results show that (1) web mutation generated very few equivalent mutants (only 6% on average across four experiments); (2) web mutation testing provides 100% coverage on web mutants whereas 12 independently developed tests designed to satisfy traditional testing criteria provides 47% coverage; (3) no traditional tests kill FOB mutants; (4) web mutation tests can help create tests that are effective at finding web faults; (5) tests designed for web mutation testing detects all kinds of method-level Java mutants; (6) tests designed for

method-level Java mutation testing missed all FOB, WCTR, and WSIR mutants; (7) Java mutants help design tests that verify individual web components whereas web mutants help design tests that verify interactions between web components; (8) while overlapping, web mutation testing and method-level Java mutation testing are complementary and can improve the quality of tests; (9) three mutation operators that are redundant and can be excluded from the testing process with minimal loss in fault detection are WFUR, WHID, and WLUD; and (10) the FOB and WSIR operators may be particularly strong and can lead to high quality tests.

Chapter 1: Introduction

"Has anyone gone a day without using the web?" Typical responses will range from using the web "every day" to "many times a day" and usually web users not only read static pages on the web but they also interact with software on the web. These software programs are called *web applications*. Without a doubt, human daily activities rely heavily on web applications.

Web applications (*web apps*) are software systems that run on a server and use web browsers as clients to display the user interfaces. A key benefit of web apps is that users can access web apps at anytime and from anywhere without installing any software. For decades, web apps have a major impact on our daily lives. Worldwide Internet usage increased more than 918.3% during 2000-2016 [100]. Profit and non-profit organizations have increasingly been shifting their services to web apps [16, 58, 97, 100] to maintain and enhance their competitive positions. The consolidation of services in web apps has significantly helped enterprises centralize their resources and improve their levels of service. For example, many e-commerce sites such as BankOfAmerica.com, amazon.com, eBay.com, and statefarm.com are web apps. Likewise, academic institutions have transformed services such as admission, grading, and registration processes to be online. Healthcare providers and health insurance companies such as Kaiser Permanente and Anthem now base their services on web apps to improve their response time, to increase availability, to ensure accessibility, and to facilitate claim filing [97]. Many government agencies have deployed web apps to facilitate operational processes such as registering vehicles and paying property taxes. The Washington D.C. U.S. Customs and Border Protection (CBP) has deployed the Automated Commercial Environment (ACE) Secure Data Portal, a web portal that facilitates and analyzes consolidated border processing information so that violations of U.S. laws can be identified and confirmed quickly [16].

However, in practice, while web apps can tremendously facilitate daily activities, industry has suffered huge losses due to failures of web apps. In October 2004, PayPal, the online payment service, had to waive all customers' transaction fees for an entire day because of a service outage after upgrading its software [38]. The service unavailability may have been due to integration errors [87]. In August 2006, Amazon.com disconnected its website for two hours because its web apps were not fully functioning, losing millions of dollars as its customers could not access the system [106]. In July 2008, Amazon's S3 systems, a web component hosted storage service, failed so that many businesses relying on this service lost information and significant revenue [17]. Consequently, Amazon lost customers and its reputation was damaged. In September 2011, Target's website was out of service for two hours and functioning intermittently for an entire day after launching a new clothing campaign. The malfunction of its web app delayed and canceled numerous customers' orders. In February 2011, Google Mail service (Gmail) was temporarily unavailable, causing the company to disable part of its services. Over 120,000 users could not access their email accounts. Google reported the cause of the outages was a fault in a Gmail storage software update [12]. In my personal experience in March 2012, purchasing two of the same items from the BodyShop website turned a buy-one-get-one 50% off to a half price each item. Another personal experience in June 2012, after placing an order through Amazon website and successfully logging off, I clicked the back button several times intending to find information from a previous screen. Typically, users would expect their payment information to be inaccessible once they successfully logged off from their online account, or at least they should be forced to re-log in. To my surprise, after navigating back several screens, I was shown a screen with full payment information for the purchase I just made. Such exposure of sensitive information can raise chances of unauthorized access and misuse of confidential information. In October 2011, Bank Of America's website was sporadically unavailable for six consecutive days due to the upgrading of its online banking [10]. In September 2012, six US major banks (Bank of America, JPMorgan Chase, Citigroup, US Bank, Wells Fargo, and PNC) suffered online attacks, resulting in denial of services to thousands of customers

[86]. Furthermore, in August 2013, Amazon.com was down for approximately 30 minutes, causing the company to lose approximately $2 million [14]. Dropbox, a cloud-based file-sharing company, suffered an outage in January 2014. The company reported that the failure was due to a fault in a system maintenance script [33]. In February 2015, Anthem suffered cyberattacks that breached personal information (Social Security numbers, birthdays, addresses, and income data) for tens of millions of customers and employees [34]. As another personal anecdote, in April 2016 my advisor lost 20 minutes worth of data entries after clicking the browser back button while using the TurboTax web software. In December 2016, at least 500 million Yahoo user accounts were compromised [32].

Moreover, records and announcements show that universities' web apps have been hacked, allowing unauthorized access to students' confidential information such as social security numbers. Without appropriate and adequate web app testing, systems crash and software fails. The impact of unreliable web apps may range from inconvenience (i.e., malfunction of the app or users' dissatisfaction), economic shortage (i.e., interruption of business), or catastrophic impact (i.e., loss of life due to failures of safety-critical systems such as medical web apps). Unreliable web apps can even raise the chance of terrorist attacks [110].

These numerous problems show that web apps are often deployed with significant faults. Ensuring the proper functioning of web apps is difficult due to the novel and powerful technologies that have been created to design and build them. Furthermore, user loyalty toward any particular website is usually low and is primarily determined by the quality of web apps [79]. The increasing reliance on web apps demands appropriate web-specific testing techniques to ensure reliability of web apps, as well as to ensure that the apps meet their users' expectations.

Web app testing is an ongoing research area. Many studies on testing web apps have been attempted. While some studies focus on different web-specific features and some consider different development frameworks, some overlap. At the moment, researchers, developers, and testers do not know how best to test web apps. This dissertation focuses on developing

a web app testing technique based on mutation analysis (Section 2.1) called *web mutation testing*. Further discussion on goals, scope, and the rationale for this research is presented in Section 1.2. Before discussing web mutation testing, an overview of challenges in testing web apps is presented below. These challenges were analyzed along with the nature of web apps to list various kinds of web faults (Chapter 3), which are later used to design web mutation operators (Chapter 4).

1.1 Challenges for Testing Web Applications

Because the effectiveness of mutation testing depends primarily on the mutation operators used to mutate the software artifacts, understanding potential web faults is mandatory prior to designing the operators. Therefore, various resources including a publicly available study on web faults and bug reports were investigated.

Thus far, no standard web fault model has been published. Several studies have attempted to classify web faults [26, 35, 69, 93]. However, the existing categorizations overlap without being consistent or complete. Therefore, this research modeled web faults intending to ensure coverage of faults that occur in communications between web components, which are referred to as *interaction faults*. To accomplish this, the nature of web apps and web development technologies were explored. An analysis of web faults from the existing study (as mentioned above) and an analysis on web specific features that can introduce web faults and that affect the testing process (i.e., challenges in testing web apps) were integrated to create a list of web faults. Further information on potential web faults according to each challenge is given in Chapter 3. Note that this research is particularly focused on interaction faults and primarily considers server-side web software.

Web apps are user interactive software apps deployed on a web server and accessible through its screens (sometimes called pages) via web browsers. Web apps can be developed from and consist of multiple diverse, distributed, and dynamically generated web software components. *Web components* are software modules independently compiled and executed.

They implement different parts of the web apps' functionality. They interact with each other to provide services to the users. Interaction between web components relies on the HyperText Transfer Protocol (HTTP). Each interaction (from sending a request to receiving a response) requires individual HTTP connection. The individual connection causes state of web app to be disconnected; i.e., information is not maintained between connections. This introduces the stateless property of the HTTP. Web components may be created with different software technologies, reside on different servers, and integrate dynamically. The appearance (user interface, which are screens rendered in web browsers) of web apps may vary depending upon state, users, time of day, and geography. In addition to the controls provided by the web apps, users may interact with the web apps using browser controls such as back, forward, and reload. The nondeterministic behavior allowed by these technologies increases complexity, the potential for errors, and the difficulty of testing. Section 2.2 discusses the engineering impacts of web app technologies in depth.

In addition to techniques used in traditional software development, many new technologies have been created to design and develop web apps. These include the HyperText Transfer Protocol (HTTP), and non-compiled languages or scripting languages such as HyperText Markup Language (HTML) and JavaScript. Many technologies have been used to control execution flow and manage the state of web apps, allowing web apps to provide a variety of services and to evolve and be released to respond to users' needs more rapidly than traditional software apps.

Although these web development techniques are powerful and enhance the functionality of web apps, they introduce new challenges in testing. The problems this research addresses are based on the following seven specific challenges.

1. **Users' ability to control web apps**: Web apps allow *operational transitions* [82], which are transitions caused by users, web clients, or system configuration. Unlike traditional programs where interactions between users and the software are definite and cannot be changed by the user, web users can use features of the web browser to bypass the app's normal execution flow. Users can navigate to previously viewed screens

by pressing the `back` and `forward` buttons, reload the screen using the `refresh` button, and rewrite URLs (Uniform Resource Locators) directly. The `back` and `forward` buttons record all the data (possibly including the current state of the app and the current state of the browser) that were posted to the server by the previous requests. These data will be resent when users navigate back and forth between screens. This kind of requests cannot be distinguished by the web server. The order in which the requests are made can affect how the app behaves. The users' ability to control web apps can disrupt the control flow, causing inconsistencies in web apps' navigation, corrupting the state of web apps, and allowing confidential information to be accessed improperly. For example, in online banking, pressing the `back` button after a user successfully transfers funds between accounts can result in duplicated transactions. Funds to be withdrawn and deposited may be inconsistent and incorrect.

Let's use the *Text Information Management System*, a small web app allowing users to maintain text information, to illustrate how web-specific features can introduce web faults and impact the testing process. Figure 1.1 illustrates the case when a user has successfully logged in to the *Text Information Management System* and tried to add information to his/her account. Instead of completing the adding process (i.e., submitting a data entry form of a `record_add` web component), he/she clicked the browser's back button, leading him/her to a screen listing his/her existing information (the `browse` web component). At this point, he/she may expect that the data recently entered but not submitted to the system still remain when he/she revisits the form (i.e., `record_add`). On the other hand, under the same circumstances, other users may expect a blank form. The design decision is up to the system owners and developers to ensure that the operational transitions are correctly allowed for. However, very often, this kind of transitions is overlooked by the developers and the testers.

Operational transitions can cause inconsistencies in web apps' navigation and corrupt the state of the apps. Almost half of user sessions include navigation caused by the use of browser's navigation features [102]. Web apps have little control over

the user's ability to change the flow of execution. At the same time, many developers are not familiar with the conceptual impacts of operational transitions. Hence, appropriate testing mechanisms are vital.

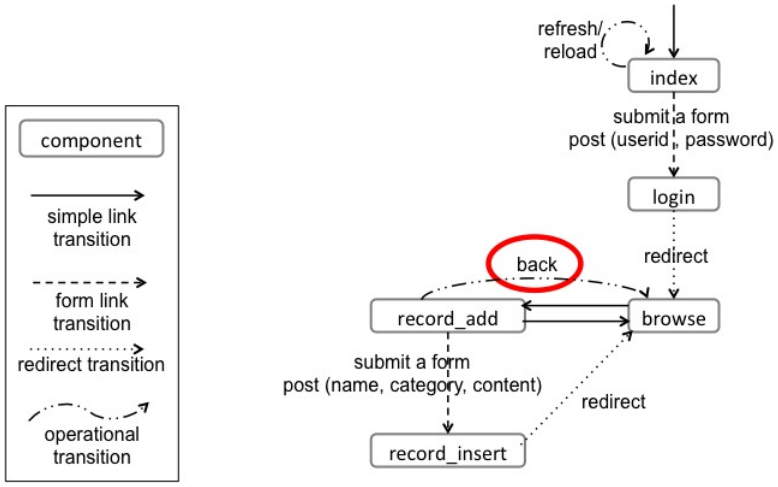

Figure 1.1: Use of web browser features (Back button)

2. **Identifying web resources with Uniform Resource Locators (URLs)**: Web resources (including images, style sheets, JavaScripts, Java Server Pages (JSPs), and Java servlets) are parts of a web app and are identified by URLs. These components are integrated dynamically during execution. They are not available until after the web app is deployed and can be modified any time during execution. Inappropriately identifying web resources could lead to unexpected behavior of a web app including an attempt to visit an unintended or nonexistent screen or a request to an incorrect JavaScript. These mistakes can lead to failures.

   Figure 1.2 shows a scenario where an incorrect URL is specified. As a web app can be developed by multiple developers, there may be multiple versions of web components that authenticate users. Using the URL to an incorrect `login` component can perform different authenticating mechanisms or access a different set of credentials, introducing inconsistencies and violating data integrity.

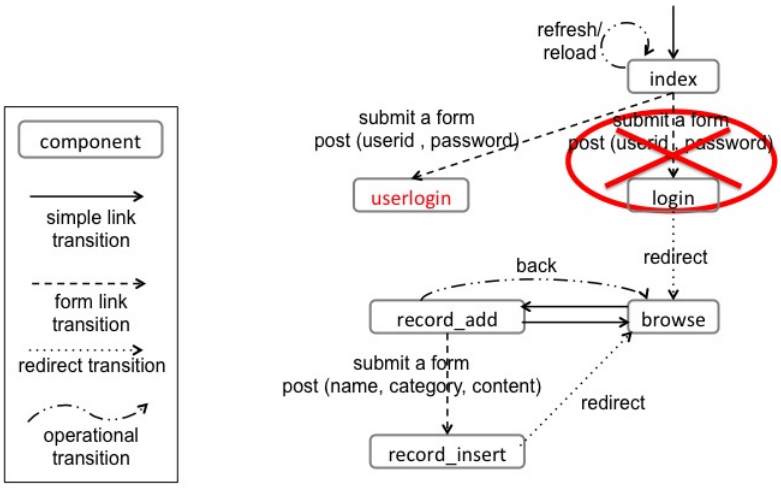

Figure 1.2: Identifying web resources with URLs (submit a form to a different web resource)

3. **Communication between web components through HTTP requests**: Each HTTP request consists of a type of the request (called a *transfer mode*) along with a URL to which the request is sent and data associated with the request. HTTP allows several transfer request modes, each of which can impact the action and response of the apps. The transfer mode determines how the data are packaged when they are conveyed to the server. The most commonly used transfer modes are `GET` and `POST`. A `GET` request transfers information as parameters (represented as name-value pairs) appended to the URL (destination) string; for example, `https://cs.gmu.edu:8443/uprapham/example/randomString?param1=value1\&param2=value2`. With most browsers, users can see the URL along with the actual data transmitted. Users then can bookmark the URL or save it for later access. A `GET` request is properly used to **retrieve** data from a web server. The size of data transmitted is limited, usually to 1024 bytes. Due to the size restriction, submitting a large data set (i.e., resulting in a long URL) using a `GET` mode may cause invalid behavior on the web server (for example, data anomalies or a system crash).

   Unlike a `GET` request, where data are attached to a URL and only the URL is sent to the server, a `POST` request packages data and constructs the request as an HTTP

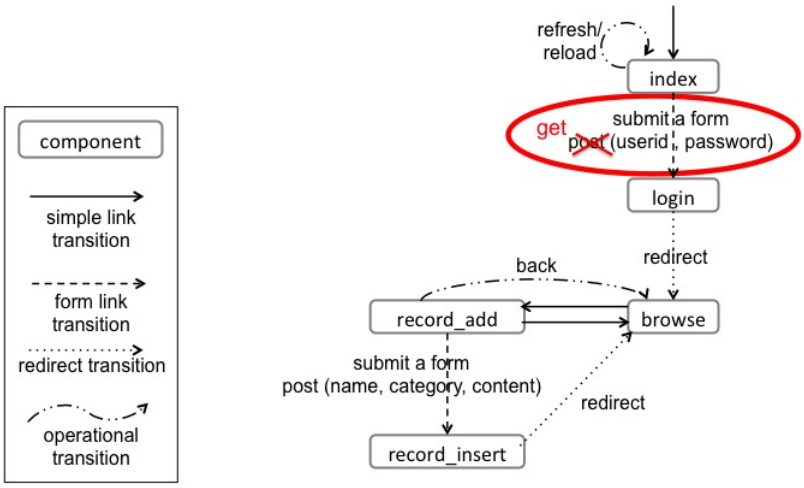

Figure 1.3: Use of HTTP request modes

message. The size of data transmitted using a `POST` request is not constrained. A `POST` request is usually used when information is sent as part of the request such as when storing or updating data into a database, uploading a file, submitting a form, or sending an email. Data transmitted using a `POST` mode usually change state on the web server. Improper use of these transfer modes can result in inconsistent behavior of web apps or reveal confidential information; for instance, a different presentation of a response screen or loss of input values. As another example, if a `GET` request is used to transfer a form that includes hidden fields (such as a counter tracking a log-in attempt), the user can see this information and may change it to gain unauthorized access.

Figure 1.3 illustrates an improper use of HTTP transfer mode. As soon as a user opens the first screen of the app (the `index` web component), he/she will be prompted to log in with a userid and password. If the form is submitted with a `POST` request, the login information will be packaged in an HTTP message and transmitted to a web component named `login` to perform an authentication. If the form is submitted with a `GET` request, the login information will be appended to the URL. This can expose confidential information.

4. **Communication via data exchanges between web software components**: Unlike communication between traditional software components, interactions between web software components rely on sending requests along with needed information (i.e., parameter-value pairs). The target component (the web component that will process the request) uses the provided information to process the requests. Due to the fact that web apps are usually developed by teams, inconsistencies in communications can occur. The provided information may be referred to by names that are different from the parameters' names the target component expects. Values being sent may be incompatible with the types expected by the target component. Although values are exchanged between web components as strings, some operations or processes may use the values to infer certain constraints. For example, a target web component that verifies the availability of funds in an online banking may expect the value to be of type `float` or `double`; a component that authenticates login attempts may expect the number of attempts to be of type `integer`. If parameter mismatches occur, the app may fail once the target component executes or parses this received information. In addition to parameters' names and their values, the number of parameters being sent may be different from the number of parameters expected by the target component. These inconsistencies can affect the behavior of the app. As an example, for a search facility, if a component sends a request with more parameters than the target component expects, the extra parameters are not taken into account and hence the search result may not succeed.

   Another example, as demonstrated in Figure 1.4, is when two online courses use the *Text Information Management System*. Both courses can be accessed through the same initial screen (the `index` web component) but only the student's login information (regardless of the course identification) is expected by the authentication web component (the `login` component), although the `courseID` may be included in the request. As a result, for a student who enrolled in both courses, once the login is successful, a student may be sent to an unintended course's screen. This inconsistency

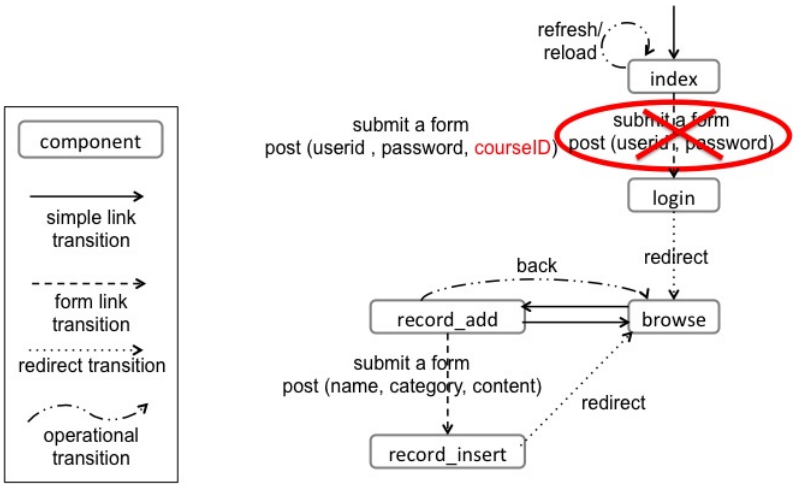

Figure 1.4: Communication via data exchange (mismatched parameters)

is because of parameter mismatches between web components.

5. **Novel control connections**: In traditional software, connections between software components are based on method calls, inheritance, and message passing. In addition to these techniques, web apps use several control connections that were invented specifically for web software. Different web app development frameworks introduce different control connections. The common and basic control connections introduced by the J2EE framework, for example, are `forward` and `redirect`. `Forward` and `redirect` connections transfer control from one web component (the *source*) to another component (the *target*), similar to a method call but with distinct differences. Most importantly, control does **not** return to the source component.

   `Forward` connections transfer control from a source component to a target component; for instance, by using Java Server Pages (JSPs) to pass parameters with a `forward` transition. The target must be on the same server. The control does not return to the source component. A `forward` does not inform the browser of the transfer and does not include the data from the original request (although the source component may share the data in-memory). The browser hosting the screen containing

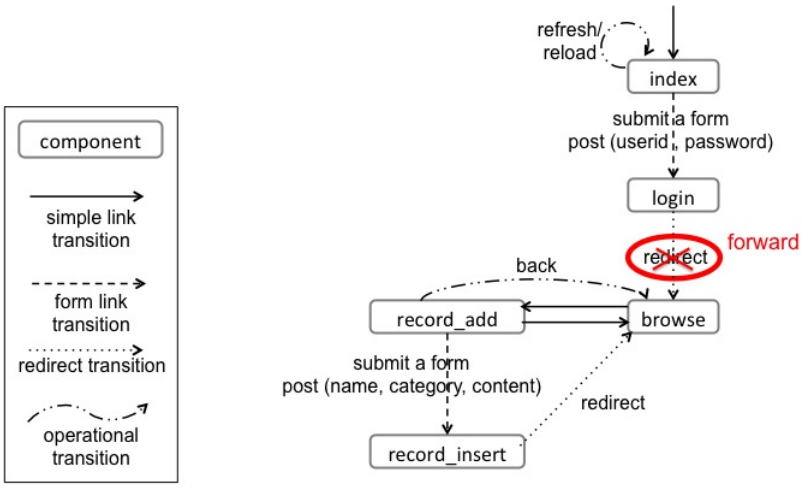

Figure 1.5: Novel control connection (forward connection instead of redirect connection)

a `forward` connection is unaware that control has been transferred. As a result, every time the browser tries to reload the screen, the original request with the original URL (identifying the source component) is repeated. Therefore, inappropriate use of a `forward` connection may result in data anomalies; for example, using it for an operation inserting or updating information to a database may lead to inadvertently duplicating an insert or an update to the database.

`Redirect` connections explicitly instruct the browser to create a new request and send it to the target component. The target may be on another server. All request data are included in the new request but the source and target might not be able to share any data. Accordingly, objects executed in the original request scope are unavailable to the new request. Similar to `forward` connections, by using `redirect` connections, control does **not** return to the source component. An example of a `redirect` connection is the Java servlet's `redirect` transition.

To summarize, both `forward` and `redirect` behave similarly in three ways: they transfer control to a new software component, they pass the user data that was included in the request (form values, etc.), and control does **not** return to the originating component (unlike a traditional method call). However, the differences are significant.

A `forward` transition must be on the same server and is handled by the server. A `redirect` transition returns to the client's browser, and the browser creates a new request that can go to a different server. Thus, some state variables, such as the request and session objects, that are available with a `forward` are not available with a `redirect`. Many developers do not understand the somewhat subtle differences between these two and often use them incorrectly, causing errors in the software state.

6. **Server-side state management**: A web user usually interacts with a web app to accomplish certain tasks and the communications involve a series of related interactions between the user and the app. Due to the stateless property of the HyperText Transfer Protocol (HTTP), each request sent between web components is handled in separate threads by components that do not share memory space. Therefore, web apps cannot maintain program state in the traditional ways of persistent objects and variables in a shared scope. To handle a user session, persistent data must be stored in an in-memory object whose accessibility must be specified.

   Web development frameworks have created new techniques to manage state. This research considers two state management mechanisms. First, in-memory objects can be used to persist data and maintain the program state. In J2EE, these objects are called *session* objects. Session objects allow web apps to keep track of data between multiple HTTP requests. Data are stored as attributes of objects so they can be accessed later. Improperly handled session objects (for instance, not creating a new instance of the object when needed) can cause data anomalies and corrupt the state of web apps.

   Another mechanism, scope setting, is used to identify the accessibility of an in-memory object. The J2EE framework (illustrated in Figure 1.6), for example, provides four levels of variable scoping: `page`, `request`, `session`, and `application`. A variable of `page` scope is only available within the component that initially receives the request. A variable of `request` scope is available to the component that receives the

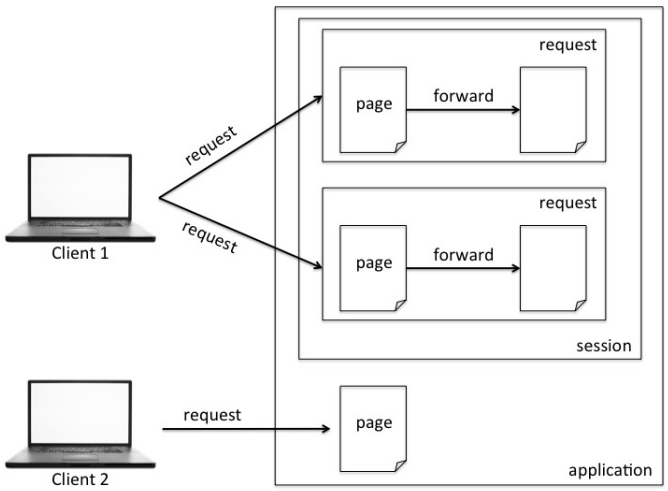

Figure 1.6: State scope of web components in J2EE [78][1]

request and all components that execute during the request. A variable of `session` scope is available to all components that are defined as being part of the web app, even across multiple requests, throughout the user session. A variable of `application` scope is available to all apps running under the same servlet context. Scope setting helps determine the lifetime of web components and their availability and accessibility. However, many programmers have difficulty understanding these scopes and frequently specify the scope inappropriately, allowing it to be accessed when it should not be and vice versa.

7. **Client-side state management**: In addition to using server-side state management techniques, the app state can be persisted across multiple requests by placing data directly into the user interface as encoded in HTML files. One common approach is the use of hidden form fields. *Hidden form fields* are form `input` elements that are specified as `hidden`, allowing web apps to place data in the response HTML that will be submitted in the next request. Hidden form fields are not rendered on the screen. However, users can access the source code to view and change their values. Hidden

[1]Reproduced with permission from J. Offutt

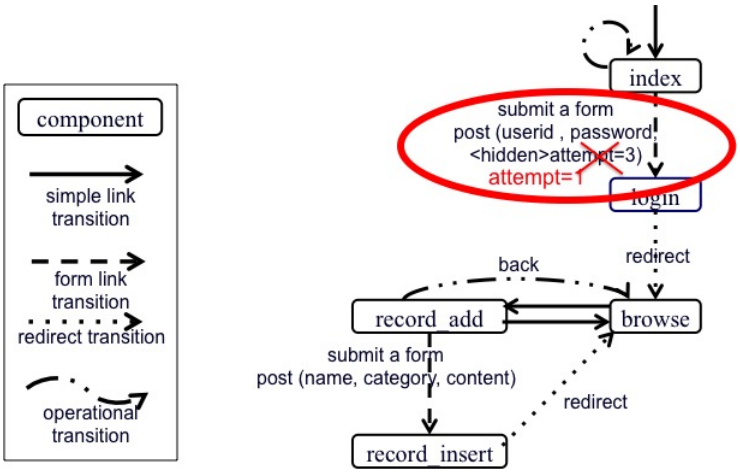

Figure 1.7: Client-side state management (replacing necessary hidden input value)

form fields can be placed into the HTML by the web app, and then are sent during the next request as normal form data. Since hidden form fields are presumably not visible (on screen) to the users, web apps do not generally validate their values. As a consequence, improper values of hidden form fields may be accepted.

Figure 1.7 shows the scenario when a hidden input `attempt` that tracks the number of login attempts is altered, incorrectly allowing a user extra login attempts. Omitting necessary data or submitting inappropriate data via hidden form fields to the server can cause unintended results, possibly leading to unauthorized access to the system.

These seven challenges lead to faults occurring in communications between web components, defined in this research as *interaction faults*. Possible web faults are discussed in Chapter 3. Interaction faults, as defined here, are new to web apps and the current state-of-art in web app testing is still unclear how they can best be detected. Interaction faults make testing more complicated. As a result, a testing approach dealing specifically with interaction faults is needed.

1.2 Goals and Scope of This Research

Web apps are subject to unique challenges because of the development frameworks used to create them and the web component generation and integration that are dynamic and complicated. The goal of this research is to invent and investigate the applicability of mutation testing to web apps with a focus on detecting interaction faults before the software is deployed. The ultimate goal of this research is to improve the quality of web apps and reduce the cost of testing web apps by using effective web mutation operators for test case design and generation.

Web apps are constructed differently from traditional software. New frameworks and features that do not exist in traditional software have been created to design and develop web apps. Existing techniques for traditional software testing do not support new web-specific features.

There have been numerous studies on testing individual web software components. Nonetheless, so far, very little research has focused on testing web component interactions. Though web app testing at the unit level is necessary as it verifies each individual web components, interaction faults cannot be detected by testing individual component in isolation. Indeed, communications between web components are common sources of faults in web apps. For instance, when two web components try to interact with each other, they may make different assumptions. Mismatched assumptions can introduce faults into the system.

Several techniques have been developed to test web apps. Some approaches focus on input validation [74, 80, 83]. Some approaches rely on finite state machines [8]. Others are based on user-sessions [30, 94, 96]. Mutation testing has been applied to web apps [27, 54]. Offutt et al. [83] introduced bypass testing, which has been extensively adopted and adapted for web app testing [56, 74, 101]. Nevertheless, web app testing is a relatively new research area and several web-specific features and challenges have not been dealt with. Though some techniques consider the interactions between HTML forms, most do not take into account the user's interactions with the apps (e.g., using `forward`, `back`, and `reload` buttons) and

interactions between non-HTML web components such as JSPs and Java servlets. This research focuses on developing an approach to test web apps and improving the quality of test cases with an emphasis on revealing interaction faults.

Six specific objectives of this research are:

1. Understand potential interaction faults
2. Design web-specific mutation operators that reveal interaction faults, i.e., inventing web mutation testing criterion
3. Evaluate the web mutation operators to select a collection of operators that are effective at finding faults, but also as cost-effective as possible
4. Evaluate the fault detection capability of tests generated with the web mutation testing criterion
5. Evaluate the overlap between web mutation testing and traditional Java mutation testing
6. Evaluate the redundancy in web mutation operators

Since this research focuses on server-side web apps, JavaScript and AJAX are excluded. Although many web development frameworks exist, mutation testing requires source code to be available, thus limiting the choices for web apps used for empirical validation of web mutation testing. This research relies on J2EE-based web apps and considers web components as software modules developed with JSPs and Java Servlets. Though the definitions of web mutation operators are based on J2EE, the underlying concepts of the operators can be applied to other web development languages and frameworks with modification on the implementation.

1.3 Hypothesis and Approach

Mutation analysis specifically targets the structural and data aspects of software. It has been shown to be effective at finding integration faults [18, 39, 54]. As many faults in web apps are due to structural and data problems, mutation analysis is an obvious candidate for these kinds of faults. This research investigates the usefulness of applying mutation analysis to web apps, and evaluates its applicability in revealing web interaction faults. This research expects that mutation testing can be used to help improve and ensure the quality of tests.

Research Hypothesis:

> ***Mutation testing can be used to reveal more web interaction faults than existing testing techniques can in a cost-effective manner.***

To verify the hypothesis, the experiments (presented in Chapter 5) are conducted in four phases, each of which serves different purposes.

The first experiment focuses on verifying whether web mutation testing can help improve the quality of tests developed with traditional testing criteria by answering the following research questions.

RQ1: How well do tests designed for traditional testing criteria kill web mutants?

RQ2: Can hand-designed tests kill web mutants?

The second experiment examines the applicability of web mutation testing to detecting web faults by answering the following research questions.

RQ3: How well do tests designed for web mutation testing reveal web faults?

RQ4: What kinds of web faults are detected by web mutation testing?

The third experiment evaluates whether web mutation testing criterion and traditional Java mutation testing criterion are complementary to each other by answering the following research questions.

RQ5: How well do tests designed for web mutants kill traditional Java mutants and tests designed for traditional Java mutants kill web mutants?

RQ6: How much do web mutants and traditional Java mutants overlap?

The last experiment concentrates on reducing the testing cost in terms of the number of mutants generated by answering the following research questions.

RQ7: How frequently can web mutants of one type be killed by tests generated specifically to kill other types of web mutants?

RQ8: Which types of web mutants are seldom killed by tests designed to kill other types of web mutants?

RQ9: Which types of web mutants (and thus the operators that create them) can be excluded from the testing process without significantly reducing fault detection?

This dissertation approaches the challenges in testing web apps by recognizing that (i) web apps are developed differently from traditional software thus existing software testing techniques are insufficient for testing web apps, (ii) the majority of web faults occur in interactions between web software components, and (iii) web faults can be imitated using mutation operators and can be detected by tests designed with web-specific mutation testing.

To design web-specific mutation testing criterion (or web mutation testing, for simplicity), the following steps are carried out.

- **Investigate faults occurring in web apps**: Faults occurring at the unit level can be detected by unit testing of web apps, which is not very different from unit testing of other software apps. On the other hand, faults occurring at the integration and system level need special treatment as web apps integrate web software components that can be on multiple hardware/software platforms, written in different languages, and do not share the same memory space. Web software components and the content presented to the user may be dynamically generated and customized according to the server state and session variables. Requests made when web apps are executed are

independent and are handled by creating new threads on the software objects that handle the requests. This can lead to problems with testing interactions between web software components that do not exist with other software apps. Accordingly, this research focuses on examining faults due to interactions between web components. Various online resources, including existing studies on web faults and bug reports, have been investigated. An analysis from these resources is integrated with an analysis on the nature of web apps that impose challenges in testing web apps to create a web fault model.

- **Define web-specific mutation operators**: Using faults categorized from the previous step, web-specific mutation operators are defined. These operators (i) imitate faults that web developers make, such as replacing one scalar variable with another or replacing one accessibility setting with another; (ii) force good tests, such as fail on back (i.e., failing if and only if a browser `back` button is exercised); and (iii) imitate faults that web developers are unaware of or do not normally make, or faults that are hard to detect such as faults that occur when the control connection of web app is used inappropriately (e.g., `forward` control connection instead of `redirect` control connection).

1.4 Conventions and Terminologies

Throughout this PhD dissertation, *italic* font is used for emphasis and introducing new terms. `Typewriter` font is used for web specific features and keywords, and J2EE JSP and servlet codes and templates. When method names of web development frameworks or of web apps under tests are referenced in the main body of text, trailing parentheses are omitted.

URLs (Uniform Resource Locators) refer to a subset of URIs (Uniform Resource Identifiers). However, for simplicity, this research uses the term URL when referring to web resources.

Faults (or *software faults*) are abnormal conditions or defects in software that can potentially lead to software failures.

Failures (or *software failures*) are states (or behaviors) of software that propagate and do not meet the software's intended functionality.

Fault detection is the process of recognizing the existence of faults in software.

Interaction faults are faults that can occur in communications between web resources (or web components).

Test cases (so-called *test inputs* or *tests*) are inputs entering to a software app under test. These inputs include a collection of data values and a series of interactions between a user and the app.

Test requirements are specific conditions or elements that test cases must cover.

Test suites (or *test sets*) are collections of test cases.

1.5 Structure of this PhD Dissertation

The remainder of this document is organized as follows. Chapter 2 provides background on mutation analysis, its core concepts, and known limitations. Because this dissertation focus on introducing an approach to appropriately test web apps that is as cost-effective as possible, the chapter also emphasizes the computationally high cost of mutation analysis. The concept of mutation analysis forms the foundation for web mutation testing. This chapter also introduces background on the characteristics of web apps and the modeling of web apps that demand novel mechanisms for testing web apps, followed by a discussion on some existing techniques used for testing web apps.

Chapter 3 discusses possible faults occurring in interactions between web components. This chapter lists web faults based on the seven challenges in testing web apps in section 1.1. The fault categorization is later used to design the web mutation operators.

Chapter 4 introduces web mutation testing and presents definitions of novel source-code, web mutation operators. The emphasis is on testing the connections between web components by mimicking potential faults that can occur in the transitions. Web mutation

operators are grouped according to the seven challenges.

Chapter 5 presents an empirical validation of web mutation testing. The validation consists of four experiments. First, the experiment ratifies web mutation operators by examining how well tests designed with traditional testing criteria kill web mutants (RQ1) and whether the quality of these tests can be improved (RQ2). Second, the experiment evaluates how well web mutation-adequate tests detect web faults (RQ3) and analyzes the kinds of web faults that can be detected by web mutation testing (RQ4). Focusing on improving mutation testing, the third experiment examines whether web mutation testing and traditional Java mutation testing overlap (RQ5 and RQ6). Then, intending to minimize the number of web mutants generated (and thus reducing the cost of mutation testing), the last experiment analyzes and identifies redundancy in web mutation operators based on how difficult each group of web mutants can be killed by tests designed for other groups of web mutants (RQ7, RQ8, and RQ9).

Chapter 6 revisits the research problems (challenges in testing web apps, to be specific) and research questions, and draws on the findings to verify the research hypothesis. It summarizes the main contributions of this research. Finally, the chapter concludes with future research directions.

Chapter 2: Background and Related Work

This chapter introduces background on mutation analysis and its core concepts. It also presents characteristics of web apps that demand novel mechanisms for testing web apps. The chapter, then, discusses some existing techniques available for testing web apps.

To test software, testers must design *test cases* (sometimes referred to as *test inputs* or *tests*). Test cases are inputs entered to an app under test. These inputs include form data values and a series of interactions between a user and the app. Test cases may be created (i) randomly, (ii) based on the testers' experience, and (iii) according to software testing criteria. While randomly generating tests can be simple and source code of the app under test is not needed, the tests' ability to detect software faults varies tremendously depending on selected test inputs. Furthermore, the quality of tests are different and relies heavily on the testers' experience. On the other hand, software testing criteria provide testers a checklist (referred to as *test requirements*) describing how the tests should be and what should be covered while testing. Precisely, the quality of test requirements determines the quality of test cases and appropriate instructions (or criteria) are vital to derive high quality test requirements. Mutation testing has been found to be an extremely effective technique for producing test requirements.

2.1 Mutation Testing

Focusing on the use of mutation analysis to test web apps, this chapter presents a background on mutation analysis and discusses an overview of techniques that have been used to test web apps and that have been used to reduce the cost of mutation testing. The chapter does not intend to provide a comprehensive survey of all existing research in mutation testing. Instead, it provides the core concepts of the underlying theory applied in this

research.

Over four decades, mutation analysis has evolved and proven to be effective at revealing faults [9, 25, 42]. Precisely, mutation testing is a fault-based testing technique that can be used to generate test cases to be used to measure the effectiveness of pre-existing tests. Notwithstanding, it can be expensive due to the number of test requirements generated.

In the past few decades, mutation testing has been applied to many types of software artifacts, including programming languages (such as Fortran 77 [22], C [67], and Java [44,63, 65]), specifications and models [11,31,55], android apps [23], and web apps and web services [54,73,75,88,90,109]. Mutation testing has also been applied to non-software artifacts such as security policies [70] and spreadsheets [4]. An alternative use of mutation analysis is to help produce candidate patches in automated software repair [53]. Extensive information on the development of mutation testing can be found in Jia and Harman's survey [42].

2.1.1 An Overview of the Mutation Testing Process

The underlying concept of mutation testing is to create modified versions of the program by syntactically changing the program. A single change made to the program signifies a *first-order* mutant whereas multiple changes made to the program represent a *higher-order* mutant. It is important to note that this research intends to provide a testing criterion to both guide testers to detect certain kinds of web faults and to help developers to avoid certain kinds of mistakes. Hence, this research relies heftily on first-order mutation testing. The variations from first-order mutation testing intend to mimic common mistakes developers could have made or force testers to check whether the program behaves appropriately under certain circumstances. Then a test suite is executed on the modified versions. The more modified versions the test suite can distinguish from the original program, the more effective the test suite is.

The general process of mutation testing consists of (i) generating mutants, (ii) executing the app under tests, (iii) executing the mutants, and (iv) determining if the tests can detect mutants [6, 21].

Mutants are variants of the app under test, where each mutant differs from the original in a small syntactic way (usually one statement is changed) [6]. Mutants are test requirements that testers must design test cases to satisfy. Most mutants represent mistakes that a programmer could have made. Other mutants may encourage good tests, such as using boundary values. Some mutants may be semantically equivalent to the original app, and are called *equivalent* [81]. To generate mutants, rules specifying syntactic variations are applied to the original source code. These rules are called *mutation operators* (also known as *mutation rules* [81]).

Prior to evaluating the effectiveness of test suite, the tests must be designed and executed on the original app. If the tests fail (i.e., the app is incorrect), the app must be fixed. The testing and fixing process is repeated until all tests pass; that is, the app is correct with respect to the tests. After successfully running the tests on the original app, the tests are executed on the mutants.

To kill a mutant, the following three conditions must be satisfied [6, 22].

- *Reachability*: The mutated location in the program must be reached and thus executed.
- *Infection*: After the mutated location is executed, the state of program must be incorrect.
- *Propagation*: The infected state must affect some part of the program output.

These conditions forms the *RIP* model [6, 22], and help testers design tests that cause the output or behavior of the mutants to be different from the output or behavior of the original app. If a mutant behaves differently from the original app, the tests can detect the faults that the mutant represents and the mutant is said to be *killed*. Dead mutants are removed from the testing process. Tests are said to be *effective* at finding faults in the app if they distinguish the app from its mutants [63]. Mutants that cannot be compiled or executed because they are syntactically illegal are called *stillborn*. Stillborn mutants are not useful in revealing faults or evaluating the quality of test cases since no further execution

or analysis can be done. Thus, stillborn mutants are excluded from this research, and when possible not created. If mutants can be killed by almost any test cases, they are called *trivial* mutants. If mutants behave exactly the same as the original app on all inputs, they are said to be *equivalent*. Equivalent mutants always behave or produce the same output as the original app; thus, no test cases can kill them. Determining equivalent mutants is theoretically undecidable for some cases, and is usually done manually.

To measure the effectiveness of a test suite, the *mutation score* is computed as a percentage of the non-equivalent mutants that have been killed [6]. The mutation score ranges from 0 to 100%, where 100% indicates that all mutants have been killed and hence the test suite is adequate.

In addition to evaluating the effectiveness of a test suite, mutation testing helps improve the quality of tests by providing a test-adequacy criterion. To do so, testers add more tests and repeat the mutation testing process until achieving a mutation score of 100% or reaching a threshold for mutation score.

Figure 2.1 presents an overview of the mutation testing process. Mutants are created for a program under test `P`. A set of test cases (`T`) is designed and run on `P`. `P` is fixed until it is correct. `T` is run on mutants. More test cases are designed and added to `T`, which in turn is run on mutants until a desired threshold is achieved.

While mutation testing has been shown to be effective at designing high quality test cases, the testing costs can be very expensive depending on the number of mutants generated. More mutants means, in general, more tests. Mutation testing requires human effort in determining meaningful test inputs and identifying equivalent mutants. Furthermore, running a large number of mutants is computationally expensive. Several studies [5, 19, 24, 43, 50, 76, 77, 81] have confirmed the challenges in testing due to the extensive number of mutants. Many researchers have developed techniques to reduce the number of mutants while maintaining the fault detection capability by selectively applying effective mutation operators [19,24,76,77] and excluding redundant mutants from the testing process [5, 50]. However, these studies have not considered web-specific mutation operators.

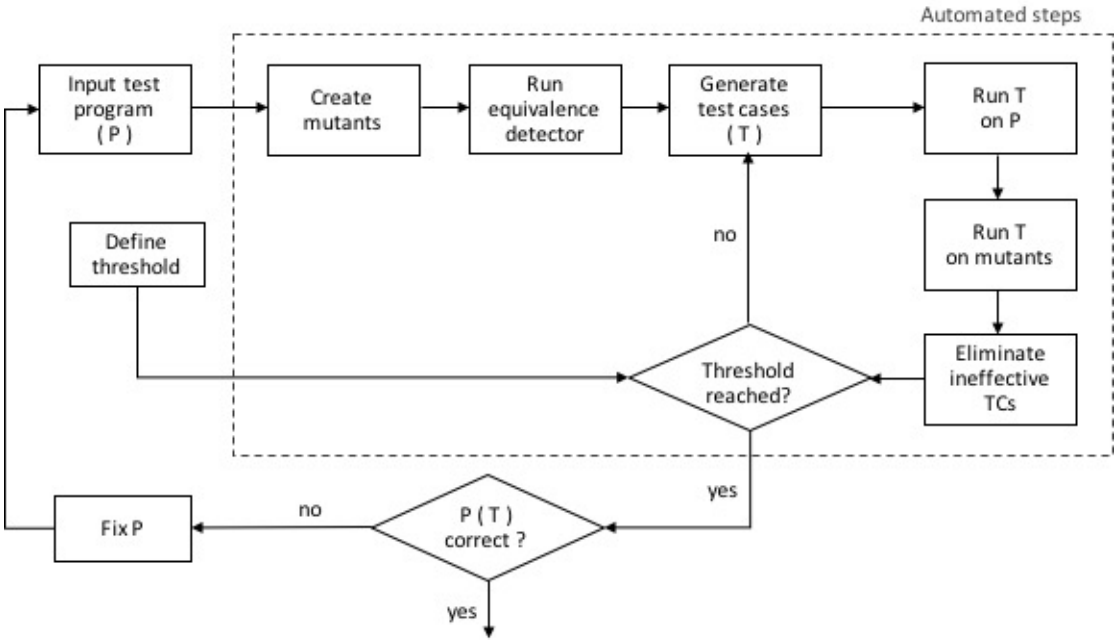

Figure 2.1: Mutation testing process [6][1]

As this research focuses on applying mutation testing to web apps, the number of mutants (which are test requirements) drive the testing cost. This research focuses on using effective mutation operators [5, 76, 77] to get fewer mutants, thereby reducing cost. To provide the foundation for analyzing redundancy in web mutation operators and identifying the effective operators, the next subsection discusses some background and an overview of approaches that have been used to reduce cost in mutation testing.

2.1.2 Cost Reduction Techniques

While mutation testing has been shown to be effective at helping testers create better quality tests, it can be computationally expensive due to the number of mutants. Several approaches have been proposed to reduce the cost of mutation testing. Offutt and Untch classified the approaches into three categories: *do-fewer*, *do-smarter*, and *do-faster* [81].

[1]Reproduced with permission from J. Offutt

Do-fewer approaches focus on running fewer mutants without sacrificing effectiveness. Do-smarter approaches emphasize distributing the computational expense; for instance, by executing mutants over several machines. Do-faster approaches concentrate on generating and running programs more quickly; for instance, using the mutant schema generation (MSG). The idea of MSG is to embed multiple mutants into each line of source code so that one source file contains all mutants [104]. Another example of do-faster is the use of MSG and Java reflection in muJava [65], which modifies Java bytecode to create mutants.

This research emphasizes the use of effective mutation operators to produce fewer mutants that lead to highly effective tests. For this reason, it follows the do-fewer approach.

Wong [108] randomly select mutants according to a uniform distribution. However, he reported that when the sampling rate was low enough to yield substantial savings, the results were weak and could not confirm the use of randomly selecting mutants. Later, instead of using the random approach, Wong and Mathur [107] suggested the idea of *selective mutation*, which uses only the most critical mutation operators. This became one of the early do-fewer approaches. Offutt, Rothermel, and Zapf [77] extended the selective mutation idea, which allows testers to perform approximate mutation testing. They demonstrated that reducing the number of mutants decreases the testing costs while providing coverage that is almost as strong as non-selective mutation. Later, Offutt et al. [76] empirically validated and recommended that only five Mothra mutation operators were sufficient and provided almost the same coverage as using all 22 Mothra mutation operators. The selective set of mutation operators (appropriately modified for Java) were implemented for Java in muJava [65].

Kaminski et al. [46] proposed to selectively generate only logic mutants. They showed that tests that weakly kill all logic mutants also strongly kill most general mutants and hence provide sufficient test coverage with less testing cost. Kaminski et al. [47] further showed that only three mutants out of the seven created by the *relational operator replacement* operator (ROR) are needed. They theoretically proved that tests that kill these three mutants are guaranteed to kill the remaining four mutants.

Untch suggested the use of a single *statement deletion* operator (SDL) [105]. He used regression analysis to demonstrated that the SDL operator can reduce the mutation testing cost by producing fewer mutants without significantly reducing fault detection. Deng et al. [24] examined the effectiveness of the SDL mutation operator for Java in comparison with other mutation operators implemented in muJava. Their experimental results confirmed that using only the SDL operator significantly reduces the number of mutants and produces few equivalent mutants. Tests that adequately kill all non-equivalent SDL mutants were created and then executed against the entire set of mutants. Deng et al. showed that tests designed specifically to kill SDL mutants can also kill other mutants. Their experiment inspired experimental design of this dissertation. The difference is that, in this dissertation, the experiment creates tests adequate to kill each type of mutant and executed these tests on all mutants. Deng et al. rely on mutation scores of the overall mutants but this dissertation consider the effectiveness of each test set on each type of mutants.

Delamaro et al. [19, 20] extended Deng et al. [24] study for programs written in C language. Delamaro et al. evaluated the effectiveness of using only the SDL operator and computed the cost-effectiveness by considering the number of tests needed and the number of equivalent mutants. They confirmed that using the SDL operator by itself leads to highly effective test sets. They concluded that the testing cost can be reduced considerably by using a single, powerful mutation operator.

Most recently, Ammann et al. [5] identified redundancy among mutants in an attempt to create a true minimal test set. They proposed excluding mutants that are redundant in the sense that they are guaranteed to be killed by a test that kills another mutant. They showed that, in theory, at least, approximately 90% of the muJava mutants and 99% of the Proteum [67] are redundant. Their on going research is attempting to achieve most of this potential savings through static and dynamic analysis of the mutants [50, 51].

2.2 Web Applications

Web apps are user interactive software apps that provide specific resources such as content and services, deployed onto a web server, and accessed through web browsers [15]. Figure 2.2 illustrates a general view of interactions between users and web apps. Once a client (a web user via a web browser or another web component) sends a request to a web app, the request is conveyed to a web server hosting the app. The web server analyzes the request then dispatches it to an appropriate web app software component on the server. A web app generates a response as an HTML document, which is then returned and rendered by the browser.

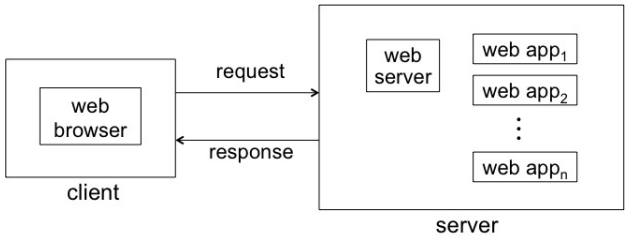

Figure 2.2: Interactions between users and web apps

To be more specific, web apps are composed of the front-end graphical user interfaces (GUIs) that are visible to users and the back-end web software components that provide services. In this research, each interface displayed to users is called a *screen*. Thus, all of the user's interaction with web apps are done through the screens. Web apps are typically developed by teams with diverse expertise that integrate diverse frameworks and web components [79].

Web components are modules that implement different parts of the web apps' functionality. Web components are independently compiled and executed software components that can be tested separately. They interact with each other to provide services to the organizations and users that operate the web apps. Web components may be generated using different software technologies such as Java Server Pages (JSPs), Java servlets, JavaScripts,

Active Server Pages (ASPs), PHP: Hypertext Preprocessor, and Asynchronous JavaScript and XML (AJAX). Web components may include static HTML files, and programs that dynamically generate HTML pages and forms with input fields. Web components may reside on different servers and are integrated dynamically.

Composed of diverse, distributed and dynamically integrated web components, web apps are heterogeneous. The appearance of web apps may vary depending upon users, time, and geography. Furthermore, the content of each screen may be customized according to data stored, the server state, or session variables at the moment the request is executed. As the user interface of a web apps is the screen rendering in a web browser, users may interact with the web apps using the browser controls (such as `back`, `forward`, and `reload` buttons) in addition to the controls provided by the web apps. This nondeterministic construction of web apps increases complexity and the difficulty of testing.

2.3 Web Modeling

This research focuses on applying mutation to test web apps; hence, understanding potential web faults is mandatory prior to defining mutation operators. Up until now, no standard web fault model is available. Though several attempts have been made to classify faults occurring in web apps [35,69,93], the existing categorizations overlap without being complete or consistent. Therefore, this research models web faults and hopes to ensure coverage of interaction faults. The fault model is later used to design web mutation operators. Discussion on the fault model is presented in Chapter 3.

To form a fault model, understanding of web faults is necessary. This research takes into account how web apps are modeled and how web components interact, and analyzes potential faults according to the seven challenges presented in Chapter 1.

Most web modeling approaches focus on representing static aspects of web apps [40, 91]. Though some attempt to capture dynamic aspects of web apps [13, 103], they specifically

focus on expressing the models using the Unified Modeling Language (UML) without considering potential transitions between web software components. Accordingly, to derive interaction faults in web apps, this research extended the types of transitions between web components presented by Offutt and Wu [82] as follows:

- **Simple Link Transition**: An invocation of an HTML <A> link causes a transition from the client to a web software component on the server. Simple link transitions are static. If there is more than one <A> link in an HTML document being considered, one of several web software components can be invoked.

- **Form Link Transition**: An invocation of an HTML <FORM> element causes a transition from the client to a web software component on the server. Form link transitions usually involve sending data to web software components that process the data. They are dynamic and data (or inputs) are required prior to the invocation. If there is more than one <FORM> element, one of several web software components can be invoked.

- **Component Expression Transition**: A component expression transition occurs when the execution of a web software component causes another component or a portion of HTML to be generated and returned to the client. The HTML contents are dynamically created and may vary depending on inputs. Not only do inputs impact the contents of the HTML, but some state of the apps or the server (for example, the user or session information, date an time, or geography) may also affect the HTML contents. In general, a web software component can produce several component expressions.

- **Operational Transition**: An operational transition is a transition that is caused by the client or system configuration. Examples of operational transitions are that the client presses the `back` button, presses the `forward` button, presses the `refresh` button, or directly alters the URL in the browser. Operational transitions also include situations when a particular screen of a web app is accessed via a bookmark (the

browser loads a screen from the cache rather than loading it from the server). Web apps have no control of this kind of transitions.

- **Redirect Transition**: A transition causes the client to regenerate the same request to a different URL. Redirect transitions go through the browser, but users are normally unaware of the redirection. This transition includes forwarding, redirecting, and including control connections between web software components.

- **Remote transition**: A remote transition occurs when a web app accesses web software components that reside in different locations (i.e., available at remote sites). The locations are usually available once the invocation is triggered. Hence, testing remote transition is difficult due to limited knowledge of the remote sites.

2.4 Web Application Testing

This section presents an overview of the current state-of-art in web app testing techniques and the techniques researchers used to validate their approaches. Existing testing techniques used for web apps are classified by how they derive tests into four groups: (i) model-based testing, (ii) mutation-based testing or syntax-based testing, (iii) input validation-based testing, and (iv) user-session-based testing. Techniques are presented in chronological order and are summarized in Tables 2.1 and 2.2.

2.4.1 Model-based Testing

Model-based testing techniques rely on the structural description of a web app. Common representations are graphs (including control flow graph and data flow graph) and finite state machines (FSMs). Nodes usually represent web components (in graphs) or state of the app (in finite state machines) and edges signify communications or transitions between web components. Other model representations for web apps are UMLs and formal specification languages. Based on these representations, some coverage criteria can be applied to test web apps.

Table 2.1: Critical research web app testing

Authors	Highlights / implications / pitfalls
Model-based testing techniques	
Kung et al. [49] and Liu et al. [59]	Used multiple models to represent interactions between web components; focused on data interactions; relied on static HTML documents
Ricca and Tonella [91]	Used a UML-based analysis model for test case generation; captured transitions based on static HTML links and sequences of URLs
Lucca et al. [61]	Extended path-based testing to represent data flow in web apps; did not consider internal states of web apps
Lucca and Penta[62]	Modeled interactions between web pages with UML statecharts; focused on the transitions caused by the browser `back` and `forward` buttons; did not consider internal states of web apps
Andrews et al. [8]	Modeled and tested web apps with finite state machines (FSMs); did not address how to handle dynamic aspects of web apps
Liu [60]	Annotated control flow graphs with def-use information to support JSP-based web app testing; did not considered operational transitions
Halfond and Orso [37]	Modeled web apps using control flow graphs; focused on parameter mismatches in communications between web components; assumed all paths were feasible; relied on Java source code
Halfond et al. [36]	Extended their control flow graph based testing; focused on specific kinds of faults due to parameter mismatches; assumed all paths were feasible; relied on Java source code
Andrews et al. [7]	Dealt with dynamic aspects by modeling web apps' states with hierarchical FSMs; reduced FSM state space explosion with input constraint annotation; modeled operational transitions with hierarchical FSMs could be expensive
Mesbah and Deursen [72]	Focused on AJAX-based web apps; modeled web apps with the state flow graph; dealt with broken links; considered the use of the browser `back` button; relied on the apps' invariants whose correctness and completeness were difficult to verified
Mutation-based testing techniques	
Lee and Offutt [54]	Applied mutation testing to XML-based data interactions in web apps; generated mutants of interaction data (not source code)
Mansour and Houri [68]	Focused on event features in .NET code; dealt with method and class levels, presentation level, and event level; did not consider state management nor the use of browser's features
Smith and Williams [98]	Used statement-level mutation operators; did not consider interactions between web components
Input validation-based testing techniques	
Offutt et al. [83, 84]	Introduced *bypass* testing; evaluated how web apps handled invalid inputs; focused on value-level, parameter-level and control flow-level; did not consider the use of browser's features nor state management and state scope handling
Tappenden et al. [101]	Applied the bypass testing concepts [83] to data stored in cookies and verified the files being uploaded; focused on security issues; did not considered challenges discussed in Section 1.1 nor provided empirical validation

Table 2.2: Critical research web app testing (continue)

Authors	Highlights / implications / pitfalls
Input validation-based testing techniques (continue)	
Papadimitriou et al. [85]	Extended the bypass testing concepts [83] and confirmed the feasibility of the concepts; did not consider the use of browser's features nor state management and state scope handling
Li et al. [56]	Altered the regular expression of valid inputs; focused on security issue relied on static analysis; did not consider state management and operational transitions
Mouelhi et al. [74]	Extended bypass testing and implemented an input validation on the server-side; focused on security issues; experimented on four small custom-built web apps; did not consider operational transitions, state management and state scope handling
Offutt et al. [80]	Refined and extended the bypass testing concepts; demonstrated the applicability using commercial web apps; did not consider the use of browser's features nor state management and state scope handling
User-session-based testing techniques	
Kallepalli and Tian [45] and Li and Tian [57]	Modeled the users' navigational patterns with Unified Markov Models analyzed faults and failures of web apps statistically; did not deal with dynamic aspects of web apps; relied heavily on static usage logs
Elbaum et al. [29]	Divided a user session into snapshots to derive test inputs; considered state of web app from each snapshot; restricted to users with similar usage profiles
Elbaum et al. [30]	Showed that user-session-based testing could be complementary to some existing white-box testing techniques; suffered from huge amount of logged data
Sampath et al. [95]	Used logged information to customize test requirements; focused on reducing the size of test suites; did not specify the kinds of web faults nor their classification
Sampath et al. [96]	Clustered logged user session to selectively generate test requirements; did not specify the kinds of web faults nor their classification
Sprenkle et al. [99]	Focused on reducing test suites by clustering logged usage information based on users' access privileges; relied heavily on the users' privilege definitions; restricted to web apps with similar definitions of users' privileges; did not specify the kinds web faults nor their classification

Among the earlier work in web testing, Kung et al. [49] and Liu et al. [59] presented a web app testing approach based on the object-oriented paradigm. Their approaches consisted of multiple models, each of which targeted a different level of the web apps. These models represented interactions between web components as control flow graphs. To support integration testing, the authors considered HTML documents as objects (i.e., web components). The graphs represent data flow interactions between HTML documents. The research's focus was on data interactions rather than control flow.

Although both models describe interactions between web components, constructing multiple models to represent the app's flow of execution can increase complexity and may result in a scalability problem. Moreover, the models representing the web apps are derived solely from source code, i.e., static or known interactions. There is no guarantee that interactions generated dynamically (while the app is running) are covered. Furthermore, the authors particularly concentrate on HTML documents, thereby possibly excluding other features in web apps such as state maintenance challenges.

Ricca and Tonella [91] proposed a UML-based analysis model to facilitate test case generation for static web pages. The model captures transitions based on HTML links and sequences of URLs. The authors applied several coverage criteria (page, hyperlink, def-use, all-uses, and all-paths) pertaining to the data dependences obtained from the models. Later, Ricca and Tonella [92] applied their UML-based model to support integration testing which took into account the states of web apps under test. However, this UML-based model representation of a web app is constructed from a static web page. Therefore, it seems uncertain how this approach could support dynamic validation including links or interactions as well as web components that are generated dynamically. Additionally, user's abilities to control execution flow via operational transitions are omitted in this work.

Lucca et al. [61] extended a traditional path-based test generation technique and applied data flow coverage to testing web apps. Later, Lucca and Penta incorporated operational transitions, specifically focusing on the transitions caused by pressing the `back` and the `forward` buttons [62]. They modeled interactions between web pages (i.e., state

transitions) with a UML statechart. Four states of the `back` and the `forward` buttons are defined: back-disabled-forward-disabled, back-enabled-forward-disabled, back-enabled-forward-enabled, and back-disabled-forward-enabled. This approach focused on revealing inconsistencies caused by the use of browser features. While both studies show some advantages for data flow web app testing, other challenges related to server-side and client-side state management are not addressed. Indeed, internal states of web apps are not taken into account.

Andrews et al. [8] developed a web app testing technique by modeling web apps with finite state machines (FSMs). By applying coverage criteria based on FSM test sequences, test requirements were derived as sequences of states. Then, test sequences were combined to generate test cases. The authors did not address how to handle dynamic aspects of web apps, such as transitions introduced by the users through the web browsers (i.e., operational transitions).

Liu [60] adapted traditional data flow testing techniques to support JSP-based web app testing. Control flow graphs were annotated with def-use information (of variables, implicit objects, and action tags of interest) to represent data interactions caused by the user's navigation paths. Three levels of data interactions were considered. Firstly, the intraprocedural data flow (the lower level model) signified interactions between statement blocks of a JSP. Secondly, the interprocedural data flow depicted interactions between functions or JSPs. Thirdly, the sessional data flow (the top level model) described interactions among JSPs introduced by a particular session object. This sessional data flow aggregated the other two data flow models. In general, a specific object, called a `session` object, was used to store information (parameter name-value pairs) of a user session. Different JSP pages within a user session shared and accessed this information for state management purpose. To deal with a user session, a control flow graph was annotated with def-use information corresponding to a particular session object. This approach does not address challenges due to operational transitions.

Communications among web components are implicit and hence information (i.e., a request with parameter-value pairs) sent from a component (a caller) and information (parameter-value pairs) that the target component (a callee) expects may be inconsistent. To try to detect this inconsistency, Halfond and Orso [37] introduced a static analysis technique to extract web app request parameters (i.e., sets of named input parameters and relevant or potential values) for Java source code. A year later, Halfond et al. [36] extended their work to detect a specific kind of faults that were due to mismatches of parameters used in communication between web components. Three kinds of mismatches included (i) missing parameters (a caller sending fewer parameters than a callee expecting), (ii) optional parameters (a caller sending more parameters than a callee expecting), and (iii) syntax errors (misspelling parameters' names or inappropriate formatting). Execution paths of web apps were represented with control flow graphs. All paths were assumed to be feasible. The graph was derived from source code. The authors considered these inconsistencies to be web faults regardless of the compatibility of data type or the corresponding values being transmitted or expected.

Later, Andrews et al. [7] extended their approach to deal with dynamic aspects and to reduce the state space explosion. They modeled the states of web apps by using hierarchical finite state machines (FSMs). The authors partitioned a web app into clusters, each of which implemented some logical functions and was represented with a FSM. Aggregated FSMs described entire web apps. Testing strategies dealt with how web components were connected and interacted. Test requirements were derived as sequences of states in the FSMs. Then, by integrating the test sequences, they generated test cases. To cope with the state space explosion issue, the FSMs were annotated with input constraints. This approach supports dynamically generated web components and deals with challenges related to state management. The authors suggested modeling operational transitions with hierarchical FSMs but it could be very expensive due to nondeterministic states of the apps.

Mesbah and Deursen [72] incorporated their AJAX-based web crawler [71] to test AJAX-based web apps. As the user interacted with the apps, the DOM tree was dynamically

updated. The crawler captured the states of the user interface as a result of changes in the DOM tree and represented them in a state flow graph. Test cases were derived from the state flow graph and were written in JUnit as the sequences of events from the initial state to a target state. Fault detection came from checking the output (HTML instance) after each state change against the apps' invariants. Broken links or URLs were taken into account. The authors also considered the inconsistency of the user interface (screen) when the browser `back` button was used. It is particularly useful to determine inconsistency though verification of the states against the apps' invariants. However, to ensure the correctness and completeness of the invariants might be challenging.

2.4.2 Mutation-based Testing

Existing web testing techniques pertaining to mutation analysis focus on mutating source code of web apps. Some researchers used mutation analysis to test web apps to govern information policies and for security purposes [70]. Up until now, there is limited empirical work that applies mutation analysis to test web apps.

Among earlier empirical work on mutation-based testing, Lee and Offutt [54] demonstrated the applicability of mutation testing to verify XML-based data interactions between individual pairs of web components. They introduced an interaction specification model using DTDs (Document Type Definitions) to describe interaction messages between web components. A set of mutation operators were defined to mutate the interaction specification and thereby to alter XML messages. Unlike this dissertation where source code was mutated, Lee and Offutt mutated XML messages being transmitted between web components. As a result, instead of creating a variation of source code, Lee and Offutt generated variations of interaction data. To determine whether the test cases detected these changes, test cases were generated iteratively where an initial set was derived from the original XML constraints. Additional test cases were generated with an attempt to kill all mutants. If a mutant produced different responses from the original version of interaction data, it was said to have failed (and marked dead). If a mutant produced the same responses from the

original version of interaction data, it was considered equivalent and was excluded from the testing. The iteration terminated if restrictions (such as time or budget limitations) applied.

Mansour and Houri [68] presented three groups of mutation operators to test .NET web apps. The focus was on event features in .NET code. The first group of operators, adopted from mutation operators used for testing traditional software presented by Kim et al. [48], dealt with method and class levels. The second set of operators mutated the presentation level of web apps, i.e., URIs or contents of the HTML documents. Thirdly, the event-level mutation operators tested interactions between web components. This was to ensure that the effect of the triggered event was implemented correctly. For instance, the operator removed hyperlinks, replaced the name of a method call, or deleted a line of code that implemented a transaction. This approach neither addresses how it would handle state management issues nor mentions challenges due to browser's features.

Smith and Williams [98] conducted an empirical study to evaluate the effectiveness of using mutation analysis to augment test cases. They applied mutation testing to a healthcare web app at the unit level, using the Jumble mutation tool (a class level mutation testing tool that mutates Java Byte Code (`http://jumble.sourceforge.net/`). The authors evaluated the applicability of mutation testing to web apps with traditional (statement-level) mutation operators, including mutation operators to negate conditions, replace binary arithmetic operators, replace increment with decrement, replace assignment values, alter return values, and modify the case in switch statements. Despite that the experiment shows effectiveness of mutation analysis to web apps, the authors' focus tends to be mainly on unit testing. Neither interactions nor transitions between web components are addressed.

2.4.3 Input Validation-based Testing

Most web apps' control flow are governed by the users' interactions through a web browser. Inappropriate data entry can result in data corruption, security vulnerabilities, or web app failures. Hence, to avoid or minimize these unexpected behavior of web apps, web inputs

must be validated [83].

Input validation ensures that data entered by the web users enter are appropriate and can be processed by web apps. For instance, an email address must contain an "@" sign; credit card information must be a combination of a credit card number, an expiration date, and a security code; a bank account consists of a routing number and an account number; and all required form fields are entered and they are of the correct types.

Several researchers have attempted to propose web app testing with a focus to ensure valid data and to control users' interactions with the software interfaces. Some techniques perform validation on the client while some run on the server.

Offutt et al. [83, 84] are among the pioneering researchers in input validation-based testing. They introduced an input validation technique called *bypass testing*. The idea was to submit invalid inputs directly to the server by bypassing client-side validation to evaluate whether a web app sufficiently checked these invalid data. It went further by also creating data that should be invalid but that may not be checked. To generate invalid inputs (i.e., test cases), the authors defined rules to violate constraints (HTML constraints and scripting constraints) that were applied to web apps under test. The authors divided bypass testing into three levels: value-level, parameter-level, and control flow-level. The value-level attempts to check whether a web app adequately evaluated invalid inputs (including restrictions on data types, value boundaries, and formats). The parameter-level verified whether related values or constraints meets the app's requirements. Because a web user had control over the app's control flow, the control flow-level intended to verify the app when a flow of execution was broken.

The original bypass testing rules [83] have been modified and additional rules have been added to address other HTML features used in real world, commercial web apps [80, 85].

Adopting the concept of bypass testing [83], Tappenden et al. [101] applied it to data stored in cookies. They also considered whether the names of files being uploaded to the web app were too long as well as whether the types of files were appropriate. Their main concern was to detect faults with a focus on security issues. It is unclear how their

extension deals with challenges discussed in Section 1.1 such as target URLs, state scopes of web components, and users' capability to control the app's execution flow. No rules or guidelines for testing are given explicitly and no empirical validation is provided.

Papadimitriou et al. [85] extended the bypass testing rules [83] and conducted empirical validation. His feasibility study proves that bypass testing can help reveal failures in numerous commercial web apps. To facilitate bypass testing, a prototype tool called AutoBypass was implemented to accept a URL to a web app under test and to automatically generate test cases.

Li et al. [56] generated invalid test inputs by perturbing valid inputs with an emphasis on security issues. They proposed six rules to alter the regular expression of valid inputs; (i) removing the mandatory sets from an expression, (ii) reordering the sequence of sets, (iii) changing the repetition time of selecting elements, (iv) selecting elements next to the boundary of the input domain, (v) inserting invalid characters into an expression, and (vi) inserting special patterns into an expression. Although this approach is complementary to testing web apps, it relies on static analysis and does not address the challenges due to state management and operational transitions.

Mouelhi et al. [74] addressed the problem that user input validation on the client was not adequate at preventing security attacks on web apps. Hackers may modify HTML and scripting code to bypass the client-side input validation. Malicious data may be sent to the server directly. The authors extended the concept of bypass testing [83]. However, unlike the original bypass testing that creates tests to validate inputs to web apps, they create an input validation software on the server-side to duplicate the input validation. They focused solely on the security issues and validated their tool using four small custom-built web apps. They did not consider operational transitions nor addressed the challenges due to state management and state scope handling.

Offutt et al. [80] demonstrated the applicability of bypass testing to web applications in practice. They refined and extended the bypass testing concept and tested widely used commercial web apps.

Offutt et al. [80, 83] and Papadimitriou [85] cover many of the challenges discussed in Section 1.1. Even though issues related to state management and state scope of web components are not addressed, bypass testing shows feasibility with additional rules.

This dissertation adopts several ideas from bypass testing [83] to define web-specific mutation operators. Note that this dissertation attempts to mimic potential faults (via mutants) and evaluate effectiveness of test cases but bypass testing intends to help design invalid inputs to evaluate adequacy of a web app's input validation. Another difference to note is that this dissertation requires source code of web apps under test (i.e., *white-box testing*) while bypass testing does not (i.e., *black-box testing*).

2.4.4 User-Session-based Testing

User-session-based testing approaches extract usage information from previously recorded users' sessions to model a web application and generate test cases. The key idea of using the logged information is to ensure that test cases are derived from real users' behavior (including how users navigate and interact with web apps). Information about each user session contains a sequence of a user's requests, which in turn indicates the base requests. The base requests, which are the request types and the target URLs to which the requests are sent, are used for test case generation.

Kallepalli and Tian [45] and Li and Tian [57] presented alternative use of user session information. They transformed logged usage information into Unified Markov Models (UMMs) that represented the users' navigational patterns. Then, both sets of authors applied statistical analysis to identify faults and failures as well as to evaluate the reliability of the app under test. It is unclear what kinds of web faults being considered. Furthermore, this approach suffers from the fact that information used to analyze and model the usage of a web app depends solely on web logs. Only parts of the app interactions were observed. No internal states were taken into consideration. It does not provide any guarantees of completeness nor indications for how to improve the test quality.

Elbaum et al. [29] presented a user-session-based testing approach by transforming logged user's requests into HTTP requests. The HTTP requests (i.e., test cases) contained the request types and the URLs along with additional corresponding information (parameter name-value pairs). The authors chose test inputs by considering the user data captured from HTML forms along with data from the previous user sessions. Elbaum et al. handled faults related to the states of web apps by breaking down logged user session information into snapshots. State-related values from each snapshot were considered to determine state changes. While this comparison may potentially indicate source of failures, the analysis only applies to users with similar operational profiles. Indeed, it is unclear how their technique may be generalized when other domain apps are considered.

Later, Elbaum et al. [30] demonstrated that user-session-based testing could be complementary to some existing white-box testing techniques. However, there appeared tradeoffs. While more user sessions could improve the fault coverage, maintaining and analyzing a large amount of logged information was crucial and costly. Elbaum et al. [28] reduced the number of test cases needed by applying constraints on the input parameters of the HTML form.

Sampath et al. [95] introduced a strategy to customize test requirements with a focus on reducing test suites. They illustrated that constructing test requirements solely from the base requests was not effective. This was because the associated data that were omitted might affect the app's execution flow. Indeed, test requirements that captured associated data (parameter names and values) resulted in test cases that revealed more faults than test cases derived from the base requests. It is unclear what kinds of web faults being considered and how the faults are classified.

The effectiveness of user-session-based testing depends primarily on the quality of usage data. However, collecting, maintaining, and analyzing large amount of user-session data can be very costly and increase the number of test cases. To reduce the testing cost while maintaining fault coverage, Sampath et al. [96] clustered logged user sessions based on concept analysis with a consideration of base requests and common subsequence of base

requests. Then, they applied test selection strategies to the clustered information. Their experiment revealed that decreasing the size of test suites lower fault detection capability. Different test selection strategies resulted in tradeoffs between the number of tests and the fault coverage. Sampath et al. recommended that data associated with the request and the sequences of requests should be taken into account when clustering the user sessions.

In another attempt to reduce the size of user sessions, Sprenkle et al. [99] proposed to cluster the logged usage information based on users' access privileges. The access privileges presumably reflected how the users navigated the apps. Rather than creating a navigation model from each user session, only one navigation model was needed to represent the usage pattern for each group. However, there were issues due to representativeness of the information used to derive tests. Also, the quality of tests depended significantly on the users' privilege definitions. Classifying user sessions based on privileges may not be applicable to some app domains; for example, web apps that do not require registration or authorization prior to access or web apps with vague definitions of access privileges. It may reduce the number of test cases needed but more information is required prior to classification.

One limitation of user-session-based testing is that it mainly relies on the usage data from previous sessions. User interactions with web apps may be subjective and be specific to certain tasks. Hence, there is no guaranteed coverage of the input domain and no systematic exploration of domain inputs. Though there is a possibility that unexpected interactions may be collected in the usage data (for example, the user pressed the `back` button), the logged data will probably not cover many unintended flow of executions. Unlike user-session-based testing research, this dissertation intends to provides systematic exploration of unintended execution flow through the use of web mutation operators.

Chapter 3: A Web Fault Categorization

Well designed mutation operators are based on realistic faults. The effectiveness of mutation testing depends primarily on the mutation operators. Hence, understanding potential web faults is mandatory prior to designing web-specific mutation operators.

Several researchers have attempted to classify faults in web apps [26,35,69,93]. However, up until now, no standard or agreement on web fault models is available. The existing categorizations overlap without being complete or consistent. Furthermore, they do not particularly consider interaction faults such as faults that are due to the use of web browser's features (`back` and `forward`) or faults that are caused by improper use of control connections (`forward` and `redirect`). Therefore, in the absence of a widely accepted fault model, this dissertation uses a structural approach based on web modeling theory as presented in Section 2.3. To accomplish this, the nature of web apps and how web development technologies can introduce faults and how the technologies affect the testing process were investigated. Related faults from existing models [26, 35, 69, 93], and information from online bug reports were also considered. Faults from existing models that are irrelevant to interaction faults were excluded; for instance, faults associated with arithmetic calculation problems. Web faults were modeled to ensure coverage of faults that may occur in communications between web components (i.e., *interaction faults*).

The fault model used in this dissertation is categorized into seven groups with respect to the challenges from Section 1.1. It is important to note that though there exist many web development frameworks and languages, due to the availability of web apps used to validate web mutation testing, the faults listed here are J2EE-specific. The underlying ideas could be adapted for faults in other frameworks or languages such as .Net or PHP with modest changes. Focusing on server-side web apps, faults related to JavaScript and AJAX are out

Table 3.1: Summary of web faults

Challenges	Potential faults
1. Users' ability to control web apps	Unintended transitions caused by the user via a web browser or intentionally bypass the app validation
2. Identifying web resources with URLs	Incorrect or inappropriate URLs
3. Communication depending on HTTP requests	Incorrect transfer mode (GET vs POST)
4. Communication via data exchanges	Mismatched parameters
5. Novel control connections	Incorrect use between redirect and forward transitions (for Java servlets)
6. Server-side state management	Incorrect scope setting
	Not initializing a session object when it should be
	Omitting necessary session info
7. Client-side state management	Omitting necessary info about hidden form fields
	Submitting incorrect info about hidden form fields
	Omitting necessary read-only info

of scope. A summary of potential faults is presented in Table 3.1. This fault model is later used to design web mutation operators in Chapter 4.

1. **Faults introduced by the users' ability to control web apps:** Web browsers allow users to circumvent normal execution flow of a web app through the web browser `back`, `forward`, and `refresh` (or `reload`) buttons. Also, users may re-enter the URLs directly or bookmark a URL for future visit. Failures may occur when the users use these browser features. Faults triggered by the use of the browser features can either propagate (resulting in noticeably unexpected behaviors or failures of web apps) or impact the internal states of web apps. Faults that propagate can be detected by simply evaluating the apps' behaviors or the outputs (i.e., HTML representations or responses). However, although these faults can be easily detected, testers are frequently unaware of them. On the other hand, faults related to internal states are not easily detected. Simply comparing the outputs is not a good indication. The app may still behave the same until a certain change to the app's state is triggered. Hence, testers need to examine all possible data and state of the app and then design tests that will trigger the changes of the states.

2. **Faults related to identifying web resources with URLs:** In web apps, all resources can be accessed by identifying URLs. J2EE allows web developers to specify web resources through several features, such as the `href` attribute of an <A> tag, the `action` attribute of a <form> tag, the `sendRedirect()` method of a `HttpServletResponse` object, the `file` attribute of an `include` directive, and the `page` attribute of a JSP `include` action and a JSP `forward` action. Faults are often due to the use of incorrect or non-existent URLs.

3. **Faults due to communication between web components through HTTP requests:** Form data can be conveyed using several transfer modes specified via the `method` attribute of a <form> element. Faults may occur when an inappropriate HTTP transfer mode is specified. This research considers faults that are due to the most commonly used HTTP transfer modes, `GET` and `POST`.

   `GET` and `POST` modes package data to be conveyed to the server differently. In general, a `GET` mode is used to request data from a specified URL (i.e., a web resource) while a `POST` mode is used to submit data to be processed to a specified URL. Faults are improper use of the transfer modes.

   Using a `GET` instead of a `POST` for sensitive data can reveal confidential information and possibly lead to unauthorized access or misuse of information. Moreover, for large form data, some of the data may be lost due to the size restriction of a `GET` method. Using a `POST` instead of a `GET` hinders the ability to bookmark and thus can affect the usability of web apps.

4. **Faults in communication via data exchanges:** Interactions between web components usually involve data exchanges in the form of parameter-value pairs. Potential faults in data exchanges are mismatches of these parameters. If the expected information and the information actually received are inconsistent (for instance, the number of parameter-value pairs is wrong, value types of parameters are incorrect, and parameters' names are inaccurate), errors may occur. Mismatched parameters may cause

unexpected behaviors of web apps, data anomalies, and web app failures.

5. **Faults introduced by misuse of novel control connections:** Web apps use several new control connections that do not exist in traditional software development. The general idea of using control connections is to transfer control flow from one web component (the *source*) to another web component (the *target*). Once the target web component finishes executing, some control connections (such as J2EE `include` connections) cause the control to return to the original web component while some do not (such as J2EE `forward` and `redirect` connections). HTML allows redirect transitions by using an `HTTP-EQUIV` attribute and specifying a forward destination using a `URL` attribute of a <`META`> command. Java servlets implement redirect transitions with the `sendRedirect()` method from a `HttpServletResponse` object and implement forward transitions with the `forward()` method of a `RequestDispatcher` object. Faults are misuse of control connections; for instance, using a forward transition instead of a redirect transition when a target web component is on a different server or when a request affects an internal state of web app or makes an update to a database.

6. **Faults due to server-side state management:** To achieve certain tasks, communications between a web user and a web app usually involves a series of related interactions. To maintain the state, persistent data are stored in an object (called a *session* object in J2EE) and its accessibility is identified via its scope. Different scopes determine different accessibility. For example, in J2EE, a `request` scope object is accessible by the component initially receiving the request and by other web components used or referred to in the same request, but a `page` scope object is only accessible within the component initially receiving the request. Potential faults are inappropriate scope setting of the object.

7. **Faults related to client-side state management:** Because of the stateless property of the HTTP, several mechanisms are used to maintain state information of web

apps. One common client-side state management technique is to store data in *hidden form fields*, which are sent to the server with the next request. Faults are omissions of necessary information and submissions of inappropriate information via hidden controls.

Other kinds of web faults relate to the use of scripting languages (such as JavaScript) and AJAX. Web apps use scripting languages to enhance their functionality. JavaScript allows developers to write function calls so that certain executions (e.g., input validation) may be performed. Potential faults are the use of incorrect or unavailable functions and mismatches of parameters. Removing a transition from the original web component to a target component skips an intended execution. For example, if a statement that calls another component (or a function in JavaScript) to filter a search result is deleted from the code, the search result may be processed and displayed unexpectedly.

Since this research focuses on server-side web apps, JavaScript and AJAX related faults are excluded for our fault model.

Chapter 4: Mutation Testing for Web Applications

This chapter presents an overview of web mutation testing and a new collection of web mutation operators targeting faults that can occur in interactions between web components.

4.1 Web Mutation Testing Process

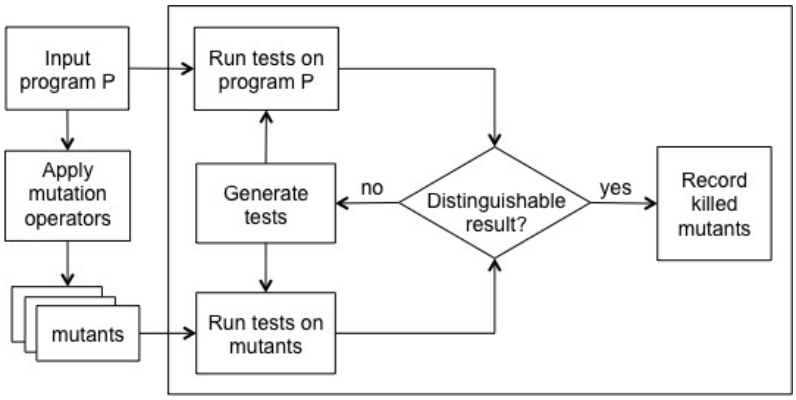

Figure 4.1: Overview of a web mutation testing process

The underlying concept of web mutation testing is based on mutation analysis, as described in Section 2.1. Figure 4.1 illustrates an overview of a web mutation testing process. Web mutation operators are applied to the server-side source code of a web app under test to generate web mutants. A set of tests is designed and executed on the original version of the web app and on the mutants. The outputs (in this research, HTML responses) of running the tests on the original version of the app and of running the tests on the mutants are compared to determine if the tests can distinguish the outputs. If the tests detect the differences, the mutants are said to be killed. Otherwise, more tests are generated and repeatedly executed to identify the differences between the outputs until all non-equivalent mutants are killed or the mutation scores reach a preferred threshold.

4.2 Web Mutation Operators

This section presents fifteen new source-code, first-order, mutation operators for web apps. Table 4.1 summarizes the web mutation operators, categorized according to the seven challenges from Section 1.1. These operators are designed to help testers to create tests that examine interactions between web components. The main emphasis is on mimicking faults that can occur in the transitions.

As mentioned earlier, this research focuses on server-side web apps. JavaScript and AJAX are out of scope.

Although many web development frameworks exist, the mutation operators designed in this research were implemented to test J2EE JSPs and Java servlets. This is due to several reasons. First, mutation testing requires source code to be available but the availability of web apps used as subjects in experiments is limited. Second, the J2EE platform has been widely used and has proven to provide various benefits. For instance, J2EE technology simplifies web development by providing infrastructures for developing web components, for managing communications between web components, and for handling sessions of the apps. Many organizations use the J2EE framework and have created many web apps and components. Additionally, J2EE web apps can be integrated in a loosely coupled, asynchronous way. Based on the Top Programming Languages 2016 survey[1], J2EE is one of high demand programming languages for web apps. Therefore, in the following section, examples illustrating the mutation operators are based on the implementation of JSPs and Java servlets.

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.

[1]IEEE Spectrum, http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016

Table 4.1: Summary of web faults and web mutation operators

Challenges	**Potential faults**	**Web mutation operators**	
1. Users' ability to control web applications	Unintended transitions caused by the user via a web browser or intentionally bypass the application validation	FOB	Insert a dummy URL to the browser history before the current URL (or screen)
2. Identifying web resources with URLs	Incorrect or inappropriate URLs	WLUR	Replace URL of `href` attr of an <A> tag
		WLUD	Remove URL of `href` attr of an <A> tag
		WFUR	Replace URL of `action` attr of a <form> tag
		WRUR	Replace URL of `sendRedirect` method of an `HttpServletResponse`
		WCUR	Replace URL of `page` attr of JSP forward action and JSP include action, and `file` attr of JSP include directive
3. Communication depending on HTTP requests	Incorrect transfer mode (GET vs POST)	WFTR	Replace a transfer mode with another transfer mode
4. Communication via data exchanges	Mismatched parameters	WPVD	Remove a parameter-value pair from JSP include action
5. Novel control connections	Incorrect use between redirect and forward transitions (for Java servlets)	WCTR	Replace a redirect transition with a forward transition and a forward transition with a redirect transition
6. Server-side state management	Incorrect scope setting	WSCR	Replace scope of <jsp:useBean> with another setting (page, request, session, application)
	Not initializing a session object when it should be	WSIR	Change a session object initialization to the opposite behavior (i.e., whether to create an instance)
	Omitting necessary session info	WSAD	Remove `setAttribute` method of a `session` object
7. Client-side state management	Omitting necessary info about hidden form fields	WHID	Remove a hidden form field
	Submitting incorrect info about hidden form fields	WHIR	Replace value of a hidden form field with another value in the same application domain
	Omitting necessary read-only info	WOID	Remove a read-only input control

4.2.1 Operator for Faults due to Users' Ability to Control Web Apps

For simplicity, an exception on a naming convention is applied to the failOnBack operator, whose abbreviation is derived straightforwardly as "FOB".

failOnBack (FOB): The FOB operator mutates the browser history by inserting a dummy URL into the browser history before the current URL is loaded (an `onload` event). This history manipulation creates a reference to an incorrect URL when the browser `back` button is clicked, rather than navigating to the previously viewed screen. FOB mutants can be killed by tests that include pressing the browser `back` button. While FOB mutants are not particularly hard to kill, they force testers to try the `back` button. Testers frequently overlook testing the `back` button and many testers are unaware that clicking the browser `back` button is an input to the web apps. Indeed, using any web browser features (e.g., pressing the `forward` or the `reload` buttons) are inputs to the apps under tests. The purpose of this operator is to ensure that the web apps properly handle the situation when the browser `back` button is pressed.

Original Code △ **FOB Mutant**

```
<html>                          <html>
  ...                             ...
  <body>                          <body onload="manipulatehistory()">
   ...                              <script src="failOnBack.js"></script>
   ...                             ...
</html>                         </html>
```

The following JavaScript code manipulates the browser history and is used by the FOB operator.

failOnBack.js

```
function manipulatehistory()
{
   var currentpage = window.document.toString();
   var currenturl = window.location.href;
   var pageData = window.document.toString();

   // add a dummy url right before the current url
   history.replaceState(pageData, "dummyurl", "failonback.html");
   history.pushState(currentpage, "currenturl", currenturl);
}

// update the page content
window.addEventListener('popstate', function(event) {
   window.location.reload();
});
```

Note on the failOnForward operator

Conceptually, the idea of the FOB operator can be applied to define a web mutation operator that checks against the use of the browser `forward` button (referred to as failOnForward operator).

The failOnForward operator inserts a dummy URL into the browser history right after the current URL, i.e., at the top of the browser history stack. Thus, when the browser `forward` button is clicked, this history manipulation causes a reference to an incorrect URL instead of navigating to the next screen (as specified by the URL at the top of the browser history stack). To kill a failOnForward mutant, test cases must contain a series of navigation between screens and must include pressing the browser `forward` button.

However, technical difficulties appeared when designing a mutation operator for forward transitions. This is due to how the browser history manipulation methods work. To be specific, two methods to manipulate the browser history, which are available in HTML5, are the `replaceState()` and `pushState()` methods of the `history`[2] object.

The `replaceState()` method replaces the current URL with a new URL (let URL′ denotes the new URL) in the browser history stack and the browser address bar, but does

[2]The `history` object contains the URLs visited within a browser.

not cause the browser to load the given URL′. It is important to note that when experimenting with the `replaceState()` method using the actual web browsers (including Firefox and Safari), the browser's address bar changes as described. Pressing the browser `reload` button causes the browser to load the URL′ and render its content on the screen. When experimenting with the `replaceState()` method using a virtual browser (or a simulated browser) in Eclipse[3], the address bar does not change. Pressing the `reload` button causes the URL′ to be loaded and its content is rendered on the screen. The inconsistent behavior is because the `replaceState()` method is relatively new in HTML5. The actual web browsers support it whereas the virtual browser does not.

The `pushState()` method adds the new URL (let URL′ denotes the new URL) at the top of the browser history stack, does not change the URL in the browser address bar, and does not cause the browser to load the given URL′. The method causes the current URL of the browser to be set to the URL specified at the top of the browser history stack (which is now URL′). Therefore, the `href` property of the `location`[4] object, which specifies the current URL of the browser, is set to URL′. The browser `back` button becomes enabled but the `forward` button is disabled, as there are multiple URLs in the history stack and the stack pointer points to the top of the stack. It is important to note that when experimenting with the `pushState()` method using the actual web browsers (Firefox and Safari), the method causes the behavior as described. This is because the actual web browsers support the `pushState()` method which is relatively new in HTML5. On the other hand, Eclipse's virtual browser does not properly support the method. When experimenting with the `pushState()` method using the virtual browser that runs on Java 1.7, the `pushState()` method causes an `UnsupportedOperationException`. When using the virtual browser that runs on Java 1.8, the URL′ is added at the top of the browser history stack, the address bar does not change, and the `back` button is disabled.

[3]In this research, web mutation operators were implemented using Eclipse Java EE IDE for Web Developers, Kepler Service Release 2.

[4]The `location` object contains information about the current URL. The `location` object is part of the `window` object, which represents an open window in a browser.

With the `pushState()` method, the browser's current URL is always set to the URL specified at the top of the browser history stack; hence the browser `forward` button is always disabled. The failOnForward operator intends to mutate the program such that the `forward` button is enabled to force testers to press it. For these reasons, it is impossible to implement the failOnForward operator, thereby excluded from this research.

Note on the failOnReload operator

Upon completion of the experiments (as presented in Section 5), it appears that a web mutation operator that checks against the use of the browser `reload` button is needed. Thus, an additional web mutation operator, named failOnReload, was designed and implemented. Although, it has not been empirically validated, its implication potentially follows the same direction as the failOnBack operator as they both are the browser's features. Future plan on validating this operator is later discussed in Section 6.

For simplicity, an exception on a naming convention is applied to the failOnReload operator, whose abbreviation is derived straightforwardly as "FOR".

failOnReload (FOR): The FOR operator mutates the browser history by replacing the current URL in the browser history with a dummy URL before the current URL is loaded (an `onload` event). This history manipulation creates a reference to an incorrect URL. When the browser `reload` button is clicked, the browser loads an incorrect URL rather than reloading the current screen. FOR mutants can be killed by tests that include pressing the browser `reload` button. While FOR mutants are not particularly hard to kill, they force testers to try the `reload` button. Testers frequently overlook testing the `reload` button and many testers are unaware that clicking the browser `reload` button is an input to the web apps. The purpose of this operator is to ensure that the web apps properly handle the situation when the browser `reload` button is pressed.

Original Code

```
<html>
  ...
  <body>
   ...
   ...
</html>
```

△

FOR Mutant

```
<html>
  ...
  <body onload="manipulatehistory()">
    <script src="failOnReload.js"></script>
    ...
</html>
```

The following JavaScript code manipulates the browser history and is used by the FOR operator.

failOnReload.js

```
function manipulatehistory()
{
   var pageData = window.document.toString();

   // replace the current url with a dummy url
   history.replaceState(pageData, "dummyurl", "failonreload.html");
}

// update the page content
window.addEventListener('popstate', function(event) {
   window.location.reload();
});
```

4.2.2 Operators for Faults due to Identifying Web Resources with URLs

In addition to the naming convention mentioned earlier, the second letter of the operator names indicates whether the operators deal with simple links ("L"), form links ("F"), redirection transitions ("R"), or control connections ("C"). The third letter of the name ("U") indicates that the operators mutate URLs. To be previse, these operators modify URLs in servlets and JSPs.

Simple link replacement (WLUR): The WLUR operator replaces a destination of a simple link transition `<A>` with another destination in the same domain of the web app under test. The change causes a reference to an incorrect or a nonexistent destination, and possibly leads to dead code. WLUR mutants can be killed by tests that include clicking

the links. While this mutant is not particularly difficult to kill, testers often focus on the main functionality (or "happy path") and overlook the links that are not part of the main functionality of the web apps under test. This operator ensures that all links are tried during testing.

Original Code		**WLUR Mutant**
<html>		<html>
...		...
<A href=URL$_1$>...	△	<A href=URL$_2$>...
<A href=URL$_2$>...		<A href=URL$_2$>...
...		...
</html>		</html>

Simple link deletion (WLUD): The WLUD operator removes the destination of a simple link transition `<A>`, hence breaking the normal execution flow. Similar to the WLUR operator, the objective of this operator is to ensure that the destination is specified properly.

Original Code		**WLUD Mutant**
<html>		<html>
...		...
<A href=URL$_1$>...	△	<A href="">...
<A href=URL$_2$>...		<A href=URL$_2$>...
...		...
</html>		</html>

Form link replacement (WFUR): The WFUR operator alters the destination of a form link transition `<form>` to another destination in the same domain of the web app under test. The modification causes a reference to a destination that does not exist or to a destination that cannot process the request. WFUR mutants can be killed by tests that submit a form with the inputs that impact the response of the request. The purpose of this operator is to guide testers to design test inputs to ensure that form submission is appropriately handled.

Original Code

```
<html>
  ...
  <form action=URL1 method="POST">
  ...
   <input type="text" name="pname1" ...
   <input type="text" name="pname2" ...
   <input type="text" name="pname3" ...
  ...
  </form>
</html>
```

△

WFUR Mutant

```
<html>
  ...
  <form action=URL2 method="POST">
  ...
   <input type="text" name="pname1" ...
   <input type="text" name="pname2" ...
   <input type="text" name="pname3" ...
  ...
  </form>
</html>
```

Redirect transition replacement (WRUR): The WRUR operator changes the destination of a redirect transition that is specified in a `sendRedirect()` method of a `HttpServletResponse` object to another destination in the same domain of the web app under test. Again, the change causes a reference to an incorrect or a nonexistent destination. WRUR mutants can be killed by tests that submit requests to the URL that triggers the `sendRedirect` method. This operator helps testers verify the correctness of the destination of the redirect transitions.

Original Code

```
public class logout extends HttpServlet
{
  ...
  public void doGet(...  )
  {
    ...
    response.sendRedirect(URL1);
  }
}
```

△

WRUR Mutant

```
public class logout extends HttpServlet
{
  ...
  public void doGet(...  )
  {
    ...
    response.sendRedirect(URL2);
  }
}
```

Control connection replacement (WCUR): As discussed in Section 1.1, web software technologies use novel control connections. Web components can be imported into other web components through a server-side include directive, JSP `include` directives, and JSP `include` actions. Control can also be handed off (without a return) through JSP `forward` actions. The WCUR operator changes the target web component of forward

and include connections to other components in the same domain of the web app under test. This operator guides testers to design test inputs to ensure that the web component's destination is specified properly.

The following example shows a WCUR mutant that modifies a `file` attribute of a server-side include directive.

Original Code

```
<html>
  ...
  <--#include file=URL1 -->
  ...
  <A href=URL2>...
  ...
</html>
```

△

WCUR Mutant

```
<html>
  ...
  <--#include file=URL2 -->
  ...
  <A href=URL2>...
  ...
</html>
```

The following example shows a WCUR mutant that replaces the destination of a `file` attribute of a JSP `include` directive with another destination.

Original Code

```
<html>
  ...
  <%@ include file=URL1 %>
  ...
  <A href=URL2>...
  ...
</html>
```

△

WCUR Mutant

```
<html>
  ...
  <%@ include file=URL2 %>
  ...
  <A href=URL2>...
  ...
</html>
```

The following example shows a WCUR mutant that changes the destination of a `page` attribute of a JSP `include` action to another destination.

Original Code

```
<html>
  ...
  <jsp:include page=URL1 />
  ...
  <A href=URL2>...
  ...
</html>
```

△

WCUR Mutant

```
<html>
  ...
  <jsp:include page=URL2 />
  ...
  <A href=URL2>...
  ...
</html>
```

The following example shows a WCUR mutant that modifies the destination of a forward transition specified in `<jsp:forward>` with another destination.

Original Code

```
<html>
  ...
  <jsp:forward page=URL1 />
  ...
  <A href=URL2>...
  ...
</html>
```

△

WCUR Mutant

```
<html>
  ...
  <jsp:forward page=URL2 />
  ...
  <A href=URL2>...
  ...
</html>
```

WCUR mutants can be killed by tests that include submitting requests to the URLs that trigger the control connections. The intent of this operator is to guide testers to design test inputs to ensure that the web component's destination is specified appropriately.

4.2.3 Operator for Faults due to Invalid HTTP Requests

In addition to the naming convention mentioned earlier, the second letter of this operator name indicates that it deals with form links ("F") whereas the third letter indicates that it mutates the transfer mode ("T").

Transfer mode replacement (WFTR): The transfer mode determines how the user's data are packaged when they are sent to the server. A `method` attribute of a `<form>` tag sets the transfer mode. The WFTR operator replaces all `GET` requests with `POST` requests and all `POST` requests with `GET` requests.

Original Code

```
<html>
  ...
  <form action=URL1 method="POST">
  ...
   <input type="text" name="pname1" ...
   <input type="text" name="pname2" ...
   <input type="text" name="pname3" ...
  ...
  </form>
</html>
```

△

WFTR Mutant

```
<html>
  ...
  <form action=URL1 method="GET">
  ...
   <input type="text" name="pname1" ...
   <input type="text" name="pname2" ...
   <input type="text" name="pname3" ...
  ...
  </form>
</html>
```

A `POST` mode (or method) constructs the form data as an HTML message sent to the action URL. An example of the request sent with a `POST` mode is shown below.

```
POST web compoent HTTP/1.1
Host:  URL1
pname1=value1&pname2=value2&pname3=value3 ...
```

A `GET` mode (or method) causes form data to be appended to the URL as

```
URL1/web component?pname1=value1&pname2=value2&pname3=value3 ...
```

WFTR mutants can be killed by tests that submit forms with inputs that impact the response of the request. For instance, for tests that submit forms with large data (including files being uploaded), some of the data is dropped if the data size exceeds the limitation of a `GET` request. Moreover, since GET requests can be cached and remain in the browser history, tests that click a browser `back` button or `reload` button will cause the GET requests to be re-executed; but the tests will cause the browser to alert the users (in some forms of pop-up windows) that the data needs to be re-submitted. An alternative way to verify is to examine the URLs to ensure that confidential information is not exposed. The purpose of the WFTR operator is to guide testers to generate test inputs to ensure that transfer modes are specified appropriately.

4.2.4 Operator for Faults due to Data Exchanges between Web Components

In addition to the naming convention mentioned earlier, the second and the third letters of the operator name indicates that it mutates a parameter-value pair used in data exchange ("PV").

Parameter-value deletion (WPVD): The WPVD removes a parameter-value pair from a JSP `include` action. WPVD mutants can be killed by tests that submit requests to the URL that trigger the JPS `include` action whose parameter-value pair is mutated. The WPVD operator helps testers to verify that the target web component receives all needed data. The purpose of this operator is to guide testers to generate tests that exercise an input that is used to communicate between web components.

Original Code

```
<html>
  ...
  <jsp:include page=URL1 />
  <jsp:param name=pname1
    value = value1 />
  <jsp:param name=pname2
    value = value2 />
  ...
</html>
```

△

WPVD Mutant

```
<html>
  ...
  <jsp:include page=URL1 />


  <jsp:param name=pname2
    value = value2 />
  ...
</html>
```

4.2.5 Operator for Faults due to Novel Control Connections

In addition to the naming convention mentioned earlier, the second and the third letters of the operator name indicates that it mutates a control transition ("CT") of the app under test.

Control transition replacement (WCTR): The WCTR operator replaces `redirect` transitions with `forward` transitions and replaces `forward` transitions with `redirect` transitions. This operator serves two purposes. First, as a forward transition only allows destinations on the same server, this operator helps ensure that the destination to be forwarded (or redirected) to is correct. Second, since the objects executed in the original request remain available when a forward transition is used, this operator helps ensure that the app handles the objects in the original request and the forward (or redirect) request properly to avoid data anomalies.

The following example shows a WCTR mutant that replaces a redirect control connection with a forward control connection in a J2EE servlet.

Original Code

```
public class logout extends HttpServlet
{
  ...
  public void doGet(...  )
  {
    ...
    response.sendRedirect(URL1);
  }
}
```

△

WCTR Mutant

```
public class logout extends HttpServlet
{
  ...
  public void doGet(...  )
  {
    ...
    getServletContext()
    .getRequestDispatcher(URL1)
    .forward(request, response);
  }
}
```

The following example shows a WCTR mutant that replaces a forward control connection with a redirect control connection in a J2EE servlet.

Original Code

```
public class logout extends HttpServlet
{
  ...
  public void doGet(...  )
  {
    ...
    getServletContext()
    .getRequestDispatcher(URL1)
    .forward(request, response);
  }
}
```

△

WCTR Mutant

```
public class logout extends HttpServlet
{
  ...
  public void doGet(...  )
  {
    ...
    response.sendRedirect(URL1);
  }
}
```

WCTR mutants can be killed in two ways. The first is when the target web component is on a different server. Another is when the target component is on the same server and a request affects the app's output or internal state. To kill WCTR mutants, tests must submit requests to a URL that triggers the control transition. When `redirect` transitions are used, the `redirect` transitions instruct the browsers to create new requests and send them to the target components. Since the browsers are aware of the transition, tests that click the browser `back` button or `reload` button have no affect on the output (for instance, no duplicate data when a request involves data manipulation). On the other hand, when

`forward` transitions are used, the browsers are unaware that the controls have been transferred. Tests that use the browser `back` button or `reload` button cause duplication of data.

An alternative way to kill WCTR mutants is to examine the URLs. When a `forward` is used, the browser address bar shows the original URL (not the forwarded URL) since the browser is unaware of the control transfer. When a `redirect` is used, the address bar shows the redirected URL as the browser is involved in the redirection.

4.2.6 Operators for Faults due to Server-Side State Management

In addition to the naming convention mentioned earlier, the second and the third letters of the operator names indicate whether the operator mutates session objects' scope ("SC"), session objects' initialization ("SI"), or session objects' attribute setting ("SA").

Scope replacement (WSCR): Web apps can be used by multiple users at once. To handle each user's transactions, persistent data are stored in in-memory objects, called the session objects. Accessibility of the object must be specified to ensure that persistent data are valid and the user's requests are properly processed. Based on J2EE, the WSCR operator alters a `scope` attribute of a `<jsp:useBean>` to each other possible scope setting (`page`, `request`, `session`, and `application`). This operator helps testers design tests that ensure the scope of an object is set properly so that data are accessible when they should be and unavailable otherwise. WSCR mutants can be killed by tests that verify the data stored in the session object.

Original Code

```
<html>
  ...
  <jsp:useBean id=id1
    scope="session"
    class=class1 />
  ...
</html>
```

△

WSCR Mutant

```
<html>
  ...
  <jsp:useBean id=id1
    scope="request"
    class=class1 />
  ...
</html>
```

Session initialization replacement (WSIR): Since a new connection to the web server is opened every time a client retrieves a web page, information about the user session is stored in in-memory objects (`session` objects in J2EE). When a session object is initialized and there is no current session object, either a new object is created or `null` is returned. The WSIR operator changes the initialization of the session object to the opposite behavior. This operator guides testers to design tests that ensure that the web app manages the session's initialization suitably. For example, when a user starts a session (which normally involves a series of requests to accomplish a certain task), a new instance of a session object should be created. This session object should remain in memory until the user completes the task or ends the session. No additional session object should be created for this user. Improperly managing the session's initialization can create extra instances of session objects. On the other hand, a new instance should be created for each user to avoid overlapping users' data and violating data integrity of the app. WSIR mutants can be killed by tests that verify whether a new session is created after the current session is expired.

Original Code

```
public class logout extends HttpServlet
{
  ...
  public void doGet(...  )
  {
    ...
    session=request.getSession(true);
    if (session != null)
    ...
  }
}
```

△

WSIR Mutant

```
public class logout extends HttpServlet
{
  ...
  public void doGet(...  )
  {
    ...
    session=request.getSession(false);
    if (session != null)
    ...
  }
}
```

Session setAttribute deletion (WSAD): When a client (or a browser) retrieves a web page, a new connection to the web server is opened. Users' data are stored as attributes in `session` objects to be available throughout a session. The WSAD operator deletes a `session` object's `setAttribute()` statement from the code. The purpose of this operator is to guide testers to design tests to verify that the web app functions properly

when necessary information is omitted. WSAD mutants can be killed by tests that access and verify the data stored in session objects.

Original Code

```
public class logout extends HttpServlet
{
  ...
  public void doGet(...  )
  {
    ...
    session.setAttribute(attr1, value1);
    session.setAttribute(attr2, value2);
    ...
  }
}
```

△

WSAD Mutant

```
public class logout extends HttpServlet
{
  ...
  public void doGet(...  )
  {
    ...
    //session.setAttribute(attr1, value1);
    session.setAttribute(attr2, value2);
    ...
  }
}
```

4.2.7 Operators for Faults due to Client-Side State Management

Hidden form input deletion (WHID): The WHID operator removes a hidden input tag. The purpose of this operator is to guide testers to generate test inputs to ensure that data submitted to the server are properly handled. WHID mutants can be killed by tests that include a form submission and that access and verify the data stored in hidden form fields.

Original Code

```
<html>
  ...
  <form action=URL1 method="POST">
    <input type="hidden"
        name=name1 value=value1>
    <input type="hidden"
        name=name2 value=value2>
   ...
  </form>
</html>
```

△

WHID Mutant

```
<html>
  ...
  <form action=URL1 method="POST">
    <!-- input type="hidden"
        name=name1 value=value1-->
    <input type="hidden"
        name=name2 value=value2>
   ...
  </form>
</html>
```

Hidden form input replacement (WHIR): The WHIR operator changes the value of a hidden form field with another value in the same domain of the web app under test. Similar to the WHID operator, this operator helps to ensure that data submitted to

the server are handled appropriately. WHIR mutants can be killed by tests that include a form submission and that access and verify the data placed in hidden form fields.

Original Code

```
<html>
  ...
  <form action=URL1 method="POST">
    <input type="hidden"
        name=name1 value=value1>
    <input type="hidden"
        name=name2 value=value2>
   ...
  </form>
</html>
```

△

WHIR Mutant

```
<html>
  ...
  <form action=URL1 method="POST">
    <input type="hidden"
        name=name1 value=value2>
    <input type="hidden"
        name=name2 value=value2>
   ...
  </form>
</html>
```

Read-only input deletion (WOID): The WOID operator removes an `<input>` tag whose `readonly` attribute is specified. The objective of this operator is to help testers verify that the web application handles data submission properly. WOID mutants can be killed by tests that include a form submission and that access and verify the data put in readonly inputs.

Original Code

```
<html>
  ...
  <form action=URL1 method="POST">
    <input type="text" readonly
        name=name1 value=value1>
   ...
  </form>
</html>
```

△

WOID Mutant

```
<html>
  ...
  <form action=URL1 method="POST">
    <!-- input type="text" readonly
        name=name1 value=value1-->
   ...
  </form>
</html>
```

Chapter 5: Experiments

This chapter presents the research's experiments that evaluate (1) the usefulness of web mutation operators, (2) the fault detection capability of web mutation testing, (3) the overlap between web mutants and traditional Java mutants, and (4) the redundancy among web mutation operators. These experiments address the research questions in Section 1.3.

This chapter starts with Section 5.1, which discusses the design and implementation of an experimental tool, *webMuJava*. This tool was developed to demonstrate the applicability of web mutation testing. After that, the experiment in Section 5.2[1] evaluates the usefulness of web mutation operators by examining how well tests designed with traditional software testing criteria (referred to as *traditional tests*) kill web mutants. It also analyzes web mutation operators that can improve the quality of traditional tests (research questions RQ1 and RQ2 in Section 1.3). The experiment in Section 5.3 examines the applicability of web mutation testing in terms of fault detection capability and analyzes the kinds of web faults that can and cannot be detected (research questions RQ3 and RQ4 in Section 1.3). The experiment in Section 5.4 analyzes how effective web mutation adequate tests kill traditional Java mutants, and then assesses the differences and overlaps between web mutants and traditional Java mutants (research questions RQ5 and RQ6 in Section 1.3). Finally, after evaluating the applicability of web mutation testing, the experiment in Section 5.5[2] focuses on analyzing the redundancy in web mutation operators to reduce the number of test requirements; i.e., decreasing the testing costs (research questions RQ7, RQ8, and RQ9 in Section 1.3).

[1]Published in Mutation 2016 [90]
[2]Published in Mutation 2017 [89]

5.1 Experimental Tool

An automated tool is very crucial and necessary to use in web mutation testing to facilitate the experiments. This is because web mutation testing is centered around the idea of injecting web faults and producing test cases to revealing the seeded faults. Manually inserting and executing web faults becomes extremely time consuming and costly.

This research aims to develop a mutation testing tool that utilizes traditional mutation operators while providing an ability to test web apps. Several mutation testing tools were investigated. This research adopted *muJava* [66] (available at `https://cs.gmu.edu/~offutt/mujava/`) as a base prototype, based on the availability of web apps used in the experiments, the familiarity, and the human support of the mutation testing tools.

muJava is a mutation analysis tool for Java software apps that allows users to generate syntactic faults (i.e., mutants), run tests against mutants, and view mutants. The latest version of muJava supports JUnit tests and Java 1.6. muJava provides fifteen selective method-level mutation operators [64]. In this research, an experimental tool, named *webMuJava*, implementing the web-specific mutation operators defined in Chapter 4, was built as an extension of muJava. webMuJava is implemented in Java and its architecture is similar to the architecture of muJava with additional web mutation operators. In addition to JUnit, webMuJava allows tests to be automated in HTMLUnit [1], JWebUnit [3], and Selenium [2], all of which allow the tests to make HTTP calls to the web apps. webMuJava also controls various aspects of the web execution, including operational transitions such as using the browser `back` and `forward` buttons. webMuJava is a semi-automated mutation testing tool as it requires testers to supply test cases (i.e., test inputs). Functionality, structure, assumptions and designs of webMuJava are discussed in the following subsections.

5.1.1 Functionality of webMuJava

The tool allows the testers to select program files to test, specify web mutation operators to use, and view the original program and web mutants. The tool allows the testers to specify

a test set, executes mutants, and analyzes how well the test set can distinguish the mutants from the original program.

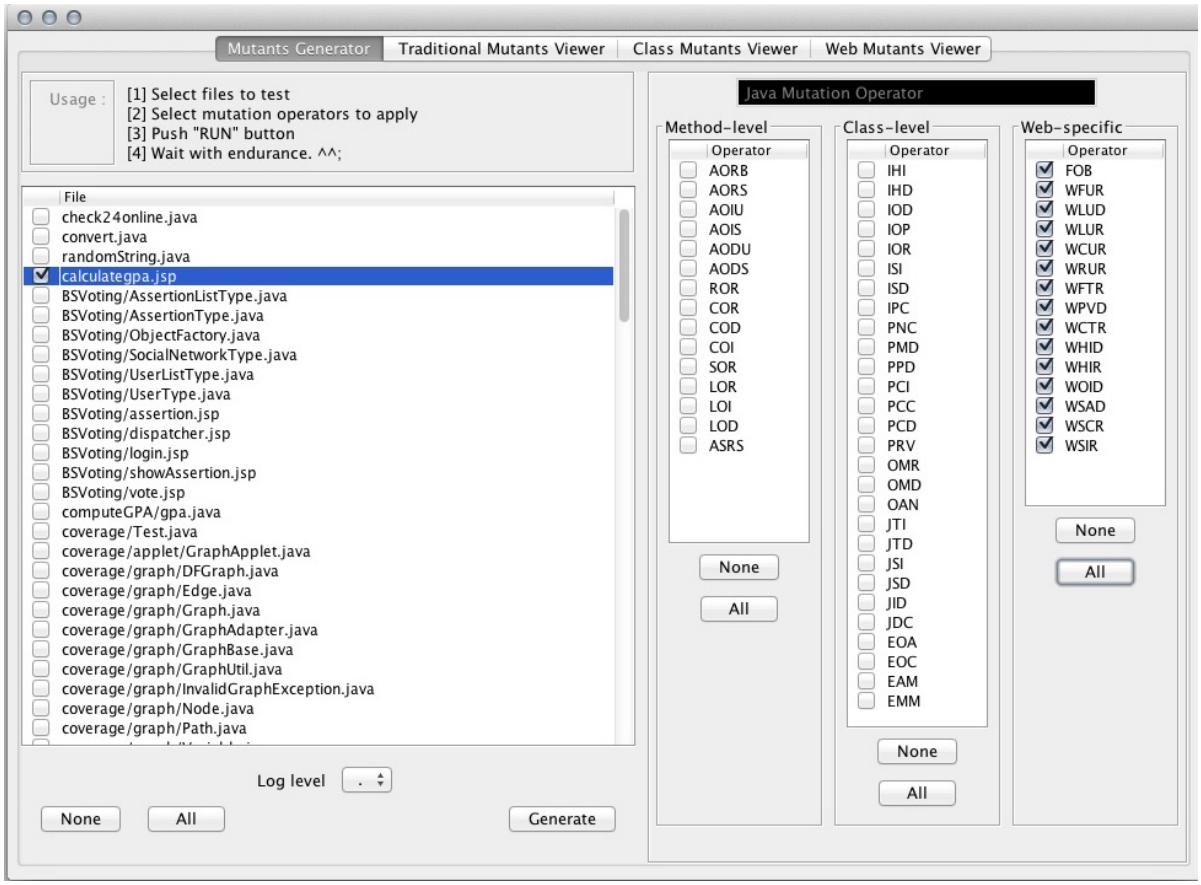

Figure 5.1: webMuJava - Screen for generating mutants

Figure 5.1 shows the webMuJava screen for generating mutants. The testers can select multiple files to test. Presumably, the testers choose files from a single web app. This ensures that the replacement code, URLs, and objects used to create mutants are from the same domain application, i.e., the replacements are meaningful. Then, the testers specify web mutation operators to apply. Once the tester clicks the `Generate` button, webMuJava automatically generates mutants from the selected operators, compiles Java servlet mutants, and organizes both servlet mutants and JSP mutants for use during execution.

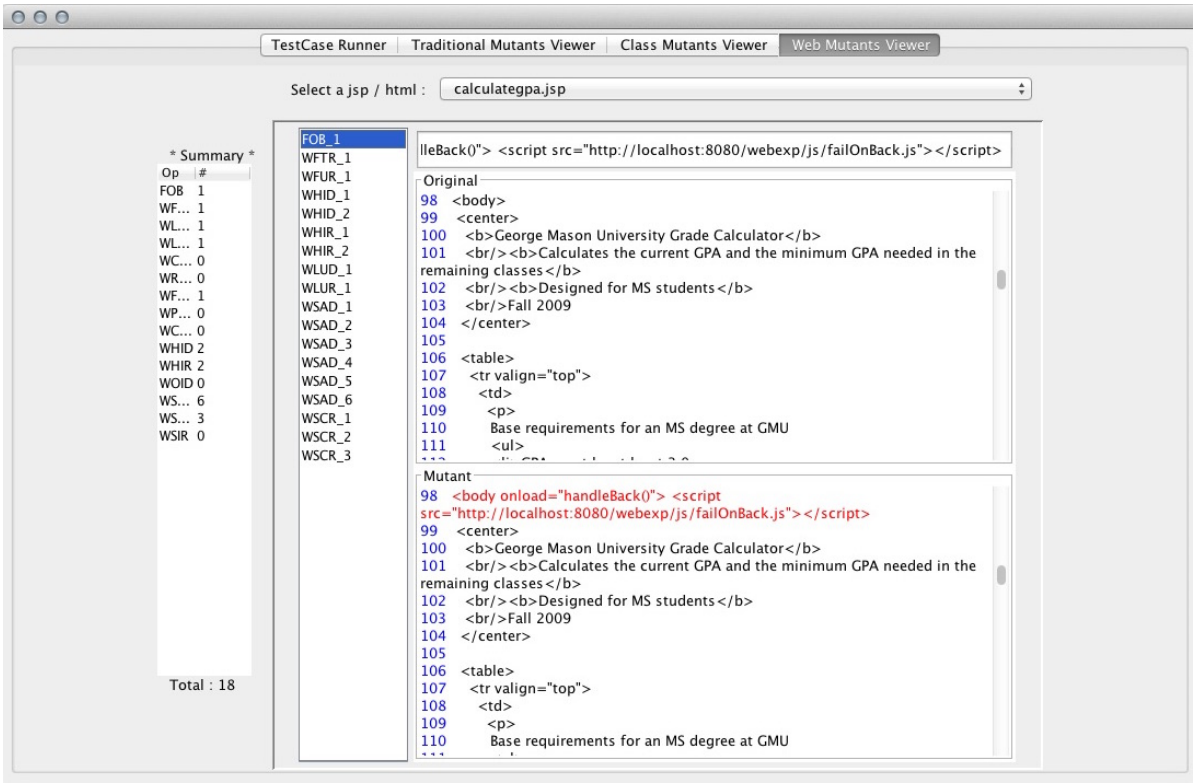

Figure 5.2: webMuJava - Screen for viewing mutants

The mutants can be viewed and compared with the original source program in the `Web Mutants Viewer` panel, as shown in Figure 5.2. The tester can view each mutant by selecting the mutant names. In this example, from 18 mutants generated for a target file `calculategpa.jsp`, FOB_1 mutant is chosen. FOB_1 mutant shows the source code being mutated at line `98`. Summary information about the mutant is presented above the original source code.

Figure 5.3 displays the webMuJava screen for executing mutants. The tester selects a target file. Accordingly, mutants generated for the chosen file are listed on the left portion of the screen. The tester supplies a set of test cases; in this example, a collection of tests named `calculategpa_mutantTest`. Figures 5.4 and 5.5 present examples of test cases for an FOB mutant and a WSCR mutant. Each test case is a sequence of requests made by a

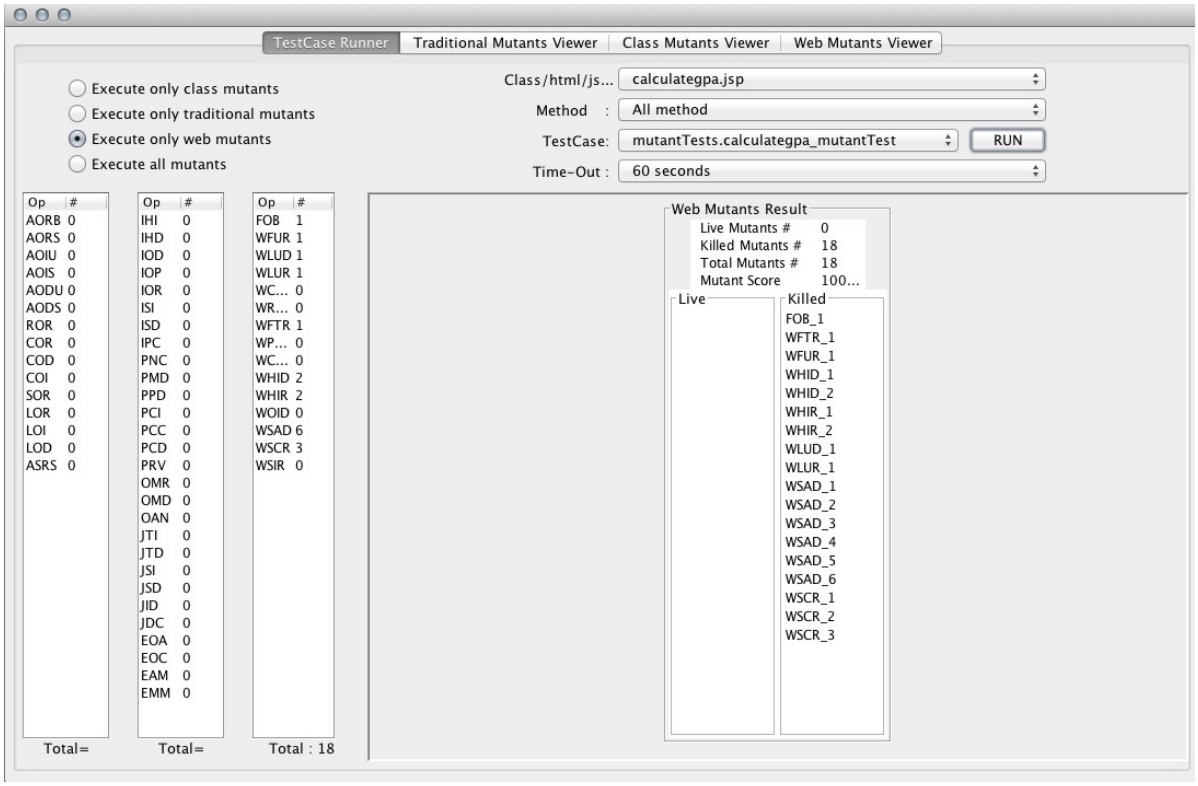

Figure 5.3: webMuJava - Screen for executing mutants and showing test results

user via a web browser or requests made between web components.

Generally, a test set consists of one or more test cases. To prepare and clean up the test environment, a test set usually comprises a test setup and a tear down. Testers may developed tests with JUnit, JWebUnit, HtmlUnit, and Selenium. Because webMuJava runs test cases in threads, testers must not assume that the test cases will be executed in the order they are written. Thus, testers must ensure that each test case has proper pre-state and post-state.

Then, the tester may specify a time-out constraint (a default time-out is 3 seconds), which is used to interrupt infinite loops, non-responsive servers[3], and other executions that may not terminate. Clicking the `RUN` button executes the selected tests against the mutants.

[3]A server connection timeout occurs when a web server is taking too long to reply to a request.

```
@Test
public void test_FOB_1()
{
    driver.get(url);

    assertEquals("Grade Calculator", driver.getTitle());

    // simulate a form submission
    WebElement element = driver.findElement(By.name("courseName1"));
    element.sendKeys("swe1");
    driver.findElement(
        By.xpath("//input[@type='radio' and @name='courseGrade1' and @value='B+']")).click();

    WebElement btn = driver.findElement(By.xpath("//button[@name='submitCourses']"));
    btn.click();

    // check if system works properly when clicking the browser back button
    driver.navigate().back();
    assertEquals(true, driver.getPageSource().contains(
        "Calculates the current GPA and the minimum GPA needed in the remaining classes"));

}
```

Figure 5.4: Example test case for a FOB mutant

Once the time-out constraint is reached, a mutant being executed is marked as killed.

Once a target file and a test set are provided, webMuJava executes the mutants. Test evaluation is based on responses from the server, usually in the form of HTML documents, and is done automatically. webMuJava compares the outputs i.e., the HTML document created by the original version of the web app under test to the HTML document produced by the mutant. If the HTML responses from the original program and from the mutant can be distinguished, the mutant is marked as killed. It is important to note that testers can tailor their test cases to target specific content on of the HTML responses. Testers can choose to verify whether the HTML response contains particular contents instead of considering an entire response. For example, the test case in Figure 5.4 verifies the existence of the string "`Calculates the current GPA and the minimum GPA needed in the remaining classes`" in the HTML responses. Then, the tool computes the mutation scores and lists the mutants that are killed and live.

```
@Test
public void test_WSCR_1()
{
    driver.get(url);

    assertEquals("Grade Calculator", driver.getTitle());

    // simulate a form submission
    WebElement element = driver.findElement(By.name("courseName1"));
    element.sendKeys("swe1");
    driver.findElement(By.xpath("//input[@type='radio' and @name='courseGrade1' and @value='A']")).click();

    WebElement btn = driver.findElement(By.xpath("//button[@name='submitCourses']"));
    btn.click();
    assertEquals(true, driver.getPageSource().contains("Your current GPA is: 4.0"));

    // simulate another form submission
    WebElement linkelement = driver.findElement(By.linkText("Add another course"));
    linkelement.click();

    element = driver.findElement(By.name("courseName2"));
    element.sendKeys("swe2");
    driver.findElement(By.xpath("//input[@type='radio' and @name='courseGrade2' and @value='B']")).click();

    btn = driver.findElement(By.xpath("//button[@name='submitCourses']"));
    btn.click();
    assertEquals(true, driver.getPageSource().contains("You need 24 more hours to graduate."));
}
```

Figure 5.5: Example test case for a WSCR mutant

5.1.2 Overview Structure of webMuJava

webMuJava is composed of four major modules: parser, mutant generator, test executor, and result analyzer, as highlighted in Figure 5.6.

Parser

webMuJava was based on muJava, which supports only Java `.class` files. Thus an additional module (the parser) was implemented in webMuJava to parse and analyze JSPs and HTMLs. The parser is responsible for analyzing web specific features, to which the mutation operators can apply. It also extracts information from a web app under test. The information includes all URLs from all transitions and all values of hidden form fields (or hidden inputs) and readonly inputs appearing in JSPs and servlets of the targeted web app along with their *frequency of reference*. Frequency of reference is the number of occurrences

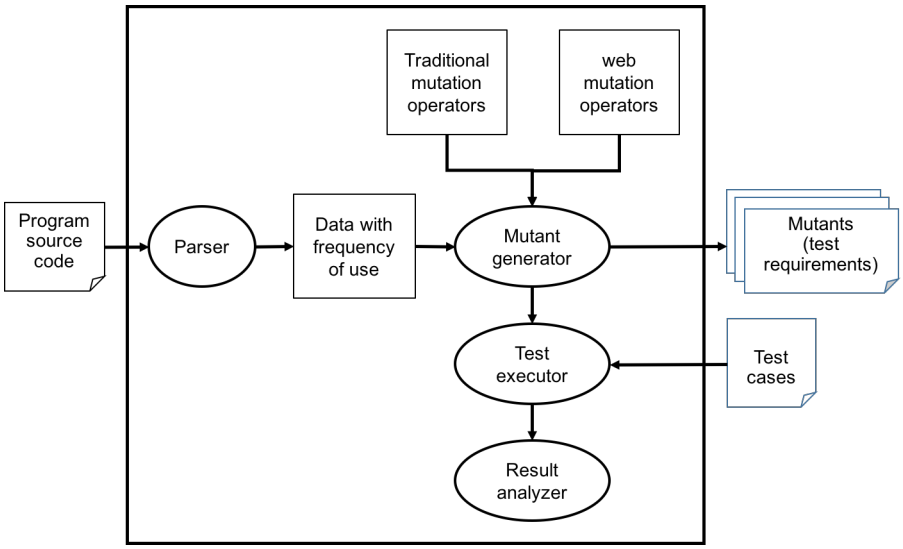

Figure 5.6: Structure of webMuJava

of the URLs (or the values of hidden inputs or readonly inputs) in the targeted web app. When the parser parses web components' source code, it keeps track of each URL (or value) and increments the frequency measurement as the URL (or the value) appears in the components. The frequency of reference helps to determine which URLs (or values) to use as replacements when creating mutants.

Mutant Generator

The mutant generator accepts program files under test. Prior to creating mutants for the given files, the mutant generator triggers the parser module to analyze and extract information to be used for replacement. Then, the mutant generator applies the tester's selected web mutation operators to the source code of the target files, creating a copy of each mutant. Each mutated file is stored in a directory indexed by its name, its mutation operator, and its mutant number; under the webMuJava's `results` directory. For instance, an FOB_1 mutant of a target file `calculategpa.jsp` is organized and stored as `/results/calculategpa.jsp/web_mutants/FOB_1/calculategpa.jsp` where a `results` directory contains mutants generated; `calculategpa.jsp` is a target file; a `web_mutants`

directory maintains the generated web mutants; `FOB_1` indicates the first web mutant created by the FOB operator; and the last `calculategpa.jsp` is the mutated file's name. The reason why the mutated file's name remains the same as the target file's name is to ensure the same test case can access both the original program and the its variation without changes. It is important to note that file extensions are attached to support a circumstance when a web app consists of multiple files of different extension with the same names.

To avoid producing equivalent mutants and redundant mutants, several design decisions were incorporated in the implementation of the tool. Further discussion on the design decisions is presented in Section 5.1.3.

Subsequently, to ensure the mutants generated are syntactically correct, the mutant generator compiles all Java servlet mutants. The corresponding `.class` files are stored in the same directory as the mutated Java files. The uncompiled Java servlet mutants are excluded from the testing process. Unlike the original muJava where each mutants is compiled right after it is created, webMuJava compiles all Java servlet mutants after all files and all web mutation operators are applied. This is because muJava does not support Java servlet and the compilation of Java servlet mutants required a specific library (`servlet-api.jar`[4]) and other applicable Java classes from the web app under tests to be included. For simplicity, webMuJava delays the compilation process until all mutants are created. It is also important to note that webMuJava does not compile JSPs. JSPs are transformed into Java servlets by the web container (Apache Tomcat, in this research), then compiled into Java servlet classes when they are requested. As a consequence, mutated JSPs that contain syntax errors must be analyzed and filtered manually.

Mutant Executor

The mutant executor runs each test case on the original and each live mutant. The executor, first, retrieves the original web app under test and then stores it in a web container. A web container is a component of a web server that handles servlets and JSPs, such as creating

[4] `http://www.java2s.com/Code/Jar/s/servlet.htm`

instances of the requested servlets and JSPs, loading and unloading the servlets and JSPs, and creating and managing request and response objects used in web apps. This research used Apache Tomcat/7.0.67 as a web container. Then, the executor runs all test cases on the original web app and records the results with respect to each test case.

The mutant executor copies each live mutant from the webMuJava's *results* directory and then successively replaces the original file with the mutant in a web container. By design, a web container loads the newest version of a JSP or a servlet when it is requested via a URL. However, since all servlet mutants are compiled at the end of the mutant generating process, there is no guarantee whether their time stamps are in the order that is consistent with the time when they are executed. Similarly, the time stamps of the JSP mutants may be older than the instances of the files under test in a web container. To ensure the mutant being considered is properly reloaded by a web container, the mutant executor updates its time stamp to the current time when it replaces the original file.

In addition to updating the mutant's time stamp, this research configures Apache Tomcat to monitor and automatically reload the incoming mutant by setting the `reloadable` attribute of the `Context` element to `true`. The `Context` element represents a web app and can be set in the file `/tomcat/conf/context.xml`, where `tomcat` is Apache Tomcat directory. That is, the `<Context>` is set to `<Context reloadable="true">`, instructing the container to reload the web app if a change is detected.

The mutant executor runs test cases in threads on the target file. For each target file, the executor opens a new web browser every time a test case is run and closes it when the test execution is completed. Launching and closing a browser for each test case introduces an overhead. To ensure the same instance of a mutant is executed by all test cases (until the mutant is killed), the executor delays copying each mutant to Tomcat. Based on a feasibility study where some test cases rely on a real browser (such as Firefox) and some rely on a virtual browser (such as a simulated browser via Eclipse), this research concludes that 10 seconds is suitable to ensure that a particular mutant is executed by all test cases. Notwithstanding, a potential drawback can be that an additional time may delay the testing

process.

For each mutant, the mutant executor triggers the result analyzer to evaluate whether the mutant is killed. If the mutant is marked as killed, it is then excluded from the testing process. The execution is repeated until all mutants and all test cases are executed.

Finally, the mutant executor computes the mutation scores based on the number of killed mutants and the number of generated mutants. Note that, the tool currently does not include any heuristics to help identify equivalent mutants. Thus, equivalent mutants must be analyzed and excluded manually.

Result Analyzer

This research uses strong mutation to determine whether a given test set can distinguish mutants from the original program, That is, the infected state must propagate to output. The result analyzer compares the outputs of running the test cases on mutants and on the original program. If they are distinguishable, the mutant is marked as killed. Otherwise, the mutant is marked as live.

5.1.3 Assumptions and Design Decisions

This research has made several assumptions and design decisions to ensure that the mutation operators use proper replacements which are in the same domain of the targeted web apps and to minimize the chance of creating equivalent mutants.

- **Mutant generation:**

  Mutants will be generated for one web app at a time, regardless of the number of its web components. This is to ensure meaningful replacements. This decreases performance, which has been decided to be less important for this research.

- **URL replacement:**

  To ensure that the mutation operators use a proper destination that is in the same domain as the targeted web app, only one web app is tested at a time. webMuJava

extracts all URLs appearing in all selected files of the web app under test. The frequency of each URL's use is analyzed. This information is recorded and later used by the operators that modify URLs in servlets and JSPs. The most frequently used URL is applied as a replacement. However, to avoid creating an equivalent mutant, if the most frequently used URL is exactly the file under test itself (for WLUR) or exactly the URL originally appearing in the form transition (for WFUR), the second most frequently used URL is selected. If there are multiple URLs with the same frequency, the first URL in the list is used. On the other hand, if a web app under test contains only one URL, a dummy URL is picked. To minimize the number of mutants created, only one URL is used for each destination replacement. The reasons webMuJava do not replace the original destination with all possible URLs are that many such mutants would be trivial and that incorrect URLs will almost always result in different behavior of the web app. Hence, a selective approach is preferred to reduce the number of mutants being created. While analyzing the frequency of use and testing one web app at a time decreases performance, they offer meaningful replacements which is more important for this research.

- **URL deletion:**

  For the operator that removes a destination of a simple link transition, no mutant is generated if the URL is exactly the file under test itself. This is because clicking the link with non-specified destination leads to the current screen, i.e., the file under test itself. This design decision reduces the likelihood of creating equivalent mutants.

- **Form hidden control replacement:**

  A similar design decision has been made for the operator that mutates hidden form fields. webMuJava extracts all hidden inputs and their values; and records the frequency of each hidden input's use. This information is later used by the operators that mutate the value of hidden inputs. The most frequently used value is selected as a replacement. If the replacement is the same as the original value, the second most frequently used value is chosen to avoid generating an equivalent mutant. For

the same reason as the destination replacement, only one value is made for each value replacement to minimize the number of mutants generated. Analyzing the frequency of use and testing one web app at a time decreases performance. However, they offer meaningful replacements which is more important for this research.

5.1.4 Known Limitations and Possible Improvement

The current implementation of webMuJava supports only servlets and JSPs. To support other web programming languages and technologies (such as JavaScript, PHP, and AJAX), some details of the implementation may need to be adapted.

The locations that store mutants and tests are predefined. Additionally, the locations that deploy web apps under test are pre-specified. Modifying the locations must be made directly to the source code of the tool. The tool must be recompiled and redeployed. To improve the tool's usability, an option allowing the testers to easily update the locations without recompiling the tool should be offered.

This research relies on Apache Tomcat 7 and Java 1.8 to deploy web apps under test. As a result, Apache Tomcat must be properly set up and maintained to ensure availability and accessibility of the apps under test. Furthermore, since the experiments conducted in this research are based on Java-based web apps, ensuring compatibility of Apache Tomcat and Java version is vital.

Delay time between each mutant is added to the mutant execution process. This introduces an overhead and can decrease performance depending on the number of mutants being executed.

The current implementation of the parser module treats all hidden inputs as string type. To make the replacement even more meaningful, improvement on semantic analysis of the data type should be incorporated.

webMuJava mutates source code of the target file. Therefore, time to generate mutants depends on time to alter the source code and time to compile necessary files. To make mutation more efficient, webMuJava could be modified to use program schema [104], which

embeds multiple mutants into each line of source code so that one source file contains all mutants.

Furthermore, webMuJava primarily follows the usual mutation testing process where killed mutants are excluded from the remainder of the testing process. To evaluate the redundancy in the mutation operators or comparing the effectiveness of the operators against each other (illustrated in an experiment is presented in Section 5.5), source code of the mutant executor module must be modified directly. The tool must then be recompiled and redeployed. This adds excise tasks, requires extra effort, and can be inflexible from the testers' perspective. To improve the usability of the tool, an option that determines an exclusion of the killed mutants should be provided on the interface for executing mutants (Figure 5.3). This allows testers to customize how the mutation testing process should be based on their interests.

Conceptually, testers may write test cases using almost any Java-based test automation frameworks (such as JUnit, HtmlUnit, HttpUnit, JWebUnit, and Selenium). However, the current implementation of the FOB operator relies on manipulating a browser history using the `pushState()` and `replaceState()` functions of a `history` object, which are supported by HTML5. Test cases developed with test automation frameworks that do not support these new functions or that use virtual browsers when executing web mutants will result in an exception (e.g., `UnsupportedOperationException` when running tests written with JWebUnit through Eclipse). Consequently, these test cases will always kill FOB mutants, regardless of whether these tests exercise the browser `back` button; thus distorting the testing result. The FOB operator should not be used in these situations.

5.2 Experimental Evaluation of Web Mutation Operators

To evaluate the usefulness of web mutation operators, this experiment focuses on answering the following research questions (previously listed in Section 1.3):

RQ1: How well do tests designed for traditional testing criteria kill web mutants?

RQ2: Can hand-designed tests kill web mutants?

The usefulness of the operators is evaluated empirically based on the number of web mutants detected by test sets that are designed with traditional software testing techniques and generated independently of mutants. This research considers traditional testing techniques that are widely used in industry such as functional requirements, input space partitioning, logic coverage, and random testing.

The underlying hypothesis was that web mutation testing can reveal more interaction faults than existing software testing techniques can. If true, the new operators can lead to improvement of test sets.

5.2.1 Experimental Subjects

In this research, web apps under test are referred to as subjects and testers are referred to as participants. Web mutation testing requires source code to be available, which limited choices for experimental subjects. In addition, tests were created by hand, further limiting the choices and the number of subjects that could be tested. Source code of web apps with existing developer-written test suites and commercial web apps were unavailable to this experiment. There were no open source or commercial web apps that had the necessary characteristics available. Therefore, this experiment took the subject web apps from examples and exercises used in web app development courses at George Mason University, and from projects by undergraduate students and graduate students.

The subject web apps were thoroughly examined and chosen to ensure that they contain various web aspects, specifically the features that can affect communication between web components and the state of web apps. The subjects had to be small enough to allow extensive hand analysis, yet large and complex enough to include a variety of interactions among web components.

Table 5.1 presents descriptive statistics of eleven Java-based web applications [5]. Components are JSPs and Java Servlets, excluding JavaScript, HTML, and CSS files. All subjects are available online at http://github.com/nanpj. *BSVoting*, *HLVoting*, and *KSVoting* are

[5]LOC, non-blank lines of code, was measured with Code Counter Pro (http://www.geronesoft.com/).

Table 5.1: Subject web apps

Subjects	Components	LOC
BSVoting (S1)	11	930
check24online (S2)	1	1619
computeGPA (S3)	2	581
conversion (S4)	1	388
faultSeeding (S5)	5	1541
HLVoting (S6)	12	939
KSVoting (S7)	7	1024
quotes (S8)	5	537
randomString (S9)	1	285
StudInfoSys (S10)	3	1766
webstutter (S11)	3	126
Total	**51**	**9736**

online voting systems, which allow users to maintain their assertions and vote on other users' assertions. *check24online* allows users to play the card game 24, where users enter four integer values between 1 and 24 inclusive and the web app generates all expressions that have the result 24. *computeGPA* accepts credit hours and grades and computes GPAs, according to George Mason University's policy. *conversion* allows users to convert measurements. *faultSeeding* facilitates fault seeding and maintains the collection of faulty versions of software. *quotes* allows users to search for quotes using keywords. *randomString* allows users to randomly choose strings with or without replacement. *StudInfoSys* allows users to maintain student information. *webstutter* allows users to check for repeated words in strings. All these subjects use features that affect the interactions between web components and the states of web apps, including form submission, redirect control connection, forward control connection, include directive, and state management mechanisms.

5.2.2 Experimental Procedure

To measure the usefulness of web mutation operators in terms of how well they can improve the quality of test sets, two types of tests were developed. The *independent* variable is the type of tests. In this scenario, a test (interchangeably referred to as a test case or a test input) is a sequence of requests made by a user via a web browser or requests made between web components. The first type of tests is generated to be adequate for web mutation

testing (as defined in Section 4.2), and the other is hand-designed tests using traditional testing criteria. Tests manually-designed with traditional testing criteria represent the most common way that web app tests are created in industry. The *dependent* variables are the number of equivalent mutants, the number of test cases required, the number of mutants generated from each operator, the mutation scores, and the usefulness of each operator. The *usefulness* of an operator is the ratio of the number of mutants from that operator **not** killed by the tests over the total number of mutants from that operator.

Four steps in conducting the experiment:

1. Generate mutants: For each subject, all fifteen operators were applied to generate fifteen kinds of mutants. M_{ij} represents mutants of the i^{th} subject that are created by the j^{th} operator. webMuJava [88] was built as an extension of muJava [65] to generate the mutants. webMuJava mutates JSPs and Java servlets. The mutated components are then compiled and integrated with the web app under test during execution.

2. Generate tests: This experiment generated two types of testes and compared their ability to detect web mutants.

Web mutation tests: A test set Tm_i was designed to kill the web mutants for each subject i. Tests were added until all web mutants were killed. Tests were created manually as sequences of requests and were automated in HtmlUnit, JWebUnit, and Selenium. To the best of our knowledge, no automated test generation tools specifically for web apps are currently available. Therefore, in this experiment, tests were created by hand.

Traditional tests: Traditional tests were generated independently from the mutation tests using the standard industrial practice of designing tests by hand from requirements. T_{ik} represents the k^{th} set of independent tests for the i^{th} subject. The tests were developed manually as sequences of requests and then automated in HtmlUnit, JWebUnit, and Selenium.

To avoid bias, this experiment took into accounts several requirements on the testers.

First, testers were not familiar with the web mutation operators. Second, testers were knowledgeable of software testing and web app development. They all had completed a software testing course, and several had industrial experience testing web apps. All but two were, or had been, professional web app software developers. Third, testers were not given specific instructions, thus were free to design tests according to their best abilities. The guide that was provided to the participants is available at `https://cs.gmu.edu/~uprapham/experiment/compareTests.html`. Two independent testers were PhD students with industrial experience. The remaining independent testers included seven teams of two part-time industrial graduate students, and three teams of two fourth year undergraduate students.

The 12 test teams design their tests independently. Each test team developed 11 test sets for 11 subjects, yielding 12 test sets for each subject web app. The testers reported that, on average, for each subject, they spent 4-6 hours to understand the subject, and design and develop test cases by hand. One test set was designed with a combination of input space partitioning and functional requirements, one with a combination of input space partitioning and logic coverage, three with a combination of logic coverage criteria and functional requirements, one with a combination of functional requirements and random testing strategies, and six with functional requirements. These tests were referred to as *traditional tests*.

3. Execute tests: Each test set was executed on all of mutants. $N(Tm_i, M_{ij})$ denotes the number of mutants of type j of subject i that were killed by tests Tm_i. $N(T_{ik}, M_{ij})$ denotes the number of mutants of type j of subject i that were killed by the k^{th} independent test set of subject i, T_{ik}. Equivalent mutants were identified manually and excluded from the testing process.

4. Compute the usefulness of the operator: The usefulness of the j^{th} operator, U_j, is computed as the ratio of the number of mutants of type j^{th} (M_j) that are **not** killed by the tests:

Table 5.2: Summary of mutants generated and killed by web mutation adequate tests

Subjects	**Mutants**	**Equivalent**	**Killed**	**Tests**
S1	27	0	27	26
S2	8	0	8	5
S3	18	0	18	14
S4	3	0	3	2
S5	115	16	99	34
S6	103	3	100	31
S7	34	0	34	17
S8	11	0	11	7
S9	5	0	5	5
S10	9	0	9	6
S11	4	0	4	2
Total	**337**	**19**	**318**	**149**

$$U_j = \frac{\sum_i^n M_{ij} - \sum_i^n N(T_{ik}, M_{ij})}{\sum_i^n M_{ij}} \tag{5.1}$$

The usefulness (U_j) ranges from 0 to 1, where 1 indicates that all mutants have not been killed (i.e., the j^{th} operator is very useful).

5.2.3 Experimental Results and Analysis

For each subject web app, Table 5.2 summarizes the number of mutants generated, the number of equivalent mutants, the number of mutants killed, and the number of tests generated with web mutation testing. Table 5.3 shows detail information on the number of mutants generated and killed by web mutation adequate tests. The upper half of the table 5.3 lists the numbers of mutants generated for each subject web app by each operator. The number of mutants created by each operator are different depending on whether the web app under test has web-specific features the operators can apply. The bottom half presents the number of web mutation adequate tests for each subject (the *Tests* column), along with the number of mutants killed by the tests (the *Mutants killed*). Since some tests killed more than one mutant, the number of tests is smaller than the number of mutants killed.

For each subject, all 15 web mutation operators were applied, generating a total of 337 mutants. Tests designed with web mutation testing criteria were able to kill 318 mutants.

Table 5.3: Number of mutants generated and killed by web mutation adequate tests

Subjects	Mutants generated															Total
	FOB	WCTR	WCUR	WFTR	WFUR	WHID	WHIR	WLUD	WLUR	WOID	WPVD	WRUR	WSAD	WSCR	WSIR	
S1	1	7	0	5	5	0	0	0	0	0	0	3	5	0	1	27
S2	1	0	0	1	1	0	0	2	3	0	0	0	0	0	0	8
S3	1	0	0	1	1	2	2	1	1	0	0	0	6	3	0	18
S4	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	3
S5	2	3	3	4	3	0	0	1	1	0	2	1	76	0	3	99
S6	4	9	0	12	12	5	5	9	9	1	0	9	17	0	8	100
S7	3	6	0	7	7	0	0	0	0	0	0	0	11	0	0	34
S8	1	0	0	2	2	0	0	2	2	0	0	0	2	0	0	11
S9	1	0	0	1	1	0	0	0	0	2	0	0	0	0	0	5
S10	2	0	0	1	1	0	0	2	3	0	0	0	0	0	0	9
S11	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	4
Total	18	26	3	36	35	7	7	17	19	3	2	13	117	3	12	318

Subjects	**Tests**	**Mutants killed**														**Total**
S1	26	1	7	0	5	5	0	0	0	0	0	0	3	5	0	1	27
S2	5	1	0	0	1	1	0	0	2	3	0	0	0	0	0	0	8
S3	14	1	0	0	1	1	2	2	1	1	0	0	0	6	3	0	18
S4	2	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	3
S5	34	2	3	3	4	3	0	0	1	1	0	2	1	76	0	3	99
S6	31	4	9	0	12	12	5	5	9	9	1	0	9	17	0	8	100
S7	17	3	6	0	7	7	0	0	0	0	0	0	0	11	0	0	34
S8	7	1	0	0	2	2	0	0	2	2	0	0	0	2	0	0	11
S9	5	1	0	0	1	1	0	0	0	0	2	0	0	0	0	0	5
S10	6	2	0	0	1	1	0	0	2	3	0	0	0	0	0	0	9
S11	2	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	4
Total	149	18	26	3	36	35	7	7	17	19	3	2	13	117	3	12	318

Again using extensive hand analysis, the other 19 were equivalent. 17 equivalent mutants were of type WSAD, one was of type WLUD, and one was of type WLUR. The WSAD operator removes a `setAttribute` statement from the source code, thereby manipulating state information of the web app. For example, one WSAD equivalent mutant (in a login component of subject S6) was due to a mutated session's attribute that was never accessed by the web app. Therefore, removing the attribute setting statement had no impact on the subject's behavior. The remaining 16 equivalent mutants were in subject S5 and were due to the fact that the mutated attributes were reset prior to being used. This suggests that the developers might have unnecessarily set the session attributes. Apparently, not only can web mutation testing criteria help detect potential faults in web apps, but it can also help web developers improve the way they implement web apps. The equivalent WLUD and WLUR mutants in subject S6 were changes to links to CSS files. Modifying the URLs to CSS files affected the presentation of the web app. However, because presentation checking was out of scope of this experiment, these WLUD and WLUR mutants were excluded. Equivalent mutants were excluded from the experiment.

For each subject, the 12 independently generated sets of tests were executed on all mutants. Table 5.4 presents data from each type of mutant. The upper half summarizes the number of mutants generated from each operator for each subject along with the total numbers of mutants. The bottom half displays the number of mutants of each type that were killed by the traditional tests. The column *Tests* gives the number of tests in each test set. The *Mutants killed* columns give the number of mutants that were killed by each test set. For example, tests T1 killed 17 WCTR mutants and 2 WCUR mutants. For several test sets, the number of tests is smaller than the number of mutants killed because some tests killed more than one mutant. To obtain these numbers, for each subject, each test was executed on all mutants.

The number of subjects and the number of test sets ought to be limited because the experiment involved extensive manual effort. In fact, many of the traditional tests were not properly prepared to automatically execute. For instance, some test sets assumed that

Table 5.4: Number of mutants generated and killed by traditional tests

Subjects	Mutants generated															Total
	FOB	WCTR	WCUR	WFTR	WFUR	WHID	WHIR	WLUD	WLUR	WOID	WPVD	WRUR	WSAD	WSCR	WSIR	
S1	1	7	0	5	5	0	0	0	0	0	0	3	5	0	1	27
S2	1	0	0	1	1	0	0	2	3	0	0	0	0	0	0	8
S3	1	0	0	1	1	2	2	1	1	0	0	0	6	3	0	18
S4	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	3
S5	2	3	3	4	3	0	0	1	1	0	2	1	76	0	3	99
S6	4	9	0	12	12	5	5	9	9	1	0	9	17	0	8	100
S7	3	6	0	7	7	0	0	0	0	0	0	0	11	0	0	34
S8	1	0	0	2	2	0	0	2	2	0	0	0	2	0	0	11
S9	1	0	0	1	1	0	0	0	0	2	0	0	0	0	0	5
S10	2	0	0	1	1	0	0	2	3	0	0	0	0	0	0	9
S11	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	4
Total	18	26	3	36	35	7	7	17	19	3	2	13	117	3	12	318

Testers	Tests	Mutants killed														Total
T1	86	0	17	2	22	27	5	3	3	3	0	1	9	76	1	4	173
T2	78	0	13	2	21	25	3	4	2	2	0	1	9	75	1	2	160
T3	168	0	16	2	20	27	7	6	6	6	0	0	10	59	2	4	165
T4	124	0	17	2	22	29	6	5	6	6	2	0	10	63	1	5	174
T5	102	0	11	2	19	23	6	6	5	5	1	0	8	57	1	4	148
T6	94	0	15	2	21	24	6	5	2	2	0	0	7	53	0	2	139
T7	118	0	17	2	23	28	6	6	4	4	1	0	8	51	1	3	154
T8	120	0	13	2	20	24	3	3	2	2	2	0	6	56	1	1	135
T9	93	0	17	2	25	30	6	6	6	6	1	0	11	60	1	4	175
T10	85	0	11	2	17	22	4	4	2	2	1	0	7	46	1	1	120
T11	95	0	14	2	21	25	6	6	2	2	0	1	9	76	1	4	169
T12	77	0	11	2	19	23	5	5	1	1	0	0	8	51	1	1	128
Average	103	0	14	2	20	25	5	4	3	3	0	0	8	60	1	2	153

test cases run in order as they were written. However, because tests were run in threads, the order of tests executed is non-deterministic. Some tests left the web apps in a state that affected the subsequent tests, causing the subsequent tests to not be able to run to completion. Consequently, all tests were inspected by hand and modified to ensure that each test case was automated properly with correctly functioning setup and teardown functions. This cleaning process was time consuming and labor intensive, limiting the number of subjects and test sets that could be used.

The mutation scores of each test on each subject are displayed in Table 5.5. The *Si* column shows the mutation score of executing a test set created by tester Tk for subject Si. For instance, test set for S1 created by test team T1 yields 0.59 mutation score. and

Table 5.5: Mutation Scores (Each test team developed one test set for each subject)

Testers	**Mutation scores (= killed mutants / total non-equivalent mutants)**											
	S1	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**	**S11**	**Average scores by tests**
T1	0.59	0.25	0.28	0.67	0.68	0.48	0.62	0.18	0.40	0.44	0.75	0.49
T2	0.67	0.25	0.44	0.67	0.68	0.36	0.44	0.18	0.40	0.44	0.75	0.48
T3	0.59	0.25	0.61	0.67	0.41	0.63	0.62	0.09	0.40	0.22	0.75	0.48
T4	0.48	0.25	0.39	0.67	0.51	0.65	0.62	0.18	0.80	0.44	0.75	0.52
T5	0.33	0.25	0.44	0.67	0.51	0.49	0.47	0.18	0.40	0.44	0.75	0.45
T6	0.59	0.25	0.28	0.67	0.39	0.41	0.71	0.18	0.40	0.22	0.75	0.44
T7	0.59	0.25	0.28	0.67	0.39	0.57	0.62	0.18	0.40	0.44	0.75	0.47
T8	0.70	0.25	0.44	0.67	0.45	0.25	0.59	0.18	0.80	0.44	0.75	0.50
T9	0.78	0.25	0.44	0.67	0.45	0.63	0.65	0.18	0.40	0.44	0.75	0.51
T10	0.52	0.25	0.44	0.67	0.35	0.28	0.56	0.18	0.40	0.44	0.75	0.44
T11	0.30	0.25	0.44	0.67	0.68	0.52	0.59	0.18	0.40	0.22	0.75	0.45
T12	0.63	0.25	0.28	0.67	0.43	0.32	0.50	0.18	0.40	0.22	0.75	0.42
Average scores by subjects	**0.56**	**0.25**	**0.40**	**0.67**	**0.49**	**0.47**	**0.58**	**0.17**	**0.47**	**0.37**	**0.75**	0.47
Average tests	**9**	**11**	**13**	**18**	**10**	**9**	**10**	**4**	**6**	**8**	**5**	
Tm	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	

test set for S7 generated by test team T6 has 0.71 mutation score. The *Average tests* indicate the average number of tests of all 12 traditional test sets created for the subject. For example, the average number of tests designed for S4 is 18 and for subject S8 is 4. On average, the mutation scores of the 12 traditional tests range from 0.17 (on subject S8) to 0.75 (on subject S11), as presented in the *Average scores by subjects*. The overall average mutation score of all non-web mutation tests is 0.47, while the overall average mutation of web-mutation tests (Tm) is 1.00.

Figure 5.7 shows the mutation scores of traditional tests on the 11 web apps. For subject S11, all test sets detected 3 out of 4 mutants, yielding the same mutation scores. Similarly, all test sets were able to distinguish WFTR and WFUR mutants of Subjects S2 and S4, but leaving the other kinds of web mutants undetected. Hence, they all yield the same mutation scores. For subject S9, two test sets detected all but FOB mutant while the remaining test sets detected only WFTR and WFUR mutants. A similar distribution applied to subject S8. In overall, most tests resulted in the same mutation scores. The deviation is relatively

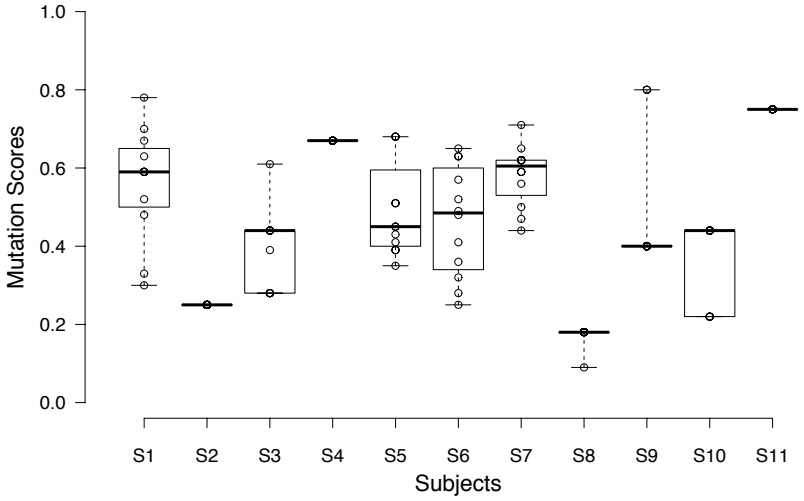

Figure 5.7: Mutation scores of traditional tests

small. One possible reason is because the tests were designed particularly based on the main functionality of the web apps under test. As a result, tests tended to check similar aspects and features of the apps, and could overlooked other web features that could affect how web apps behave.

To determine which web mutation operators would most help find web interaction faults, this research introduced a *usefulness measure* to assess the value of the operators. The usefulness measure is computed using the number of non-equivalent mutants that the traditional tests were unable to kill. Operators that created more mutants that were hard to kill are considered to be more useful at helping the testers to design strong test cases.

Table 5.6 shows the overall usefulness of each operator with respect to how many mutants that are **not** killed by each test set and the average percentages of the overall usefulness. The overall usefulness of each operator is computed as a ratio of the cumulative number of unkilled mutants of type `i` and the total number of non-equivalent mutants of type `i`. For some subjects, some types of mutants were not generated because the subjects lack the features being mutated. The distribution of the usefulness of web mutation operators is displayed in Figure 5.8.

Table 5.6: Overall usefulness of web mutation operators

Testers	**Tests**	Usefulness (= unkilled mutants of type i^{th} / total non-equivalent mutants of type i^{th})														
		FOB	WCTR	WCUR	WFTR	WFUR	WHID	WHIR	WLUD	WLUR	WOID	WPVD	WRUR	WSAD	WSCR	WSIR
T1	86	1.00	0.35	0.33	0.39	0.23	0.29	0.57	0.83	0.85	1.00	0.50	0.31	0.35	0.67	0.67
T2	78	1.00	0.50	0.33	0.42	0.29	0.47	0.43	0.89	0.90	1.00	0.50	0.31	0.36	0.67	0.83
T3	168	1.00	0.38	0.33	0.44	0.23	0.00	0.14	0.67	0.70	1.00	1.00	0.23	0.50	0.33	0.67
T4	124	1.00	0.35	0.33	0.39	0.17	0.14	0.29	0.67	0.70	0.33	1.00	0.23	0.46	0.67	0.58
T5	102	1.00	0.58	0.33	0.47	0.34	0.14	0.14	0.72	0.75	0.67	1.00	0.38	0.51	0.67	0.67
T6	94	1.00	0.42	0.33	0.42	0.31	0.14	0.29	0.89	0.90	1.00	1.00	0.46	0.55	1.00	0.83
T7	118	1.00	0.35	0.33	0.36	0.20	0.14	0.14	0.78	0.80	0.67	1.00	0.38	0.56	0.67	0.75
T8	120	1.00	0.50	0.33	0.44	0.31	0.57	0.57	0.89	0.90	0.33	1.00	0.54	0.52	0.67	0.92
T9	93	1.00	0.35	0.33	0.31	0.14	0.14	0.14	0.67	0.70	0.67	1.00	0.15	0.49	0.67	0.67
T10	85	1.00	0.58	0.33	0.53	0.37	0.43	0.43	0.89	0.90	0.67	1.00	0.46	0.61	0.67	0.92
T11	95	1.00	0.46	0.33	0.42	0.29	0.14	0.14	0.89	0.90	1.00	0.50	0.31	0.35	0.67	0.67
T12	77	1.00	0.58	0.33	0.47	0.34	0.29	0.29	0.94	0.95	1.00	1.00	0.38	0.56	0.67	0.92
Average	**103**	**1.00**	**0.45**	**0.33**	**0.42**	**0.27**	**0.25**	**0.30**	**0.81**	**0.83**	**0.78**	**0.88**	**0.35**	**0.49**	**0.67**	**0.67**

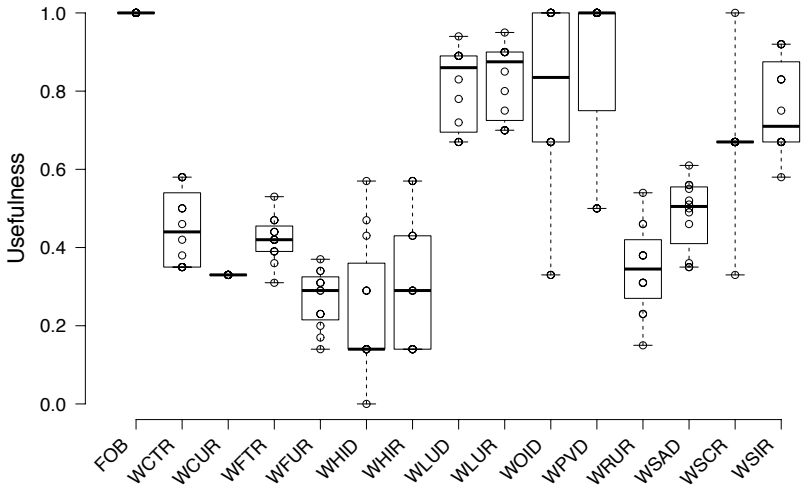

Figure 5.8: Usefulness of web mutation operators

RQ1: How well do tests designed for traditional testing criteria kill web mutants?

Although seven out of 12 test teams (i.e., traditional tests) killed more than 85% of WHID mutants, the other five test teams missed approximately half. The average usefulness of WHID operator is 0.25. The killed mutants are mostly hidden form inputs that directly affected the subjects' main functionality. For example, WHID mutants removed hidden inputs that pass user credentials through a login component of subject S6. This caused the program to display a list of assertions from a guest user's view, incorrectly limiting the user's ability to work with the software. Another example of killed WHID mutants is in subject S3. A hidden input keeping track of the number of credit hours used to compute GPA was removed, impacting the way grades were computed. On the other hand, the tests missed some WHID mutants because they removed hidden inputs that had little influence on the subjects' behavior. For instance, a hidden input field that stored a course name was removed from subject S3. Removing the course name did not affect the GPA calculation. Similar analyses apply to many WHIR mutants. While many tests can detect this kind of unexpected behaviors, many tests do not thoroughly verify the contents on the screen or

they only check for certain keywords and minimal portions of contents. This observation suggests that testers do not normally include the necessary tests even though many WHID and WHIR mutants can be easily killed.

At least 60% of WFUR mutants were killed by the traditional tests with the average usefulness of 0.27. Because WFUR mutants modify a <form> tag, test cases that submit the form will kill WFUR mutants. Many WFUR mutants were unkilled because they modified data values that were unused in the subject's main execution flow. This is a common issue with designing tests from functional requirements; testers tend to focus on "happy paths," and do not fully test other requirements. Indeed, this is a strong indication that the WFUR operator can improve test quality.

Few WLUD and WLUR mutants were killed by the traditional tests; their average usefulness scores were 0.81 and 0.83. WLUD and WLUR mutants delete and replace URLs, thus checking for broken links. The high usefulness scores indicate that testers tend not to check all static links (<A> tags). This is a strong indication that these operators can improve test quality.

On average, 58% of WFTR mutants were killed by the traditional tests. WFTR mutants replace transfer modes in <form> tags; therefore, a killing test must submit the form. Since WFUR mutants also modify <form> tags (by replacing the target URL), it seems possible that tests that kill WFUR mutants could also kill WFTR mutants. However, based on the experimental results, this assumption does not apply. Tests that killed WFUR mutants did **not** usually kill WFTR mutants on the same form. Thus, WFTR's usefulness score is higher than WFUR's usefulness score.

WSCR's average usefulness score is 0.67. WSCR mutants change the scope of JSP beans. WSCR mutants that decreased the scope (for example, from `session` to `page` scope) required tests that verified the bean was available. WSCR mutants that increased the scope could only be killed by tests that made multiple requests. For example, changing an object from `session` scope to `application` scope in subject S3 caused the number of credit hours to be cumulatively incorrect, thereby influencing the GPA computation. The

experimental results show that most traditional tests did neither of these actions. Again, this is a strong indication that WSCR can improve test quality.

Finally, the results show that none of the traditional tests killed any FOB mutants. In fact, none of the tests used the browser's back button.

In conclusion, these results clearly indicate that tests designed for traditional testing criteria miss many web mutants. They do not do well on killing web mutants.

RQ2: Can hand-designed tests kill web mutants?

This question requires test sets that were web mutation adequate. From Table 5.2, the 149 web-mutation tests for the 11 subjects killed all the web mutants. This finding answers RQ2 in the affirmative.

Table 5.4 shows an average of 103 traditional tests. Although the traditional tests were fewer in number, which is a major component of the testing cost, they only killed an average of 47% of the mutants (from Table 5.5). The largest set (T3) killed 48% and the strongest (T4) only killed 52%.

A detailed inspection of the web-mutation tests revealed that the web-mutation tests exercised many more web-specific control features than the traditional tests. These included using the back button (and other operational transitions), hitting **all** links and forms, changing the server-side state in more ways, and stressing objects in the session object. In turn, the traditional tests, which followed standard industrial practice of basing tests on requirements, tended to particularly focus on the main functionality of the web apps.

In conclusion, web mutation tests can kill more web mutants (all mutants, in this experiment) than traditional tests can. The experiment indicates that designing tests with web mutation testing can improve the quality of tests.

5.2.4 Threats to Validity

Like all empirical studies, this study has some limitations that could have influenced the experimental results. Some of these limitations may be minimized or avoidable consequences

while some may be unavoidable.

Internal validity: The quality of tests may vary depending upon the testers' experience. To minimize this threat, this experiment intentionally chose testers with a wide variety of testing knowledge and experience. Replication experiments that incorporate automatically generated tests would help confirm the result and analysis. Similarly, test values to satisfy web mutation adequate tests were created by hand. Additional experiments with different test values are vital to confirm the findings. Another potential threat is that this experiment identifies equivalent mutants by hand. Also, this experiment performed manual computation and analysis; hand analysis may introduce human errors.

External validity: While the subjects used in this experiment contained a variety of interactions among web components, it is impossible to guarantee that they are representative. Additionally, the subjects are Java-based; the results may differ when using other languages. Thus, additional experiments using subjects of other web development frameworks and languages are necessary implication for future work.

Construct validity: This experiment assumed that webMuJava worked correctly.

5.3 Experimental Evaluation for Fault Study

The previous experiment shows the ability of mutation analysis to improve the quality of test cases. This experiment further evaluates the applicability of web mutation testing for its fault detection ability. The following questions (previously listed in Section 1.3) are addressed:

RQ3: How well do tests designed for web mutation testing reveal web faults?

RQ4: What kinds of web faults are detected by web mutation testing?

To understand how well web mutation testing helps reveal web faults, the following null hypothesis H_0 and an alternative hypothesis H_a are evaluated.

H_0: Given the sets of web-mutation adequate tests, the average of fault coverage on web faults is no greater than zero (that is, the tests are not effective at revealing web faults).

H_a: Given the sets of web-mutation adequate tests, the average of fault coverage on web faults is greater than zero (that is, the tests are effective at revealing web faults).

5.3.1 Experimental Subjects

As this experiment is part of a series of attempts to validate the effectiveness of web mutation testing, this experiment also used subject web apps from the previous experiment (presented in Section 5.2). In addition to these subjects, four additional subject web apps were added to this experiment.

This experiment used 15 subject web apps. Eleven subject web apps were taken from the previous experiment. One web app, *coverageWebApp*, was taken from a tool supporting Ammann and Offutt's Introduction to Software Testing textbook [6]. One web app, *E-invite*, was taken from projects by undergraduate students who took web app development course at George Mason University. One web app, *smallTextInfoSys*, was taken from an experimental subject used by Offutt and Wu [82] to validate the atomic sections of web app modeling and also used in an initial experiment on applying mutation testing to web apps [88]. One web app, *WIBookMyDoc*, was an open source software that supports interactions between doctors and patients.

Table 5.7 lists the 15 Java-based web apps used in this experiments[6]. All subjects except *coverageWebApp* and *WIBookMyDoc* are available online at `http://github.com/nanpj`; *coverageWebApp* is available upon request to Ammann and Offutt [6] and *WIBookMyDoc* is available at `http://www.java2s.com/Open-Source/Java_Free_Code/Web_Application/Download_BookMyDoc_Free_Java_Code.htm`.

BSVoting, *HLVoting*, and *KSVoting* are online voting systems, which allow users to maintain their assertions and vote on other users' assertions. Information is stored in XML files. *check24online* allows users to play the card game 24, where users enter four integer values between 1 and 24 inclusive and the web app generates all expressions that have the result 24. *computeGPA* accepts credit hours and grades and computes GPAs, according to George

[6]LOC refers to non-blank lines of code, measured with Code Counter Pro (http://www.geronesoft.com/)

Table 5.7: Subject web apps

Subjects	**Components**	**LOC**
BSVoting (S1)	11	930
check24online (S2)	1	1619
computeGPA (S3)	2	581
conversion (S4)	1	388
coverageWebApp (S5)	25	11043
E-invite (S6)	16	1276
faultSeeding (S7)	5	1541
HLVoting (S8)	12	939
KSVoting (S9)	7	1024
quotes (S10)	5	537
randomString (S11)	1	285
smallTextInfoSys (S12)	24	1103
StudInfoSys (S13)	3	1766
webstutter (S14)	3	126
WIBookMyDoc (S15)	57	5080
Total	**173**	**28238**

Mason University's policy. *conversion* allows users to convert multiple measurements simultaneously. *coverageWebApp*, a web-based software app supporting Ammann and Offutt's Introduction to Software Testing textbook (`http://cs.gmu.edu/~offutt/softwaretest/`) [6], allows users to enter test inputs and subsequently produces test requirements and test paths. *CoverageWebApp* consists of four components: Graph Coverage, Data Flow Coverage, Logic Coverage, and Minimal-MUMCUT Coverage. Users can choose to produce test requirements and test paths using these coverage criteria. It is important to note that due to the lack of hand-seeded faults in the Minimal-MUMCUT Coverage component, this feature is excluded from the experiment; reflecting the number of components and LOC shown in the table. *E-invite* helps users to maintain events and invitations. Simultaneously, it allows other users to view and confirm whether they would attend the events or accept the invitations. Information is stored in XML files. *faultSeeding* is a tool developed specifically to facilitate the fault seeding process, compile the faulty Java files and Java servlets, and maintain the collection of faulty versions of software. *faultSeeding* was also used by some participants to introduce faults into subjects used in this experiment. *quotes* allows users to search for quotes using keywords. *randomString* allows users to randomly choose strings with or without replacement. *smallTextInfoSys* allows users to maintain text information,

using a MySQL database. *StudInfoSys* allows users to maintain student information, using XML files. *webstutter* allows users to check for repeated words in strings. *WIBookMyDoc* is an open source software that maintains doctor appointments and medical records, also using a MySQL database.

All experiment subjects use features that affect the interactions between web components and the states of web apps, including form submission, redirect control connection, forward control connection, include directive, and state management mechanisms. These subjects consist of combinations of JSPs, Java servlets, Java Beans, JavaScripts, HTMLs, XMLs, CSS files, and images. It is important to note that JavaScript and CSS are out of scope. Also, web mutation testing is not applicable to CSS, XML, and images. Therefore, this experiment excluded all JavaScripts, HTMLs, XMLs, CSS files, and images, leaving the number of components and LOC as displayed in the table.

5.3.2 Experimental Procedure

Two *independent* variables are web mutation operators (as presented in Chapter 4) and hand-seeded fault. The *dependent* variables are the number of mutants, the number of equivalent mutants, the number of test cases needed, and the mutation scores (i.e., the fault coverage).

The experiment was conducted in five steps:

1. Generate mutants: This experiment uses webMuJava to generate web mutants. For each subject, all fifteen operators were applied to generate fifteen kinds of mutants. M_{ij} represents mutants of the i^{th} subject that are created by the j^{th} operator.

2. Generate tests: A test set was designed specifically to kill mutants of each subject. This yields sets of tests T_i, that is, test set created to kill mutants of the i^{th} subject. Tests were created manually as sequences of requests and were written in HtmlUnit, JWebUnit, and Selenium, allowing the tests to make HTTP calls to the web apps.

3. Execute mutants: Each test set was executed on all mutants. $N(T_i, M_{ij})$ denotes the number of mutants of the i^{th} subject and the j^{th} mutant type that were killed by tests T_i. Equivalent mutants were identified manually and excluded from the testing process. The mutation score, i.e., indicating fault detection capability, was computed as the ratio between the number of killed mutants of all types and the total number of non-equivalent mutants of the subject. The mutation score ranges from 0 to 1, where 1 indicates that all mutants have been killed (i.e., all faults represented by the mutants have been detected) and hence the test suite is adequate. In this experiment, tests were added (repeating step 2) until the mutation score is 1, which indicates that the tests were adequate to kill all web mutants. These tests were later referred to as *mutation-adequate tests*.

4. Insert faults: Web apps with real web faults were unavailable to this experiment. Faults were hand inserted into web apps under test.

With an attempt to mimic real web faults and to avoid potential bias, certain criteria were set prior to selecting the persons who performed hand-seeded faults. First, the persons who would insert faults must not be otherwise involved in this research. Second, the persons who would seed faults must be familiar with web apps and have years of experience in web development (and possibly in software testing or web app testing). Third, no instruction or specification on the kinds of faults were given, allowing the persons to freely introduce any kinds of faults they had experienced in web apps. This experiment refers to the persons who introduced faults as participants. The initial guide that was given to the participants is available at `https://cs.gmu.edu/~uprapham/experiment/faultSeeding.html`.

For all subjects, the seeded faults were artificial and were based on the participants' experience. Some participants inserted faults manually whereas others used the *faultSeeding* tool (one of the experimental subjects, S7). Faults were inserted by 19 undergraduate students who took a Web Development course, 5 graduate students who have experienced developing web apps and 3 of them were, or had been, professional web app developers.

Hand-seeded faults were manually inspected. Faults that cause compilation errors were discarded immediately as they were not useful at determining the fault detection of mutation

tests. Note that, because JSPs are automatically compiled when they are accessed through web browsers, only faulty versions of Java files and Java servlets were compiled prior to being executed. Faults that were exactly web mutants or looked like web mutants were also excluded from the experiment to minimize potential bias toward web mutation testing.

5. Execute hand-seeded faults: To ensure comparability of fault coverage, for each subject, the same test set (T_i) used to execute web mutants was used to execute hand-seeded faults. The number of faulty versions detected was recorded. For each subject, the ratio between the number faults detected and the total number of non-mutant faults was computed. This experiment considers this ratio as fault detection and is comparable to mutation score.

5.3.3 Experimental Results and Analysis

This subsection presents the results and analysis on web mutants. It analyzes equivalent mutants and summarizes the results from running the mutation-adequate tests on hand-seeded faults. This subsection also presents the descriptions of faults detected and undetected and finally concludes the findings in response to the research questions (RQ3 and RQ4).

Results and Analysis on Web Mutants

Table 5.8 summarizes the number of mutants generated and killed. For each subject, all 15 web mutation operators were applied, generating a total of 1,738 mutants. All Java web mutants were compiled and none resulted in compilation error. JSP web mutants were automatically compiled by Apache Tomcat (a web container used to deploy web apps) as soon as they were accessed via web browsers. None of the JSP web mutants caused compilation error. Tests were designed specifically for each subject, yielding 15 test sets for 15 subjects. Across all subjects, the tests killed 1,578 mutants in total. The number of tests ranged from two to 543. On average, approximately one test killed one to three mutants.

Using extensive hand analysis, approximately 11% of generated mutants were equivalent mutants. Twelve equivalent mutants were of type WCTR, four were WHID, four were

Table 5.8: Summary of mutants generated and killed

Subjects	Mutants	Equivalent	Killed	Tests	Scores
S1	27	0	27	26	1.00
S2	8	0	8	5	1.00
S3	18	0	18	14	1.00
S4	3	0	3	2	1.00
S5	32	9	23	20	1.00
S6	95	0	95	46	1.00
S7	115	16	99	34	1.00
S8	103	3	100	31	1.00
S9	34	0	34	17	1.00
S10	11	0	11	7	1.00
S11	5	0	5	5	1.00
S12	205	15	190	98	1.00
S13	9	0	9	6	1.00
S14	4	0	4	2	1.00
S15	1069	117	952	543	1.00
Total	**1738**	**160**	**1578**	**856**	

WHIR, 58 were WLUD, 58 were WLUR, 22 were WSAD, and two were WSIR.

Twelve WCTR equivalent WCTR mutants (nine in subject S5 and 3 in subject S15) were due to replacing a *redirect* transition with a *forward* transition and replacing a *forward* transition with a *redirect* transition. In subject S5, the WCTR mutants were triggered when users click the other tool buttons (allowing users to create test requirements and test paths using other software testing coverage). Clicking these buttons lead to a new screen without transferring data or objects. The content on the screens and the URL of the mutated versions were indistinguishable from the original version of the subject. Another example of these equivalent mutants was the *New Graph* button of the *Graph Coverage* feature. Clicking this button led to a blank form to create test requirements and test paths using the Graph Coverage criteria. The targeted URL and the content on the screen remain the same. In subject S15, the WCTR mutants were triggered when users failed to login at least once (via *fail_admin_login.jsp*, *fail_doc_login.jsp*, and *fail_pat_login.jsp*) and were allowed other attempts. The redirection led to the same URLs without transferring data or objects. Therefore, the mutated code had no impact on the subject's behavior.

Four equivalent WHID mutants and four equivalent WHIR mutants were in subject S12. Three equivalent WHID mutants and three equivalent WHIR mutants dealt with changes

of values of non-keys of records to be updated to or deleted from the database. The others involved changes of hidden controls of non-keys of records to be sorted. Therefore, replacing and removing values of these hidden controls had no impact on the subject's behavior.

Fifty-eight equivalent WLUD and WLUR mutants (one WLUD and one WLUR in subject S8 and 57 WLUD and 57 WLUR in subject S15) involved changes to links to CSS files. Modifying URLs to CSS files reflected the presentation and appearance on the screen but had no impact on the subject web apps' functionality. The presentation checking was out of scope for this experiment; therefore, these mutants were excluded from study.

Six equivalent WSAD mutants (one in subject S8 and five in subject S12) were due to mutated sessions' attributes that were never accessed by the subject web apps. Thus, removing the attribute setting statements had no effect on the subjects' behaviors. The other 16 equivalent WSAD mutants were in subjects S7 and S12. For these mutants, the mutated attributes were reset prior to being accessed. This suggests that the developers might have unnecessarily set the session attributes. This observation implies that not only does the WSAD operator help verify whether data are maintained properly in session objects, it also guides developers when data shall be maintained in the sessions.

The two equivalent WSIR mutants were due to a session initialization that was changed from `request.getSession(false)` to `request.getSession(true)` in components of subject S7. Instead of not creating a new session (i.e., returning a `null`) if none exists, the mutated code issued a new session. However, because these components were included in another component, the session object was managed by its container. As a result, these WSIR mutants had no impact on the subject's behavior. This suggested that the WSIR mutant is useful only when the mutated code is not included in another component.

All equivalent mutants were removed after identified. Table 5.9 presents the non-equivalent mutants of each type. The upper half summarizes the number of mutants generated from each operator for each subject along with the total numbers of mutants. The number of mutants created by each operator varied by the web specific features the operators can apply. The bottom half displays the number of killed mutants of each type. The

column *Tests* shows the number of tests created for each subject. The column *Mutants killed* displays the number of mutants killed by each test set. The number of tests is smaller than the number of killed mutants because some tests killed more than one mutant. The tests that killed all web mutants are referred to as mutation-adequate tests.

Results and Analysis on Hand-seeded faults

Table 5.10 summarizes the number of hand-seeded faults and the results from executing the mutation-adequate tests on hand-seeded faults. The column *Faults inserted* shows the total number of faults introduced for each subject, excluding uncompilable faults. The number of hand-seeded faults varies among JSPs, Java servlets, and Java files (ranging from 10 to 367) and are not equally distributed. The remaining columns are discussed in the following paragraphs.

With an attempt to avoid potential bias toward web mutation testing, this experiment divided the hand-seeded faults into *mutant-faults* and *non-mutant faults*. *Mutant-faults* are hand-seeded faults that look like or are exactly web mutants whereas *non-mutant faults* are hand-seeded faults that do not look like or cannot be described by web mutants. The columns *Mutant-faults* and *Non-mutant faults* represent the number of mutant-faults and the number of non-mutant faults inserted into each subject. Figure 5.9 displays an overview of the number of mutant-faults and non-mutant faults introduced across all subjects.

Mutant-faults

Manual analysis revealed that approximately 27% of hand-seeded faults looked like or were exactly web mutants. Most mutant-faults were due to the use of incorrect or nonexistent URLs in form submissions (using <form> tags), simple link transitions (using <a> tags), and redirect transitions (using the `sendRedirect()` method in servlets) and forward transitions (using the <`jsp:forward`> action tags in JSPs). Many related to changes in form transfer mode (`GET` and `POST`). Several others were due to changes in scope setting of `jsp:useBean`s. For example, in S12, a `scope` attribute was altered from `session`

Table 5.9: Number of mutants generated and killed

Subjects	Mutants generated															Total
	FOB	WCTR	WCUR	WFTR	WFUR	WHID	WHIR	WLUD	WLUR	WOID	WPVD	WRUR	WSAD	WSCR	WSIR	
S1	1	7	0	5	5	0	0	0	0	0	0	3	5	0	1	27
S2	1	0	0	1	1	0	0	2	3	0	0	0	0	0	0	8
S3	1	0	0	1	1	2	2	1	1	0	0	0	6	3	0	18
S4	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	3
S5	3	0	0	3	3	0	0	0	0	0	0	9	5	0	0	23
S6	7	12	9	7	7	0	0	8	8	0	0	11	2	24	0	95
S7	2	3	3	4	3	0	0	1	1	0	2	1	76	0	3	99
S8	4	9	0	12	12	5	5	9	9	1	0	9	17	0	8	100
S9	3	6	0	7	7	0	0	0	0	0	0	0	11	0	0	34
S10	1	0	0	2	2	0	0	2	2	0	0	0	2	0	0	11
S11	1	0	0	1	1	0	0	0	0	2	0	0	0	0	0	5
S12	16	0	31	12	12	7	7	32	32	0	3	0	5	31	2	190
S13	2	0	0	1	1	0	0	2	3	0	0	0	0	0	0	9
S14	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	4
S15	50	58	0	24	24	18	18	344	367	0	0	23	23	0	3	952
Total	**94**	**96**	**43**	**82**	**81**	**32**	**32**	**401**	**426**	**3**	**5**	**56**	**152**	**58**	**17**	**1581**

	Tests	Mutants killed															Total
S1	26	1	7	0	5	5	0	0	0	0	0	0	3	5	0	1	27
S2	5	1	0	0	1	1	0	0	2	3	0	0	0	0	0	0	8
S3	14	1	0	0	1	1	2	2	1	1	0	0	0	6	3	0	18
S4	2	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	3
S5	20	3	0	0	3	3	0	0	0	0	0	0	9	5	0	0	23
S6	46	7	12	9	7	7	0	0	8	8	0	0	11	2	24	0	95
S7	34	2	3	3	4	3	0	0	1	1	0	2	1	76	0	3	99
S8	31	4	9	0	12	12	5	5	9	9	1	0	9	17	0	8	100
S9	17	3	6	0	7	7	0	0	0	0	0	0	0	11	0	0	34
S10	7	1	0	0	2	2	0	0	2	2	0	0	0	2	0	0	11
S11	5	1	0	0	1	1	0	0	0	0	2	0	0	0	0	0	5
S12	98	16	0	31	12	12	7	7	32	32	0	3	0	5	31	2	190
S13	6	2	0	0	1	1	0	0	2	3	0	0	0	0	0	0	9
S14	2	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	4
S15	543	50	58	0	24	24	18	18	344	367	0	0	23	23	0	3	952
Total	**856**	**94**	**96**	**43**	**82**	**81**	**32**	**32**	**401**	**426**	**3**	**5**	**56**	**152**	**58**	**17**	**1581**

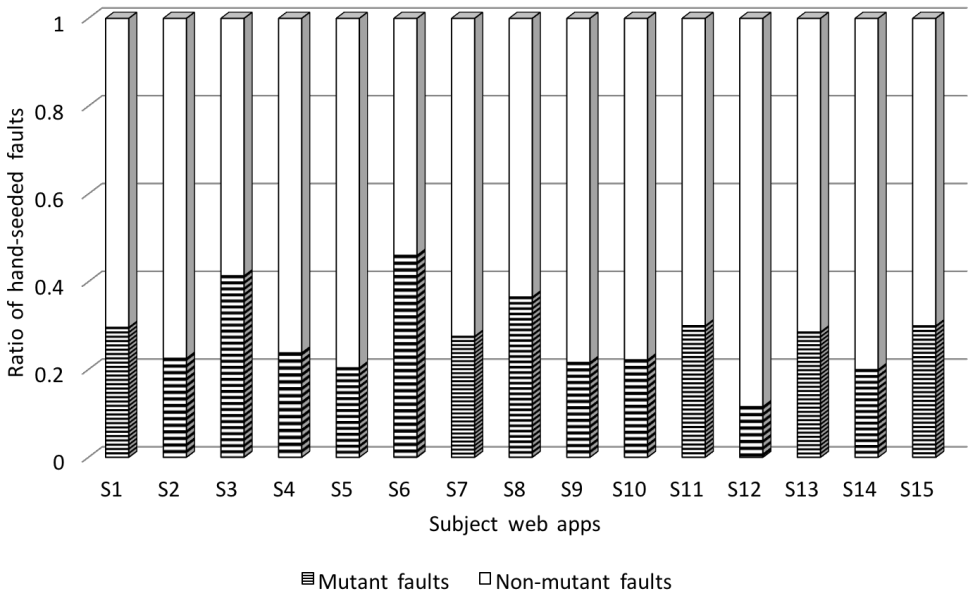

Figure 5.9: Hand-seeded fault (mutant-faults and non-mutant faults)

to `page` as <`jsp:useBean id="iconst" scope="page" class="stis.ConstBean"`>. Others related to changes in session initialization to the opposite behavior; that is, instead of creating an instance of a session object when none existed, no instance was created and a `null` value was returned. For example, in a `ControlServlet` component of S15, `request.getSession(true)` was altered to `request.getSession(false)`, causing an initialization to result in a `null`. Fault descriptions along with the number of mutant-fault are summarized in Table 5.11. Since participants were freely introduced faults based on their experience, the data suggested that participants had experienced the number of faults in web apps that can be represented by web mutants. This implies that web mutation operators have great potential to describe and substitute web faults.

These mutant-faults were excluded from the experiment after being identified.

Table 5.10: Summary of hand-seeded faults

Subjects	**Tests**	**Faults inserted**	**Mutant-faults**	**Non-mutant faults**	**Faults undetected**	**Faults Found**	**Ratio** $(= \frac{Faults detected}{Faults inserted})$
S1	26	27	8	19	2	17	0.89
S2	5	31	7	24	4	20	0.83
S3	14	29	12	17	2	15	0.88
S4	2	21	5	16	0	16	1.00
S5	20	49	10	39	7	32	0.82
S6	46	63	29	34	11	23	0.68
S7	34	58	16	42	8	34	0.81
S8	31	52	19	33	5	28	0.85
S9	17	37	8	29	3	26	0.90
S10	7	18	4	14	1	13	0.93
S11	5	10	3	7	0	7	1.00
S12	98	147	17	130	12	118	0.91
S13	6	21	6	15	2	13	0.87
S14	2	15	3	12	1	11	0.92
S15	543	367	110	257	48	209	0.81
Total	**856**	**945**	**257**	**688**	**106**	**582**	**Avg=0.85**

Table 5.11: Summary of mutant-faults

Fault description	**Number of faults**
Inappropriate scope setting (`page`, `request`, `session`, `application`)	64
Inappropriate session initialization (replace `true` or `false` in `getSession()`	5
Incorrect form transfer mode	87
Incorrect URLs of form submissions	73
Incorrect URLs of link transitions	18
Incorrect URLs of redirect transitions	5
Incorrect URLs of forward transitions	5
Total	**257**

Non-mutant faults

For each subject, the mutation-adequate test set was run on non-mutant faults. The columns *Faults undetected* and *Faults detected* in Table 5.10 present the numbers of non-mutant faults that the tests missed and found. The experiment marked faults that crashed the system immediately when the app under test was accessed as being killed by any tests associating with the app. Then, the experiment computed the fault detection capability of the tests on each subject as a ratio between the number of faults found and the total number of non-mutant faults, as displayed in the column *Ratio*. This ratio was later used to analyze for how well tests designed for web mutation testing reveal web faults. Figure 5.10 shows a visual representation of the number of faults detected by mutation-adequate

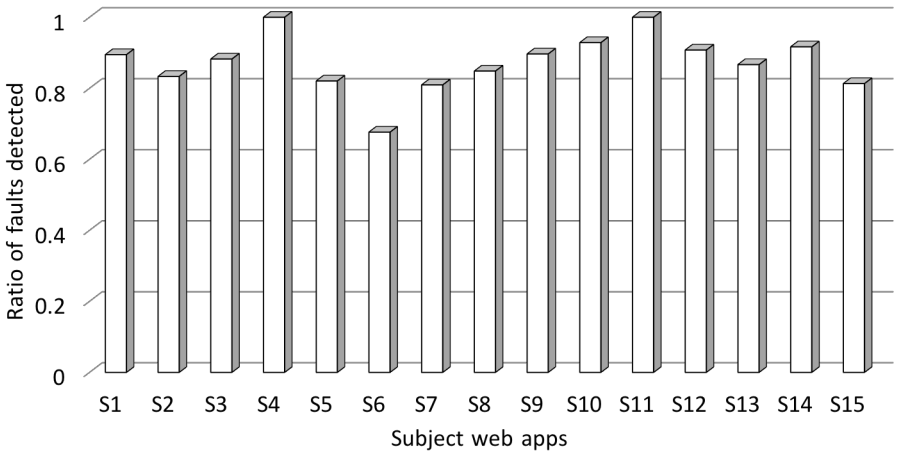

Figure 5.10: Non-mutant faults detected by web mutation tests

tests for each subject.

RQ3: How well do tests designed for web mutation testing reveal web faults?

According to Table 5.10, web mutation-adequate tests revealed at least 68% of hand-seeded faults, with an average fault detection of 85% across 15 subjects. Two subjects that the tests detected all hand-seeded faults were S4 and S11.

The majority of faults in S4 were due to changes in arithmetic operations. Some involved changes in conditional operators and relational operators. Others related to variable initialization. These faults directly affected the main functionality of the app, i.e., producing incorrect outputs and unintended behaviors. These faults were easily detected by tests that verified the main functionality of the app or tests that focused on happy paths. In subject S11, some faults were due to changes in Boolean comparison. These faults caused the execution flow of the app to an opposite behavior. Some faults involved changes in conditional operators and relational operators. Others were due to changes in initialization in loops. These faults had an impact on the main functionality of the app and could be detected by any tests that verified the app's main functionality.

The mutation-adequate tests missed approximately 32% of faults in subject S6. Most of the undetected faults in S6 were due to the use of inappropriate types of form input (as indicated in <input type="text" name="pwd"> instead of <input type="password" name="pwd">. While these faults could have been detected, no tests considered the representation of data entry and did not verify such content. Another example of type changes was the use of text type for a form element that should allow multiple lines of text entry; that is, <input type="text" name="description"> instead of <textarea name="description" rows="3" cols="50">. Another fault involved changes of the number of columns in a <textarea> element. Some faults related to changes in layout and presentation of the screen. In fact, no tests specifically considered presentation alteration of the content. Thus, these faults were missed. One fault was due to a missing input validation when a data entry is an empty string (""). No tests verified an empty string data entry, leaving this fault undetected.

Statistical Analysis of Experimental Result

To analyze how well the tests revealed web faults, this experiment, using QQplot, firstly examined the distribution of the percentages of the fault detection. The QQplot, as presented in Figure 5.11, shows that all the plots lie closely to the line. Though some plots may be slightly farther away from the line, they are still close enough to be considered as normally distributed. Hence, this experiment considered the fault detection to be normally distributed.

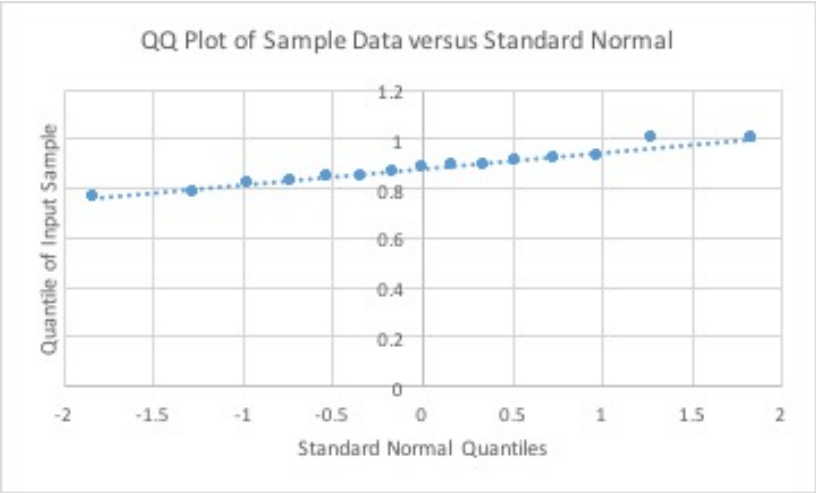

Figure 5.11: Q-Q Plot for Fault Detection

This experiment used the student's t test [41] to evaluate the null hypothesis that web mutation testing is ineffective in revealing web faults.

H_0: $\mu \leq 0$

H_A: $\mu > 0$ (research hypothesis)

Based on the hypotheses, this will be a one-tailed test. The level of significance was set to 0.05 as it has been a commonly used threshold value [41].

Table 5.12 summarizes the statistical analysis based on the fault coverage from Table 5.10. The t-value was 2.78, which is greater than the t-critical, resulting in rejection of the null hypothesis. In other words, the alternative hypothesis that web mutation testing is effective in finding faults holds. With 95% confidence, the average number of faults detected by tests designed for web mutation testing can be between 14 and 63.

Table 5.12: Summary of statistical analysis using Student's t test

Sample size	15
Sample means	38.80
Standard deviation	54.03
Standard Errors	13.95
Level of significance	0.05
Degree of freedom	14
t-value	2.78
t-critical	1.76
Confidence interval	[14.23, 63.37]

RQ4: What kinds of web faults are detected by web mutation testing?

The non-mutant faults fall into several categories. Table 5.13 lists the descriptions of non-mutants faults and summarizes the number of faults detected and undetected by the mutation-adequate tests. Faults detected by the tests were faults that had direct impact on the behaviors and functionality of the web apps under test. The majority of faults found were due to incorrect relational operators (<, <=, >, >=, ==, and !=). With these faults, the flow of execution of the apps was logically incorrect and resulted in incorrect outputs or unintended behaviors. Many other faults were due to changes in arithmetic operators and operands. These faults caused incorrect computation reflecting the behaviors of the web apps. Others were faults that altered conditional operators (&&, ||), which distorted the execution flow of the apps. Some faults related to incorrect identifications of form input elements. With these faults, the apps attempted to access non-existent form elements.

A fault related to improperly handling HTML tags in subject S7 could be detected by tests with inputs that included </textarea> tags. Subject S7 allows users to select a file to insert faults. It then loads the file source code and displays it in a multiple-line text area. The multiple-line text area, which is part of the screen, is implemented with a <textarea> tag. If the input file source code contains a </textarea> tag, this </textarea> tag overlaps with the </textarea> tag of S7. As a result, the multiple-line text displays the input source code up to the character where the </textarea> tag of the file source code encountered. The remaining file source code is displayed as text outside of the multiple-line text area. As faults can be introduced by modifying the multiple-line text area, the file source code displayed outside of the multiple-line text area is excluded. This action changes the intended behavior of subject S7.

Several others are due to the changes between an `equals` method and an equal sign (==). An example in S12 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`

Table 5.13: Summary of non-mutant faults detected and undetected by web mutation-adequate tests

Fault description	**Faults detected**	**Faults undetected**
Inappropriate condition setting in loops	29	0
Inappropriate initialization in loops	16	3
Inappropriate variable initialization	34	0
Including extra form elements (such as `text` input, `radio` button, and `checkbox`)	0	10
Incorrect Arithmetic operation (operator and operand)	104	0
Incorrect access/reference to form input (wrong name or id)	57	0
Incorrect conditional operators (&&, ||)	63	0
Incorrect form input names	32	0
Incorrect relational operators (<, <=, >, >=, ==, !=)	126	0
Incorrect return values	17	0
Incorrect scope setting in Java (`private` instead of `public`)	2	0
Incorrect update to data (XML and database)	19	0
Incorrect values of checkbox and radio button	14	3
Inverse Boolean in comparison (true instead of false, and false instead of true)	37	0
Misuse between `equalsIgnoreCase()` and `equals()`	1	3
Misuse between `session.getAttribute()` and `request.getAttribute()`	3	0
Modifying layout, format, and presentation of the screen	0	41
Not handling improper HTML tags	1	0
Omitting input validation	0	9
Omitting return values	11	0
Omitting try-catch blocks	0	2
Omitting variable initialization	14	0
Using “” instead of `null` and `null` instead of “”	0	5
Using == instead of `equals()`	0	13
Using HTTP instead of HTTPS when HTTPS is required	2	0
Using inappropriate type of form inputs (for example, `text` for password or `text` for multiple lines form input)	0	17
Total	**582**	**106**

`("password") == ""`. However, the input validation blocked empty and null strings. Therefore, this kind of faults was masked and no test 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. Nonetheless, it may be reasonable to advocate more robust testing where the masked faults are explicitly tested for, yet that is beyond the scope of this experiment.

5.3.4 Threats to Validity

Like any other research, this experiment has some limitations that could have influenced the experimental results. Some of these limitations may be minimized or avoidable consequences while some may be unavoidable.

Internal validity: One potential threat to internal validity is that hand-seeded faults and recreated faults were used as opposed to real web faults. The experience of the persons who inserted faults might affect the representativeness of web faults. There is no guarantee that the seeded faults would naturally represent real web faults.

In reality, multiple faults may present simultaneously in web apps. Detecting faults may be even more complicated. In this experiment, each individual fault was considered. Moreover, the experiment assumed each fault had no impact on others. Therefore, fault detection scenarios was simplified.

Another potential threat is that the quality of tests may vary depending upon the testers' experience. For this experiment, tests were generated manually by only one person. To minimize this threat, multiple testers should develop tests. Alternatively, test generating tools should be incorporated.

External validity: The application domain and representativeness of web apps under test may reflect the coverage, thereby distorting an analysis of the results. Though the subject web apps used in this experiment contained a variety of interactions between web components, there is no guarantee that all possible interactions were included. Hence, the results could be generalized only to web apps with similar domains and interactions between

web components. To minimize this threat, web apps with varieties of application domains need to be considered. Replication of this experiment on other web apps will also confirm the results and analysis.

Construct validity: The experiment assumed that webMuJava worked correctly.

5.4 Experimental Evaluation of Web Mutation Testing on Traditional Java Mutants

Both web mutation operators and traditional Java mutation operators target Java-based software; the former specifically deal with web-specific features. Moreover, this research implemented an experimental tool (webMuJava) as an extension of a Java-based mutation testing (muJava). Hence, understanding how well tests designed with web mutation operators do on Java mutants and how well tests designed with Java mutation operators do on web mutants can beneficially improve the overall quality of tests. This experiment focuses on examining whether web mutants and traditional Java mutants overlap. It also evaluates if web mutation testing and traditional Java mutation testing are complementary to each other.

This experiment answers the following questions (previously listed in Section 1.3):

RQ5: How well do tests designed for web mutants kill traditional Java mutants and tests designed for traditional Java mutants kill web mutants?

RQ6: How much do web mutants and traditional Java mutants overlap?

While considering traditional Java mutants, the experiment's prime concern is to understand the characteristics of faults represented by Java mutants that can be detected by tests designed for web mutants. This experiment uses traditional method-level Java mutation operators [64] implemented in muJava. These Java mutation operators target the unit testing level and their underlying concepts are applicable to other programming languages in general.

5.4.1 Experimental Subjects

Similar to previous experiments, subject web apps used in this experiment were constrained by the source code requirements of mutation testing. The availability of source code of web apps, existing developer-written test suites, and commercial web apps were limited. Furthermore, writing test cases required thorough understanding of the web apps and tests were written by hand, which further restricted the choices and the numbers of subjects that could be tested. To ensure that the subject web apps contain various web aspects that can affect communications between web components and state of web apps, the subjects had to be small enough to allow extensive hand analysis, yet large and complex enough to include variety of interactions between web components. This experiment used twelve subject web apps from previous experiment (presented in Section 5.3), excluding three subjects due to the lack of tests. Table 5.14 lists the twelve Java-based web apps[7] used in this experiments.

Table 5.14: Subject web apps

Subjects	**Components**	**LOC**
BSVoting (S1)	11	930
check24online (S2)	1	1619
computeGPA (S3)	2	581
conversion (S4)	1	388
faultSeeding (S5)	5	1541
HLVoting (S6)	12	939
KSVoting (S7)	7	1024
quotes (S8)	5	537
randomString (S9)	1	285
smallTextInfoSys (S10)	24	1103
StudInfoSys (S11)	3	1766
webstutter (S12)	3	126
Total	**75**	**10839**

5.4.2 Experimental Procedure

Focusing on evaluating how well tests designed for web mutants do on traditional Java mutants and how well tests designed for traditional Java mutants do on web mutants, the *independent* variables are web mutation operators (as presented in Section 4.2) and the

[7]LOC refers to non-blank lines of code, measured with Code Counter Pro (http://www.geronesoft.com/)

traditional method-level Java mutation operators[8]. For simplicity, traditional Java mutation operators and traditional Java mutants will be referred to as *Java mutation operators* and *Java mutants*. The *dependent* variables are the number of web mutants and Java mutants, the number of equivalent web mutants and equivalent Java mutants, the number of test cases needed for web mutants and for Java mutants, the number of mutants killed by each test set, and the mutation scores.

The experiment was conducted in four steps:

1. Generate mutants: For each subject, the experiment created two groups of mutants.

Web mutants: All fifteen web mutation operators were applied to generate fifteen sets of web mutants for the subject; let M_{W_i} represents web mutants of the i^{th} subject.

Java mutants: All fifteen method-level Java mutation operators were applied to create fifteen sets of Java mutants for the subject; let M_{J_i} represents Java mutants of the i^{th} subject.

2. Generate tests: For each subject, the experiment generated two sets of tests.

Web mutation tests: A test set T_{W_i} was designed to kill web mutants of the i^{th} subject. Tests were created manually as sequences of requests and were automated in HtmlUnit, JWebUnit, and Selenium.

Java mutation tests: A test set T_{J_i} was generated independently from the web mutation tests, specifically to kill Java mutants of the i^{th} subject. Tests were created manually as sequences of requests and were automated in HtmlUnit, JWebUnit, and Selenium.

3. Execute tests: For each subject, this experiment divided test executions into four series. First, web mutation tests designed for the subject were executed on all web mutants of the subject. The experiment keeps adding tests until all web mutants were killed. These tests were referred to as *web-mutation adequate tests*. Equivalent web mutants were identified manually and excluded from the testing process.

[8]The method descriptions are available at `http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf`.

Second, Java mutation tests designed for the subject were executed on all Java mutants of the subject. The experiment keeps adding tests until all Java mutants were killed. These tests were referred to as *Java-mutation adequate tests*. Equivalent Java mutants were hand identified and excluded from the testing process.

Third, web-mutation adequate tests were executed on all non-equivalent Java mutants of the subject. Finally, Java-mutation adequate tests were executed on all non-equivalent web mutants of the subject.

4. Compute the mutation scores: The mutation score, which indicates fault detection capability, was computed as the ratio between the number of killed mutants and the total number of non-equivalent mutants. To be specific, this experiment considered two sets of the mutation scores: (i) scores from running web-mutation adequate tests on Java mutants, and (ii) scores from running Java-mutation adequate tests on web mutants. The mutation score ranges from 0 to 1, where 1.00 indicates that all mutants have been killed (i.e., all faults have been detected), and hence the test suite is adequate.

5.4.3 Experimental Results and Analysis

This section presents four sets of analysis. The first set discusses an analysis of web mutants generated and killed by tests designed specifically for web mutants. Since equivalent mutants impact the overall mutation testing cost, equivalent web mutants across all 12 subjects were analyzed. Then, detailed analysis on the killed web mutants are presented. The second set discusses an analysis of Java mutants generated and killed by tests designed specifically for Java mutants, an analysis of equivalent Java mutants, and detailed analysis on the killed Java mutants. The third set discusses Java mutants that are killed and not killed by tests designed for web mutants. The forth set discusses web mutants that are killed and not killed by tests designed for Java mutants. Finally, the section concludes by responding to the research questions.

Table 5.15: Summary of web mutants killed by web mutation tests

Subjects	Web mutants	Equivalent	Killed	Tests	Scores
S1	27	0	27	26	1.00
S2	8	0	8	5	1.00
S3	18	0	18	14	1.00
S4	3	0	3	2	1.00
S5	115	16	99	34	1.00
S6	103	3	100	31	1.00
S7	34	0	34	17	1.00
S8	11	0	11	7	1.00
S9	5	0	5	5	1.00
S10	205	15	190	98	1.00
S11	9	0	9	6	1.00
S12	4	0	4	2	1.00
Total	**542**	**34**	**508**	**247**	

Web mutants generated and killed by web mutation tests

Overview of web mutants generated and killed

Table 5.15 summarizes the results from running web mutation tests on web mutants. This experiment generated a total of 542 mutants. Tests were designed for each subject and killed 508 mutants. Using extensive hand analysis, the experiment identified 34 equivalent web mutants (6.3%). Four equivalent mutants were of type WHID, four were of type WHIR, one were of type WLUD, one were of type WLUR, twenty-two were of type WSAD, and two were of type WSIR. All equivalent web mutants were removed after identified.

Analysis of equivalent web mutants

Four equivalent WHID mutants and four equivalent WHIR mutants were in subject S12. Three equivalent WHID mutants and three equivalent WHIR mutants involved changes of values of non-keys of records to be updated to or deleted from the database. The others involved changes of hidden form fields of non-keys of records to be sorted. Therefore, replacing and removing values of these hidden form fields had no impact on the subject's behavior.

The equivalent WLUD and WLUR mutants in subject S6 involved changes to links to CSS files. As presentation checking was out of scope for this experiment, these mutants were excluded from study.

Six equivalent WSAD mutants (one in subject S6 and five in subject S10) were due to mutated sessions' attributes that were never accessed by the subject web apps. Thus, removing the attribute setting statements did not affect the subjects' behaviors. The other 16 equivalent WSAD mutants were in subjects S5 and S10. For these mutants, the attributes that were mutated were reset prior to being accessed. This suggested that the developers might have unnecessarily set the session attributes.

The two equivalent WSIR mutants were due to a session initialization that was changed from `request.getSession(false)` to `request.getSession(true)` in components of subject S5. The mutated code issued a new session if none exists instead of not creating a new session if none exists. However, because these components were included in another component, the session object was managed by its container. As a result, these WSIR mutants had no impact on the subject's behavior. This suggested that the WSIR mutant is useful only when the mutated code is not included in another component.

Detail of killed web mutants

Table 5.16 shows data from running web mutation tests on web mutants. The upper half summarizes the number of mutants created from each operator for each subject (as displayed in the columns below *Generated web mutants)* along with the total number of mutants. For example, for subject S1, webMuJava generated one FOB mutant, seven WCTR mutants, and 27 total mutants. The number of mutants generated by each operator varied due to the web specific features the operators can apply.

Some types of mutants are not generated for some subject web apps because they lack the features being mutated. Some features, such as hidden form fields and readonly inputs, are rarely used.

The bottom half of Table 5.16 presents the number of mutants of each type that were killed by tests designed for the subject. The column *Tests* indicates the number of tests created for each subject. That is, the total of 26 tests were needed to kill all 27 web mutants of subject S1. The *Killed web mutants* columns present the number of web mutants that were killed by each test set. For example, 26 tests designed for subject S1 killed one FOB

Table 5.16: Number of non-equivalent web mutants killed by web mutation tests

Subjects	Generated web mutants															Total
	FOB	WCTR	WCUR	WFTR	WFUR	WHID	WHIR	WLUD	WLUR	WOID	WPVD	WRUR	WSAD	WSCR	WSIR	
S1	1	7	0	5	5	0	0	0	0	0	0	3	5	0	1	27
S2	1	0	0	1	1	0	0	2	3	0	0	0	0	0	0	8
S3	1	0	0	1	1	2	2	1	1	0	0	0	6	3	0	18
S4	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	3
S5	2	3	3	4	3	0	0	1	1	0	2	1	76	0	3	99
S6	4	9	0	12	12	5	5	9	9	1	0	9	17	0	8	100
S7	3	6	0	7	7	0	0	0	0	0	0	0	11	0	0	34
S8	1	0	0	2	2	0	0	2	2	0	0	0	2	0	0	11
S9	1	0	0	1	1	0	0	0	0	2	0	0	0	0	0	5
S10	16	0	31	12	12	7	7	32	32	0	3	0	5	31	2	190
S11	2	0	0	1	1	0	0	2	3	0	0	0	0	0	0	9
S12	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	4
Total	**34**	**26**	**34**	**48**	**47**	**14**	**14**	**49**	**51**	**3**	**5**	**13**	**122**	**34**	**14**	**508**

	Tests	Killed web mutants															Total
S1	26	1	7	0	5	5	0	0	0	0	0	0	3	5	0	1	27
S2	5	1	0	0	1	1	0	0	2	3	0	0	0	0	0	0	8
S3	14	1	0	0	1	1	2	2	1	1	0	0	0	6	3	0	18
S4	2	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	3
S5	34	2	3	3	4	3	0	0	1	1	0	2	1	76	0	3	99
S6	31	4	9	0	12	12	5	5	9	9	1	0	9	17	0	8	100
S7	17	3	6	0	7	7	0	0	0	0	0	0	0	11	0	0	34
S8	7	1	0	0	2	2	0	0	2	2	0	0	0	2	0	0	11
S9	5	1	0	0	1	1	0	0	0	0	2	0	0	0	0	0	5
S10	98	16	0	31	12	12	7	7	32	32	0	3	0	5	31	2	190
S11	6	2	0	0	1	1	0	0	2	3	0	0	0	0	0	0	9
S12	2	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	4
Total	**247**	**34**	**26**	**34**	**48**	**47**	**14**	**14**	**49**	**51**	**3**	**5**	**13**	**122**	**34**	**14**	**508**

mutant, seven WCTR mutants, and five WFTR mutants. The number of tests is smaller than the number of killed mutants because some tests killed more than one mutant. As these tests killed all web mutants, they were referred to as *web-mutation adequate tests*.

Java mutants generated and killed by Java mutation tests

Overview of Java mutants generated and killed

This experiment used 15 method-level mutation operators of muJava [66], producing 36,522 Java mutants. 72 Java mutants caused syntax errors and could not be compiled. An example of uncompilable mutants was a LOI mutant in subject S6 that changed `i++` in a statement `for (int i=0; i<predictionNodes.getLength(); i++)` to `for (int i=0; i<predictionNodes.getLength(); ~i++)`. Another example of uncompilable mutants was a COI mutant in subject S11 that mutated `update = true;` to `!update = true;`. The uncompilable mutants are referred to as *stillborn* mutants. These stillborn mutants are comprised of 43 AOIS mutants, 12 COI mutants, and 17 LOI mutants. Of all 72 uncompilable Java mutants, 4 are in subject S1, 28 are in S2, 8 are in S4, 12 are in S5, 7 are in S7, and 13 are in S11. The experiment excluded them from the testing process as they could not be executed and thus were not useful at evaluating the quality of tests, leaving 36,450 Java mutants in total.

Table 5.17 summarizes the number of Java mutants generated and the results from running Java mutation tests on them. The number of mutants that were generated ranged from 0 (in subject S3) to 33,224 (in subject S2). Subject S3 consists of one JSP and two Java beans. muJava does not mutate a JSP file and the two Java beans lack features that the mutation operators can apply. Thus, no Java mutant was generated for subject S3.

The experiment designed tests for each subject, creating 1,567 tests in total. Across all subjects, the tests killed 27,794 mutants. The number of tests ranged from 12 to 913. Upon execution, 327 mutants crashed the subject web apps under test as soon as the apps were accessed. Mutants that crashed the apps are comprised of 26 AOIS, 3 AORB, 97 AOIU, 25 AORS, 23 COI, and 153 LOI mutants. The crashed mutants (or *trivial* mutants)

Table 5.17: Summary of Java mutants killed by Java mutation tests

Subjects	Java mutants	Equivalent	Killed	Tests	Scores
S1	86	16	68	22	0.97
S2	33224	7958	25266	913	1.00
S3	0	*n/a*	*n/a*	*n/a*	*n/a*
S4	1010	196	814	224	1.00
S5	118	3	115	46	1.00
S6	166	39	127	49	1.00
S7	199	57	133	59	0.94
S8	243	66	177	81	1.00
S9	54	24	30	12	1.00
S10	121	7	114	51	1.00
S11	1155	257	898	93	1.00
S12	74	22	52	17	1.00
Total	**36450**	**8645**	**27794**	**1567**	**0.99**

caused an `HTTP 500 Internal Server Error` status. These trivial mutants were marked as being killed by any tests designed for the subjects. Eleven live Java mutants are of type COI. With extensive hand analysis, 8,645 of generated Java mutants were equivalent mutants (24%). The equivalent Java mutants consists of 7,926 AOIS, 4 AOIU, and 715 ROR mutants. All equivalent Java mutants were removed after identified, leaving 27,805 non-equivalent mutants in total.

Analysis of equivalent mutants

The number of equivalent AOIS mutants ranges from 0 (subject S9) to 7,334 (subject S2)[9]. The equivalent AOIS mutants conducted post increment and decrement after a value was used. Most equivalent AOIS mutants involved post increment/decrement to variables in assignment statements. For example, in subject S4, a variable `n` was mutated in an assignment statement `num2 = (float)(n / (float)100.0);` to `num2 = (float)(n-- / (float)100.0);` and `num2 = (float)(n++ / (float)100.0);`. Changes made to the variable did not affect the expression. As another example, mutating a `num2` variable when converting a meter-to-centimeter measurement in S4 from `n = Math.round( num2 * (float)100.0);` to `n = Math.round( num2++ * (float)100.0);` did not affect the computation since the value was used before the change was made. Similarly, AOIS mutants

[9]Of all equivalent AOIS mutants, 14 are in subject S1, 7,334 in S2, 168 in S4, 2 in S5, 33 in S6, 52 in S7, 38 in S8, 20 in S9, 243 in S11, and 22 in S12.

in subject S7 that modified `vote[1] = unsureCount;` to `vote[1] = unsureCount++;` and `vote[1] = unsureCount--;` had no impact on the assignment statement.

Many equivalent AOIS mutants involved post increment and decrement after variables were used in conditional statements. For example, in subject S2, the AOIS operator mutated a variable, `rslt` in `if (rslt == 24 && rslt == (float)a - (float)b / (float)c * (float)d)` to `if (rslt == 24 && rslt++ == (float)a - (float)b / (float)c * (float)d)` and mutated a variable `b` in the same if-statement to `if (rslt == 24 && rslt == (float)a - (float)b++ / (float)c * (float)d)`. The increment and decrement changes made to `rslt` and `b` reflected the variables' values after the expressions were evaluated. Therefore, they had no influence on the conditional statements.

An AOIS mutant in S6 mutated a variable `arrSize` in an if-statement; changing from `if (arrSize > -1)` to `if (arrSize-- > -1)` and `if (arrSize++ > -1)`. The variable `arrSize` was not used after the changes. The statement that got executed when the if-statement became `true` simply printed textual information without using the variable (`out.println("<p>Predictions ordered by the most agreed with</p>");`. The outputs of the original version of web app and the mutated version were indistinguishable. A similar reason applies to AOIS mutants in subject S4 that changed `switch(count)` to `switch(count--)` and `switch(count++)`.

Some AOIS mutants caused increment and decrement after the value was returned. For example, in a `getConvinced()` method of a Java bean of subject S1, a variable `convinced` indicating the number of convincing vote was increment and decrement after the value was returned. That is, `return convinced;` was mutated to `return convinced++;` and `return convinced--;`. The mutated code did not affect the app's behavior.

The number of equivalent ROR mutants ranges from 0 (subject S12) to 624 (subject S2)[10]. All equivalent ROR mutants involved changes in relational statements from `>`, `>=`, `<`, `<=`, and `==` to `true`. For example, an ROR mutant in subject S6 mutated `if (arrSize > -1)` that checks if there exist predictions to `if (true)`. Therefore, a test case

[10]Of all equivalent ROR mutants, 2 are in subject S1, 624 in S2, 28 in S4, 1 in S5, 6 in S6, 5 in S7, 28 in S8, 4 in S9, 7 in S10, 10 in S11, 0 in S12.

that includes viewing a list of predictions does not differentiate between the original version and this mutated version of S6. Similarly, an ROR mutant in subject S8 altered `if (searchRes.getSize() < 0)` that checks for existence of quotes that meet that search criteria to `if (true)`. The mutated code did not affect the app's behavior. Thus, test cases that search for quote does not distinguish between the original version and the mutated version.

Detail of killed Java mutants

Table 5.18 shows data from running the Java mutation tests on Java mutants. The upper half summarizes the number of Java mutants that were generated from each operator for each subject (as displayed in the columns below *Generated Java mutants*) along with the total number of mutants. Thus, for subject S1, muJava generated one AODU mutant, eight AOIS mutants, and 71 total mutants. The number of mutant generated by each operator varied due to the specific features the operators can apply. Similar to web mutant generation, some types of Java mutants are not generated for some subject web apps because they lack the features being mutated.

The bottom half of Table 5.18 presents data on killing mutants. The column *Tests* indicates the number of tests created for each subject. Thus, 22 tests were designed to kill 71 Java mutants of subject S1. The *Killed Java mutants* columns display the number of Java mutants of each type that were killed by the test set designed for the subject. For example, 22 tests designed for subject S1 killed one AODU mutant, eight AOIS mutants, and three AOIU mutants. For all subjects, the number of tests is smaller than the number of mutants killed because some tests killed more than one mutant.

Eleven live Java mutants are of type COI, two of which are in subject S1 and nine are in subject S7. All these live mutants involved changes in `try-catch` blocks handling a `NullPointerException`. Since the subjects (S1 and S7) perform input validation to ensure all form inputs are entered, the input validation blocks test cases from supplying a `null` data entry. For this reason, the tests missed some mutants in subjects S1 and S7; these tests are not Java-mutation adequate tests.

Table 5.18: Number of non-equivalent Java mutants killed by Java mutation tests

Subjects	Generated Java mutants															**Total**
	AODS	AODU	AOIS	AOIU	AORB	AORS	ASRS	COD	COI	COR	LOD	LOI	LOR	ROR	SOR	
S1	0	1	8	3	16	3	0	0	8	2	0	10	0	19	0	70
S2	0	0	7334	676	7488	0	0	0	936	624	0	4464	0	3744	0	25266
S3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S4	0	0	168	98	296	0	0	0	42	28	0	98	0	84	0	814
S5	0	0	2	1	64	0	0	0	16	22	0	1	0	9	0	115
S6	1	0	33	18	4	4	0	2	12	6	0	28	0	19	0	127
S7	0	0	22	19	8	8	0	0	20	4	0	29	0	32	0	142
S8	0	0	28	13	0	4	0	3	28	14	0	26	0	61	0	177
S9	0	0	12	5	0	2	0	0	0	0	0	0	0	11	0	30
S10	0	0	2	0	0	0	0	4	36	26	0	1	0	45	0	114
S11	0	0	137	81	8	13	0	11	185	170	0	176	0	117	0	898
S12	0	0	18	3	12	3	0	0	2	0	0	14	0	0	0	52
Total	**1**	**1**	**7764**	**917**	**7896**	**37**	**0**	**20**	**1285**	**896**	**0**	**4847**	**0**	**4141**	**0**	**27805**

	Tests	**Killed Java mutants**														Total
S1	22	0	1	8	3	16	3	0	0	6	2	0	10	0	19	0	68
S2	913	0	0	7334	676	7488	0	0	0	936	624	0	4464	0	3744	0	25266
S3	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*
S4	224	0	0	168	98	296	0	0	0	42	28	0	98	0	84	0	814
S5	46	0	0	2	1	64	0	0	0	16	22	0	1	0	9	0	115
S6	49	1	0	33	18	4	4	0	2	12	6	0	28	0	19	0	127
S7	59	0	0	22	19	8	8	0	0	11	4	0	29	0	32	0	133
S8	81	0	0	28	13	0	4	0	3	28	14	0	26	0	61	0	177
S9	12	0	0	12	5	0	2	0	0	0	0	0	0	0	11	0	30
S10	51	0	0	2	0	0	0	0	4	36	26	0	1	0	45	0	114
S11	93	0	0	137	81	8	13	0	11	185	170	0	176	0	117	0	898
S12	17	0	0	18	3	12	3	0	0	2	0	0	14	0	0	0	52
Total	**1567**	**1**	**1**	**7764**	**917**	**7896**	**37**	**0**	**20**	**1274**	**896**	**0**	**4847**	**0**	**4141**	**0**	**27794**

RQ5: How well do tests designed for web mutants kill traditional Java mutants and tests designed for traditional Java mutants kill web mutants?

The following discussion presents experimental results on running web-mutation adequate tests on Java mutants and running Java mutation tests on web mutants. The analysis includes detail on the mutants that are easily killed and the mutants that are seldom killed.

Java mutants killed by web mutation tests

Table 5.19 summarizes the results from running web mutation tests on Java mutants. Across all twelve subjects, on average, web mutation tests killed 66% of Java mutants. For each subject, this experiment executed web mutation tests designed for it and recorded the number of killed Java mutants of the subject. The mutation scores range from 0.38 (subject S4) to 0.87 (subject S12).

Table 5.19: Summary of non-equivalent Java mutants killed by web mutation tests

Subjects	**Java mutants**	**Killed**	**Tests**	**Scores**
S1	70	29	26	0.41
S2	25266	21559	5	0.85
S3	0	*n/a*	*n/a*	*n/a*
S4	814	310	2	0.38
S5	115	81	34	0.70
S6	127	77	31	0.61
S7	142	76	17	0.54
S8	177	104	7	0.59
S9	30	22	5	0.73
S10	114	86	98	0.75
S11	898	764	6	0.85
S12	52	45	2	0.87
Total	**27805**	**23153**	**233**	**Avg=0.66**

The experimental results show no particular kinds of Java mutants that web mutation tests missed entirely. In other words, web mutation tests were able to detect all kinds of Java mutants generated for the twelve subjects but the number of killed mutants varied. To understand how well web-mutation adequate tests do on Java mutants, this experiment considers two groups of killed Java mutants: (i) mutants that are easily killed and (ii) mutants that are seldom killed.

Table 5.20 presents the detailed results from running web mutation tests on Java mutants. The structure of this table is similar to Table 5.18, except that the column *Tests* shows the number of web-mutation adequate tests designed for each subject and the bottom displays the number of Java mutants killed by web-mutation adequate tests. The *ratio* indicates the proportion of killed Java mutants of each type (i.e., the number of killed Java mutants divided by the number of generated Java mutants of that type).

Java mutants that are easily killed by web mutation tests

Java mutants that are relatively easy to kill are mutants that directly affect the main behaviors or functionalities of the subjects under test. These mutants include most of AOIS, AOIU, AORB, AORS, COD, COI, and LOI mutants.

The AOIS operator conducts pre and post increment/decrement to variables. AOIS mutants that interfere an execution of a loop and AOIS mutants whose mutated variables are used immediately (or closely) after the mutated code are relatively easy to kill. For example, in subject S6, a statement `for (int i=arrSize; i >= 0; i--)` was mutated to `for (int i=arrSize; i++ >= 0; i--)`. A post increment to a variable `i` caused an infinite loop. This kind of AOIS mutants can be killed by any tests with non-empty arrays. Other easily killed AOIS mutants are those involve use of variables after they are conducted a post increment/decrement. For instance, a variable `i` in subject S6 was mutated in `if (i > -1)` to `if (i-- > -1)`. The statement `out.println(predArr.get(i).getPred());`, which was executed as a result of the if-statement, was influent by the changes made to the variable `i`. This AOIS mutant can be killed by tests with non-empty `predArray` arrays (i.e., there exist predictions).

While most AOIS mutants are quite easy to detect, some AOIS mutants that involve skipped indexes when iterating over arrays require additional verification. For example, a statement `for(int i = 0; i < predictionNodes.getLength(); i++)` in subject S6 was changed to `for(int i = 0; i++ < predictionNodes.getLength(); i++)`. The mutated code skipped an index when iterating over a `predictionNodes`. Instead

Table 5.20: Number of non-equivalent Java mutants killed by web mutation tests

Subjects		**Generated Java Mutants**															**Total**
		AODS	AODU	AOIS	AOIU	AORB	AORS	ASRS	COD	COI	COR	LOD	LOI	LOR	ROR	SOR	
S1		0	1	8	3	16	3	0	0	8	2	0	10	0	19	0	70
S2		0	0	7334	676	7488	0	0	0	936	624	0	4464	0	3744	0	25266
S3		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S4		0	0	168	98	296	0	0	0	42	28	0	98	0	84	0	814
S5		0	0	2	1	64	0	0	0	16	22	0	1	0	9	0	115
S6		1	0	33	18	4	4	0	2	12	6	0	28	0	19	0	127
S7		0	0	22	19	8	8	0	0	20	4	0	29	0	32	0	142
S8		0	0	28	13	0	4	0	3	28	14	0	26	0	61	0	177
S9		0	0	12	5	0	2	0	0	0	0	0	0	0	11	0	30
S10		0	0	2	0	0	0	0	4	36	26	0	1	0	45	0	114
S11		0	0	137	81	8	13	0	11	185	170	0	176	0	117	0	898
S12		0	0	18	3	12	3	0	0	2	0	0	14	0	0	0	52
Total		**1**	**1**	**7764**	**917**	**7896**	**37**	**0**	**20**	**1285**	**896**	**0**	**4847**	**0**	**4141**	**0**	**27805**
	Tests	**Killed Java Mutants**															Total
S1	26	0	0	7	3	0	3	0	0	6	2	0	4	0	4	0	29
S2	5	0	0	7119	676	7201	0	0	0	936	264	0	4464	0	899	0	21559
S3	*n/a*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S4	2	0	0	96	29	104	0	0	0	14	7	0	43	0	17	0	310
S5	34	0	0	0	0	64	0	0	0	10	3	0	0	0	4	0	81
S6	31	1	0	30	6	4	3	0	2	11	3	0	8	0	9	0	77
S7	17	0	0	12	13	4	3	0	0	2	4	0	18	0	20	0	76
S8	7	0	0	20	8	0	0	0	3	17	6	0	13	0	37	0	104
S9	5	0	0	12	5	0	0	2	0	0	0	0	0	0	3	0	22
S10	98	0	0	0	0	0	0	0	1	36	15	0	1	0	33	0	86
S11	6	0	0	137	81	8	13	0	11	182	88	0	176	0	68	0	764
S12	2	0	0	14	3	9	3	0	0	2	0	0	14	0	0	0	45
Total	**233**	**1**	**0**	**7447**	**824**	**7394**	**27**	**0**	**17**	**1216**	**392**	**0**	**4741**	**0**	**1094**	**0**	**23153**
Ratio		**1.00**	**0.00**	**0.96**	**0.89**	**0.94**	**0.73**	***n/a***	**0.85**	**0.95**	**0.44**	***n/a***	**0.98**	***n/a***	**0.26**	***n/a***	

of accessing an array `predictionNodes` with index 0, 1, 2, 3, up to the size of the array; the iterating over the array will be with index 0, 2, 4, up to the size of the array. Another AOIS mutant in subject S6 changed a statement `for (int i=arrSize; i >= 0; i--)` to `for (int i=arrSize; i-- >= 0; i--)`. Similarly, an index was skipped. Other AOIS mutants involved changes in assignment statements. For instance, in subject S6, an AOIS mutant altered `org.w3c.dom.Node inst = predictionNodes.item(i);` to `org.w3c.dom.Node inst = predictionNodes.item(i++);`. Some other AOIS mutants involved changes in print statements. For example, `out.println("<input type ='hidden' name='indexToDelete' value='" + i + "'>" );`, which is part of a for-loop in subject S7, was mutated to `out.println("<input type='hidden' name='indexToDelete' value='" + i++ + "'>");` During an iteration, the post increment and decrement caused an index to be skipped. These mutants are killed by web mutation tests that verify every index (i.e., every element) of the arrays. Nevertheless, the experimental result shows that because only some of web mutation tests verify every element in the arrays, the tests missed some of AOIS mutants that resulted in skipped indexes.

The AOIU operator negates variables' values. For instance, in subject S2, some AOIU mutants changed a statement `rslt = (a * b - c) / d;` to `rslt = (a * b - c) / -d;` or changed `rslt = a * (b * (c / d));` to `rslt = a * (b * (-c / d));`. Any tests with non-zero values of the variables `d` (as in the former example) and `c` (as in the later example) will kill these mutants. As another example, in subject S6, some AOIU mutants changed `vote[0] = agreeCount;` to `vote[0] = -agreeCount;`. Negating a variable in an assignment statement is easily distinguishable.

Furthermore, some AOIU mutants caused run-time errors due to accessing a negative index of an array. For example, an AOIU mutant in subject S6 mutated a statement `out.print(predArr.get(i).getAgree());`, which is part of a for-loop (with an index `i` and under a `predArr` $<$ `0` condition), to `out.print(predArr.get(-i).getAgree());`. A `predArr` is an array containing predictions. An attempt to access a prediction in an array with a negative index results in a run-time error. Any tests that view the existing predictions

will kill this mutant. A similar reason applies to a change from `org.w3c.dom.Note node = predItems.item(j)` to `org.w3c.dom.Note node = predItems.item(-j)`. Another AOIU mutant in subject S6 changed from `if (username.equals(predArr.get(i).getUser()` to `if (username.equals(predArr.get(-i).getUser()` where `i` refers to an index of an array `predArr`. Test cases that attempt to view every element in an array of existing predictions will detect the mutated behavior.

The AORB operator mutates arithmetic operation. Hence, it causes the computation to be evaluated differently. For example, in subject S2, `if (rslt == 24 && rslt == (float)a * ((float)b * ((float)c / (float)d )))` was changed to `if (rslt == 24 && rslt == (float)a / ((float)b * ((float)c / (float)d )))`. In subject S6, a statement `int arrSize = predArr.size() - 1;` was changed to `int arrSize = predArr.size() * 1;`. Modifying the arithmetic operator distorts the computation, hence producing incorrect outcomes. Any tests that reach the mutated statements and that with non-zero values will kill AORB mutants.

While killing many AOIS, AOIU, and AORB mutants across all subjects, web mutation tests missed many of them in subject S4. In this subject, all AOIS, AOIU, and AORB mutants involved changes to measurements conversion. To exercise all conversions, all form inputs must be entered. Web mutation tests designed for this subject included approximately half of the form inputs. The tests were able to detect AOIS, AOIU, and AORB mutants that affected the conversion of inputs the tests exercised. However, the tests missed the remaining mutants that affected the conversion of inputs the tests did not exercise. This suggests that web mutation tests (to be precise, the tests that were designed specifically to kill WFUR mutants) can potentially detect these Java mutants. When designing tests to kill WFUR mutants, tests only need to include a form submission regardless of how many form inputs to supply. This implies that if web mutation tests supply more form inputs and perform a form submission, they are likely to detect more of these AOIS, AOIU, and AORB mutants. On the other hand, if web mutation tests supply fewer form inputs and perform a form submission, they are likely to detect fewer of these Java mutants.

The AORS operator replaces a decrement with an increment and an increment with a decrement. This causes an opposite behavior. For example, an AORS mutant in subject S6 mutated a statement `for(int j=0; j < predItems.getLength(); j++)` to `for(int j=0; j < predItems.getLength(); j--)`. This mutant causes an infinite loop and can be killed by any tests with non-empty `predItems`. Another AORS mutant in subject S6 changed `for (int i=arrSize; i >= 0; i--)` to `for (int i=arrSize; i>=0; i++)`. This mutant also causes an infinite loop. Any tests with non-empty `arrSize` can kill it.

The COD operator deletes a unary conditional operator (`!`) from conditional expressions whereas the COI operator inserts a unary conditional operator (`!`) into conditional expressions. These operators reverse the execution paths of web apps. For example, a COD mutant in subject S11 altered `if (!(new java.io.File(FileName)).exists()` to `if ((new java.io.File(FileName)).exists()`. The mutated code causes the conditional expression to be evaluated in an opposite behavior. Any tests that reach the if-statements will kill these mutants. An example of COI mutants in subject S6 involved a change from `for (int i=0; i < predictionNodes.getLength(); i++)` in a `doPost()` method to `for (int i=0; !(i < predictionNodes.getLength(); i++)`. The change reverses the state of the comparison and hence resulting in an incorrect execution path. Any tests that reach the mutated code, with a HTTP POST request in this example, will kill this COI mutant. A COI mutant in subject S11 altered a statement `if (verifyStud( req, "new" ) == true)` to `if (!(verifyStud( req, "new" ) == true))`. Another example, a COI mutant in subject S5 changed a statement in a `doPost()` method from `if (mask != null)` to `if (!(mask != null))`. These mutants change the execution flow. Hence, any tests with a form submission will kill them.

Although many COI mutants are quite easy to kill, web mutation tests missed some COI mutants that involve changes in `try-catch` blocks that handle a `NullPointerException` in subjects S1 and S7. Due to the input validation of the subjects, form data entires were filtered. The input validation blocked the tests from supplying a `null` input. As a result, the tests did not kill these COI mutants.

The LOI operator inserts a unary logical operator ("∼"). Many LOI mutants involved inserting a "∼" to indexes when accessing arrays. For example, a LOI mutant in subject S6 changed a statement `org.w3c.dom.Node inst = predictionNodes.item(i);` to `org.w3c.dom.Node inst = predictionNodes.item(∼i);`. A `predictionNodes`, in this example, contains information about predictions that allow users of the subject web app to view and vote on. This LOI mutant causes a negative indexing, which results in a run-time error. Any tests with existing predictions can kill it. Other examples of LOI mutants that resulted in negative indexing are also in subject S6. For instance, a statement `count++;` was changed to `∼count++;`, a statement `switch(count)` was changed to `switch(∼count)`, a statement `out.println("<input type='hidden' name='indexToDelete' value='" + i + "'>");` was changed to `out.println("<input type='hidden' name='indexToDelete' value='" + ∼i + "'>");`, and a statement `out.print(predArr.get(i).getArgs());` was changed to `out.print(predArr.get(∼i).getArgs());`. The mutated statements cause runtime errors. Any tests that reach the mutated code will kill them.

Java mutants that are seldom killed by web mutation tests

Java mutants that are seldom killed by web mutation tests are COR and ROR mutants.

The COR operator replaces binary conditional operators (`&&`, `||`, and `^`) with other binary conditional operators. Web mutation tests were able to detect COR mutants that involved changes from `&&` to `^` whereas they missed COR mutants that replaced `&&` with `||`. For example, in subject S4, a statement `if (gAsStr != null && gAsStr.length() > 0)`, which checked a form input `gallon` prior to converting it to an ounce measurement was mutated to `if (gAsStr != null ^ gAsStr.length() > 0)`. Since the tests verified that `gAsStr` was not `null` and had a value (either numeric or float), the original statement was evaluated to `true` whereas the mutated statement were evaluated to `false`. Hence, the tests killed this mutant. On the contrary, when a statement `if (gAsStr != null && gAsStr.length() > 0)`, was mutated to `if (gAsStr != null || gAsStr.length() > 0)`, both original statement and the mutated statement were evaluated to `true`. Thus, the tests did not kill this mutant. Similarly, web mutation tests were able to differentiate

a mutated statement `if (CAsStr != null ^ CAsStr.length() > 0)` from the original statement `if (CAsStr != null && CAsStr.length() > 0)` but were unable to detect the mutated statement `if (CAsStr != null || CAsStr.length() > 0)`.

The ROR operator replaces relational operators (`>`, `>=`, `<`, `<=`, `==`, `!=`) with other relational operators, and replaces the entire predicate with `true` and `false`. Web mutation tests missed many ROR mutants because the tests did not supply the form inputs that were affected by the mutants. For example, in subject S4, the ROR operator produced seven mutants by changing a statement that verified the form input `Celsius` from `if (CAsStr != null && CAsStr > null)` to (1) `if (CAsStr != null && CAsStr >= null)`, (2) `if (CAsStr != null && CAsStr < null)`, (3) `if (CAsStr != null && CAsStr <= null)`, (4) `if (CAsStr != null && CAsStr == null)`, (5) `if (CAsStr != null && CAsStr != null)`, (6) `if (CAsStr != null && true)` , and (7) `if (CAsStr != null && false)`. Since no tests supplied the form input `Celsius`, the mutated conditional statements did not affect the subject's behavior. Therefore, the tests did not kill these ROR mutants.

Web mutants killed by Java mutation tests

Table 5.21 summarizes the results from running Java mutation tests on web mutants. Across all twelve subjects, on average, Java mutation tests killed 41% of web mutants. For each subject, this experiment executed Java mutation tests designed for it and recorded the number of killed web mutants of the subject. The mutation scores range from 0 (subject S3) to 0.67 (subject S4). This experiment does not design Java mutation tests for subject S3 since it has no Java mutants. Hence, all web mutants of S3 were marked as unkilled.

The experimental results show Java mutation tests did not detect some kinds of web mutants, as displayed in Table 5.22. To understand how well Java mutation tests do on web mutants, this experiment considers three groups of killed web mutants: (i) mutants that are easily killed, (ii) mutants that are seldom killed, and (iii) mutants that the tests missed.

Table 5.21: Summary of non-equivalent web mutants killed by Java mutation tests

Subjects	Web mutants	Killed	Tests	Scores
S1	27	13	22	0.48
S2	8	2	913	0.25
S3	18	0	0	0.00
S4	3	2	224	0.67
S5	99	29	46	0.29
S6	100	51	49	0.51
S7	34	19	59	0.56
S8	11	6	81	0.55
S9	5	2	12	0.40
S10	190	93	51	0.49
S11	9	2	93	0.22
S12	4	2	17	0.50
Total	**508**	**221**	**1567**	**Avg=0.41**

Table 5.22 presents the detailed results from running Java mutation tests on web mutants. The structure of this table is similar to Table 5.16, except that the column *Tests* shows the number of Java-mutation tests designed for each subject and the bottom displays the number of web mutants killed by Java-mutation tests. The *ratio* indicates the proportion of killed web mutants of each type (i.e., the number killed web mutants divided by the number of generated web mutants of that type).

Web mutants that are easily killed by Java mutation tests

Web mutants that directly affect the main functionality of the subject web apps are easily killed. These mutants include WRUR, WFUR, WCUR, most WFTR, and many of WHID and WHIR mutants.

The WRUR operator replaces a URL specified in a `sendRedirect()` method of an `HttpServletResponse` object with another URL. Across the twelve subject web apps, all WRUR mutants caused the execution flow to transfer from one web component to another web component. Mutating the targeted component directly affect the behavior of the app under test. For example, in subject S6, a WRUR mutant changed a statement `response.sendRedirect(http://localhost:8080/jsp/HLVoting/index.jsp);` to `response.sendRedirect(http://localhost:8080/jsp/HLVoting/predict.jsp);`. Thus,

Table 5.22: Number of non-equivalent web mutants killed by Java mutation tests

Subjects		Generated web mutants															Total
		FOB	WCTR	WCUR	WFTR	WFUR	WHID	WHIR	WLUD	WLUR	WOID	WPVD	WRUR	WSAD	WSCR	WSIR	
S1		1	7	0	5	5	0	0	0	0	0	0	3	5	0	1	27
S2		1	0	0	1	1	0	0	2	3	0	0	0	0	0	0	8
S3		1	0	0	1	1	2	2	1	1	0	0	0	6	3	0	18
S4		1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	3
S5		2	3	3	4	3	0	0	1	1	0	2	1	76	0	3	99
S6		4	9	0	12	12	5	5	9	9	1	0	9	17	0	8	100
S7		3	6	0	7	7	0	0	0	0	0	0	0	11	0	0	34
S8		1	0	0	2	2	0	0	2	2	0	0	0	2	0	0	11
S9		1	0	0	1	1	0	0	0	0	2	0	0	0	0	0	5
S10		16	0	31	12	12	7	7	32	32	0	3	0	5	31	2	190
S11		2	0	0	1	1	0	0	2	3	0	0	0	0	0	0	9
S12		1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	4
Total		**34**	**26**	**34**	**48**	**47**	**14**	**14**	**49**	**51**	**3**	**5**	**13**	**122**	**34**	**14**	**508**
	Tests	**Killed web mutants**															Total
S1	22	0	0	0	5	5	0	0	0	0	0	0	3	0	0	0	13
S2	913	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	2
S3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S4	224	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	2
S5	46	0	0	3	3	3	0	0	0	0	0	0	1	19	0	0	29
S6	49	0	0	0	8	12	5	5	0	0	0	0	9	12	0	0	51
S7	59	0	0	0	5	7	0	0	0	0	0	0	0	7	0	0	19
S8	81	0	0	0	2	2	0	0	0	0	0	0	0	2	0	0	6
S9	12	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	2
S10	51	0	0	25	10	12	4	4	13	13	0	3	0	3	6	0	93
S11	93	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	2
S12	17	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	2
Total	**1567**	**0**	**0**	**28**	**38**	**46**	**9**	**9**	**13**	**13**	**0**	**3**	**13**	**43**	**6**	**0**	**221**
Ratio		**0**	**0**	**0.82**	**0.79**	**0.98**	**0.64**	**0.64**	**0.27**	**0.25**	**0**	**0.60**	**1.00**	**0.35**	**0.18**	**0**	

instead of redirecting a user to the main function of the system after successfully logging in, the mutated code redirects the user to another component that cannot provide the intended service. This mutant can simply be killed by tests that verify the login feature of the subject. As another example, a WRUR mutant in a `login` component of subject S5 mutated a statement `response.sendRedirect(faultSeedingServlet);` to `response.sendRedirect(logout);`. Tests that include a successful login to the subject will kill this mutant.

The WFUR mutation operator replaces the URL of a <form> with another URL. For example, in subject S10, a statement <p><form method="get" action="categories .jsp"> was mutated to <p><form method="get" action="page_header.jsp">. Because form submission usually has direct impact on the main functionality of web apps, modifying the target URL causes a form to be submitted to an incorrect web component. Hence, WFUR mutants are easily detected. Unsurprisingly, tests that include form submissions can distinguish them from the original versions of web apps. Java mutation tests killed all WFUR mutants, except the one of subject S3 due to the lack of Java mutation tests.

The WCUR mutation operator replaces the URL of a `page` attribute of a JSP include action and the URL of a `file` attribute of a JSP include directive with another URL. The include action and directive cause the content of one component to be included in another component. For instance, in a `browse.jsp` component of a subject S10, a statement <%@ include file="page_footer.jsp" %> was changed to <%@ include file="index.jsp" %>. The `page_footer.jsp` component consists of menu options such as adding a new textual record, modifying a record, and adding a new category whereas the `index.jsp` component comprises of a login form. Including the `index.jsp` instead of the `page_footer.jsp` causes the options to become unavailable. Any tests that access the `browse.jsp` component of this subject and pressing one of the menu options will kill this mutant.

Java mutation tests kill many WFTR mutants. The WFTR mutation operator replaces a transfer mode with another transfer mode. If WFTR mutants affect the HTTP response

(thus changing the HTML document rendered on the screen), tests that kill WFUR mutants will also kill WFTR mutants. WFTR mutants that affect the HTTP response are mutants that involved submitting forms with data entry. It is important to note that Java mutation tests only consider the content rendered on the screen and do not verify the URL in the browser's address bar. If WFTR mutants do not affect the content of the screen (such as mutants that involve form submission without data entry), Java mutation tests do not detect them.

Java mutation tests also killed many WHID and WHIR mutants. The WHID mutation operator removes a hidden form field. The WHIR mutation operator replaces a value of a hidden form field with another value in the same application domain. WHID and WHIR mutants that the tests detected are mutants that directly affect the subject under test. For example, in subject S6, a statement <`input type='hidden' name='reqType' value='viewSubmit'/`> was removed (WHID mutant). In this subject, the hidden form field `reqType` instructs the subject to redirect a users to a screen to view existing predicates (with the `value` attribute equals to `viewSubmit`). Omitting necessary information causes the web app to behave incorrectly. An example of WHIR mutants in the same subject changed a statement <`input type='hidden' name='reqType' value='viewSubmit'/`> to <`input type='hidden' name='reqType' value='" + i + "'/`>. Replacing the state information causes the subject web app to behave inaccurately. These mutants straightforwardly impact the app's behavior; thus, they can be killed easily. On the other hand, if the mutated hidden form fields are not part of the main functionality, the tests are likely to miss them. For instance, a WHIR mutant in subject S10 modified a statement <`input type='hidden' name='rec_sort' value='InfoCategory'`> to <`input type='hidden' name='rec_sort' value='" + i + "'"`>. The hidden form field `rec_sort` indicates how the textual information should be sorted on the screen for viewing information. No Java mutation tests verify the sorting feature of the subject as it is not a primary functionality of the subject; thus, the tests missed this WHIR mutant.

Web mutants that are seldom killed by Java mutation tests

Web mutants that are seldom killed by Java mutation tests are mutants that do not directly affect the main functionality of the subject web apps but are necessary to check to ensure the apps work properly. These mutants include WLUD, WLUR, WSAD, and WSCR mutants.

The WLUD mutation operator removes a URL of the `href` attribute of an HTML <A> tag. The WLUR mutation operator replaces a URL of the `href` attribute of an HTML <A> tag. Java mutation tests were able to detect WLUD and WLUR mutants whose mutated statements influence the apps' main functionality. For example, a WLUD mutant in subject S10 involved a change in statement <td><a href="category_edit.jsp?origCategory= <%= cat_name %>">rename</td> to <td>rename </td>. A WLUR mutant changed the same statement to <td>rename</td>. This <a> link allows users to edit a particular category name. Some Java mutation tests designed for the subject include clicking the link to modify a category name. Hence, the tests killed these WLUD and WLUR mutants.

On the other hand, if the mutated links are not part of the apps' main functionality, Java mutation tests tend to overlook and do not click them. For instance, in subject S10, a statement <th>Category </th> was mutated to <th>Category</th> (creating a WLUD mutant) and to <th>Category</th> (creating a WLUR mutant). This <a> link allows users to sort the information. However, the sorting feature is not the main functionality of the subject. No Java mutation tests exercise the sorting link. Therefore, the tests missed these WLUD and WLUR mutants.

While it is possible and simple to kill WLUD and WLUR mutants, many of them are undetected. One possible reason is that they are not part of the apps' main functionality. Most tests focus on verifying the main service of the subjects and neglect to check for incorrect or broken links. Hence, it is safe to conclude that the WLUD and WLUR mutation operators can potentially improve the quality of tests.

The WSAD mutation operator removes the `setAttribute()` method of a `session` object. As the `setAttribute()` method is used to maintain the app's state across requests throughout a session, removing the `setAttribute()` method causes the state information to be discarded. Across all subjects, some WSAD mutants involve user login information. For example, in subject S6, a statement `session.setAttribute("loginFail","")` was removed. The `loginFail` attribute of a session indicates whether the user is successfully logging in to the system. The attribute is reset upon successful login and prior to redirect the users to the app's main screen (`index.jsp`). Excluding the `setAttribute()` statement directly affect the app's behavior. Java mutation tests that verify the login feature of the subject will detect this mutant.

Three other examples of WSAD mutants that straightforwardly influence the apps' main functionality are in subject S5. Statements `session.setAttribute("selectedFile", selectedFile);`, `session.setAttribute("orgCode", org_str);`, and `session.setAttribute("faultyFile", retrievedFile);` were removed. The attribute `selectedFile` keeps track of the source file's name to insert faults. The attribute `orgCode` maintains the source code used in the fault seeding process. The attribute `faultyFile` indicates the faulty file to be updated. Neglecting the necessary information causes inconsistencies in the app under test. Tests that verify the main functionality of the app will detect the mutated behavior.

In contrast, Java mutation tests do not detect WSAD mutants that involve sessions' attributes not directly affecting the main content of the apps' functionality. For example, in subject S5, as the faulty version of source code get compiled, the subject maintains the compilation status through session attributes. One WSAD mutant caused a statement `session.setAttribute("compileError", "Yes");` to be removed. Another excluded a statement `session.setAttribute("errorCode", err_msg);` from the app. These attributes provide additional information on a fault being introduced to the input source code. If the faulty version can be compiled, the subject S5 produces a faulty source file and a faulty class file and stores them in the file system. Otherwise, the subject displays the

attributes' information on the screen. The subject creates a faulty source file but does not produce its corresponding class file. To kill these WSAD mutants, tests must verify either the compilation status on the screen or the existence of the class file. No Java mutation tests verify the supplemental information nor the class file. Hence, the tests missed these WSAD mutants.

The WSCR mutation operator replaces a scope of a Java bean object with another setting. If the mutated scope is more restricted than the original scope, information maintained in the object becomes inaccessible within the session. For example, in a `login.jsp` component of subject S10, a statement `<jsp:useBean id="iconst" scope="session" class="stis.ConstBean">` was mutated to `<jsp:useBean id="iconst" scope="page" class="stis.ConstBean">`. The mutated code limits the bean object to be available only within the current web component instead of throughout the session. Hence, information maintained in the bean object is unavailable to other components and requests in the same session. Due to the nature of this subject web app, the user's login is needed to retrieve his or her textual information. Since the user's login becomes unavailable to other web components or requests in the same session, his or her information cannot be retrieved. This directly affects the subject's behavior. Java mutation tests that attempt to login and view the information killed this mutant. Nonetheless, for WSCR mutants in a component that edits a category or a component that adds information, the state information stored in a bean object has no impact on the main functionality as these components directly update the changes to the database. Other components or requests retrieve information from the database. To kill these mutants, tests must specifically verify information maintained in the bean objects. The experiment found that no Java mutation tests verify the bean objects, hence they missed these WSCR mutants.

On the contrary, if the mutated scope is less restricted than the original scope, information maintained in the bean object becomes cumulative throughout the session. For instance, in subject S10, `<jsp:useBean id="iconst" scope="session" class="stis.ConstBean">` was mutated to `<jsp:useBean id="iconst" scope="application" class=`

"stis.ConstBean">. To kill the bigger scoped WSCR mutants, tests must include a series of requests and verify the data stored in the object. The experiment found that no Java mutation tests killed these WSCR mutants. Therefore, it is safe to conclude that WSCR mutants are essential to improve the quality of tests.

Web mutants that Java mutation tests missed

Web mutants that Java mutation tests missed are mutants that involve a series of interaction with the subjects and that involve operational transitions (such as transitions caused by the client). These mutants include FOB, WCTR, and WSIR mutants.

The FOB mutation operator inserts a dummy URL to the browser history before the current URL. When a browser `back` button is pressed, the dummy URL is loaded instead of the previously viewed screen. For example, in subject S7, the FOB mutation operator inserts an `onload` function call to the <body> tag and include a JavaScript `failOnBack.js`, which manipulates the browser history to the program. That is, <body> was mutated to <body onload="handleBack()"> and a statement <script src="http://localhost:8080/experiment/js/failOnBack.js"></script> was included in the code. To kill the FOB mutants, tests must include pressing the browser `back` button. The experiment found that no Java mutation tests clicked the browser `back` button. Hence, across twelve subject web apps, the tests missed all FOB mutants.

While it is possible to kill FOB mutants, the experimental results show all FOB mutants were undetected. One possible reason why the browser features are overlooked is that the Java mutation tests target Java mutants and focus on verifying the main functionality of the subject. These tests do not consider interactions caused by the browser features as inputs to the apps. Therefore, FOB mutants are important to improve the quality of tests.

The WCTR mutation operator replaces a redirect transition with a forward transition and a forward transition with a redirect transition. If the targeted URL (i.e., destination of the redirect and the forward transition) is on the same server as the URL currently rendered on the screen, the contents on the screen are most likely indistinguishable between the original program and the mutated program. As a result, the redirect

transition and the forward transition can be differentiated by examining the URL in the browser address bar. The experiment finds that, across the twelve subjects, all WCTR mutants involve the URLs on the same server. For example, in subject S1, a statement `getServletContext().getRequestDispatcher("/BSVoting/assertion.jsp").forward (request, response);` was changed to `response.sendRedirect("/BSVoting/assertion .jsp");`. Another WCTR mutant in the same subject involved a change in a statement `response.sendRedirect("dispatcher.jsp");` to `getServletContext().getRequestDis patcher("dispatcher.jsp").forward(request,response);`. Since Java mutation tests verified the contents on the screen but not the URLs in the address bar, the tests missed all WCTR mutants.

The WSIR mutation operator changes a session object initialization to the opposite behavior. For instance, in subject S5, a statement `HttpSession session = req.getSession (true);` was mutated to `HttpSession session = req.getSession (false);`. As a result, instead of creating a session object when none exists, the `false` boolean indicates that a `null` is returned. To kill WSIR mutants, tests must invalidate the session object. The experiment found that no Java mutation tests kill WSIR mutants. Therefore, this implies that WSIR mutants can potentially improve the quality of tests.

In conclusion, the experimental results suggest that Java mutants help design tests that check all form inputs and verify individual web components whereas web mutants help design tests that focus on interactions between web components. Using both Java mutants and web mutants can improve the quality of tests.

RQ6: How much do web mutants and traditional Java mutants overlap?

Based on the ratio indicating Java mutants that were killed by web mutation tests in Table 5.20 and the ratio indicating web mutants that were killed by Java mutation tests in Table 5.22, the overlap between web mutants and traditional mutants is affirmative. The visual views illustrating the ratios are presented in Figures 5.12 and 5.13.

While web mutation tests were able to detect many Java mutants (including AOIS, AOIU, AORB, AORS, COD, COI, and LOI mutants), the tests missed many COR and ROR mutants. Likewise, Java mutation tests were able to detect many web mutants (including WRUR, WFUR, WCUR, most WFTR, and many of WHID and WHIR mutants). However, the tests seldom killed WLUD, WLUR, WSAD, and WSCR mutants and missed all FOB, WCTR, and WSIR mutants. The experimental results reveal that web mutation operators can potentially improve the quality of tests designed with traditional mutation testing and, at the same time, traditional Java mutation operators can help improve the quality of tests designed with web mutation testing. In conclusion, web mutation testing and traditional Java mutation testing are complementary.

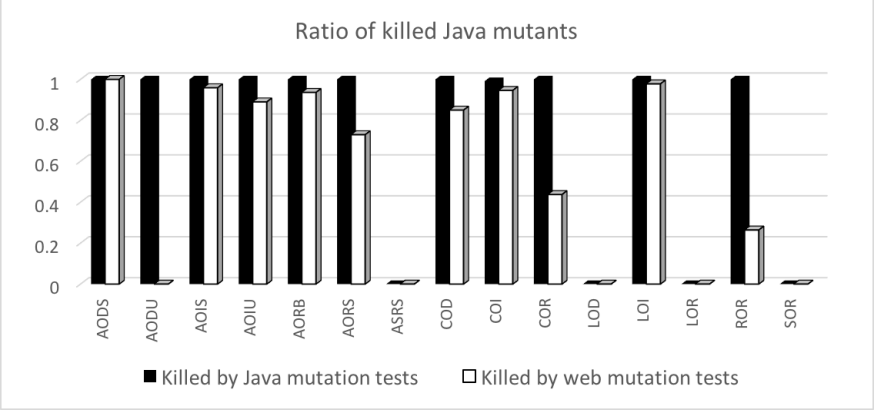

Figure 5.12: Ratio of killed Java mutants by operators (number of killed Java mutants / number of generated Java mutants)

5.4.4 Threats to Validity

Similar to any other research, this experiment has some limitations that could have influenced the experimental results. Some of these limitations may be minimized or avoided while some may be inevitable.

Internal validity: This experiment relies on tests manually created by only one person. As the quality of tests depends upon the testers' experience, the results may differ

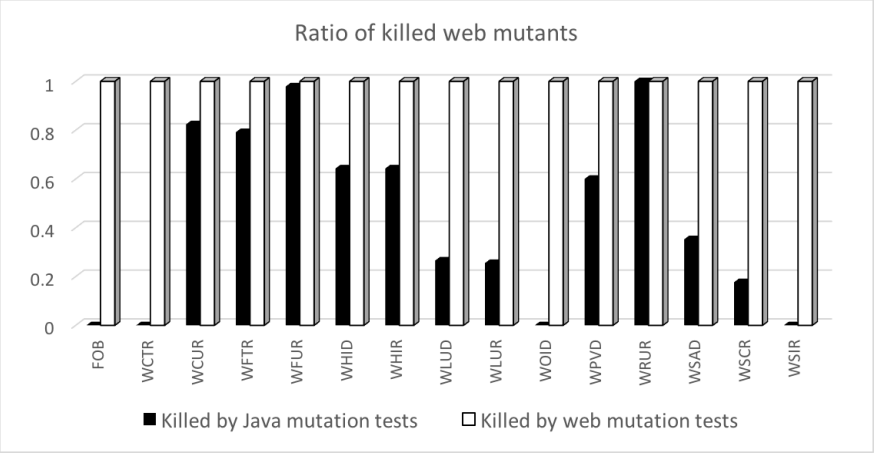

Figure 5.13: Ratio of killed web mutants by operators (number of killed web mutants / number of generated web mutants)

with different tests. Furthermore, some of the computation and analysis such as identifying equivalent mutants was performed by hand and hence may introduce human errors.

External validity: Though the subjects used in this experiment contained a variety of web component interactions, it is impossible to guarantee that they are representative. The results may differ with other web apps. This is a threat that is prevalent in almost all software engineering studies.

Construct validity: The experiment assumed that webMuJava worked correctly.

5.5 Experimental Evaluation of Redundancy in Web Mutation Operators

The findings from previous experiments demonstrated that web mutation testing can help create tests that are effective at finding web faults. While mutation testing has been shown to be effective at improving the quality of test cases, the testing costs can be very expensive depending on the number of mutants and the percentage of equivalent mutants generated. To reduce the cost, this research focuses on using effective mutation operators (a *do-fewer*

approach). The goal of this experiment is to analyze the redundancy in web mutation operators to recommend which operators are to exclude; i.e., the fewer the mutants generated, the fewer the tests needed. This experiment answers the following research questions:

RQ7: How frequently can web mutants of one type be killed by tests generated specifically to kill other types of web mutants?

RQ8: Which types of web mutants are seldom killed by tests designed to kill other types of web mutants?

RQ9: Which types of web mutants (and thus the operators that create them) can be excluded from the testing process without significantly reducing fault detection?

To analyze the redundancy in web mutation operators, this experiment applies all operators to generate mutants. All mutants generated by the same operator are said to be of the same *mutation type* (or *type*). If different types of mutants can be killed by the same tests, they are said to be overlapping. That is, there is redundancy in the operators that create them.

5.5.1 Experimental Subjects

Web mutation testing requires that source code is available and some of the analysis are necessarily done by hand. Similar to the previous experiments, this experiment chose subjects that are small enough for reasonable hand analysis, yet large and complex enough to include a variety of interactions among web components. Thus this experiment shared a set of subject web apps used in previous experiments.

Table 5.23 lists the twelve Java-based web apps used in this experiments[11]. Components are JSPs and Java Servlets, excluding JavaScript, HTML, and CSS files. All subjects are available online at http://github.com/nanpj. *BSVoting*, *HLVoting*, and *KSVoting* are online voting systems, which allow users to maintain their assertions and vote on other users' assertions. *check24online* allows users to play the card game 24, where users enter four integer values between 1 and 24 inclusive and the web app generates all expressions that have

[11]LOC refers to non-blank lines of code, measured with Code Counter Pro (http://www.geronesoft.com/)

the result 24. *computeGPA* accepts credit hours and grades and computes GPAs, according to George Mason University's policy. *conversion* allows users to convert measurements. *faultSeeding* facilitates fault seeding and maintains the collection of faulty versions of software. *quotes* allows users to search for quotes using keywords. *randomString* allows users to randomly choose strings with or without replacement. *smallTextInfoSys* allows users to maintain text information, using a MySQL database. *StudInfoSys* allows users to maintain student information. *webstutter* allows users to check for repeated words in strings. All these subjects use features that affect the interactions between web components and the states of web apps, including form submission, redirect control connection, forward control connection, include directive, and state management mechanisms. These subjects consist of combinations of JSPs, Java servlets, Java Beans, JavaScripts, HTMLs, XMLs, CSS files, and images. JavaScript, XML, CSS, and images are out of the research's scope. Therefore, they were excluded from the experiment, leaving the number of components and LOC as displayed in the table.

Table 5.23: Subject web apps

Subjects	**Components**	**LOC**
BSVoting (S1)	11	930
check24online (S2)	1	1619
computeGPA (S3)	2	581
convert (S4)	1	388
faultSeeding (S5)	5	1541
HLVoting (S6)	12	939
KSVoting (S7)	7	1024
quotes (S8)	5	537
randomString (S9)	1	285
smallTextInfoSys (S10)	24	1103
studInfoSys (S11)	3	1766
webstutter (S12)	3	126
Total	**75**	**10,839**

5.5.2 Experimental Procedure

The *independent* variables are web mutation operators (as presented in Section 4.2). The *dependent* variables are the number of equivalent mutants, the number of test cases required,

the number of mutants (generated from each operator) killed by each test set, and the redundancy among web mutation operators.

The experiment was conducted in four steps:

1. **Generate mutants**: For each subject, all fifteen operators were applied. This yields sets of mutants M_{ij}, that is, mutants of the i^{th} subject created by the j^{th} operator.

2. **Generate tests**: Tests were designed independently and specifically to kill all mutants of each types for each subject. The set T_{ij} contains tests of the i^{th} subject that target the j^{th} mutation type. Test cases were created manually as sequences of requests and were written in HtmlUnit, JWebUnit, and Selenium.

3. **Execute tests**: For each subject, this experiment executed each test set on all mutants of the subject. $N(T_{ij}, M_{ik})$ represents the number of mutants of the k^{th} type of the i^{th} subject that were killed by test T_{ij}. Equivalent mutants were identified by hand and excluded from the testing process. For each mutation type of the subject, the mutation score was computed as the ratio between the number of killed mutants and the total number of non-equivalent mutants. This experiment kept adding tests until all mutants of the given type were killed; i.e., the mutation score is 1. These tests were referred to as *mutation-adequate tests*.

4. **Compute the redundancy of the operator**: The redundancy of the j^{th} operator, R_{T_j}, is computed as the percentage of the mutants of the i^{th} type (m_i) that are killed by tests designed specifically to kill mutants of the j^{th} operator, T_j: $R_{T_j} = \frac{m_i}{M_i} \times 100$.

5.5.3 Experimental Results and Analysis

This section presents an overview of mutants generated and killed. Since equivalent mutants impact the overall mutation testing cost, this section analyzes and discusses the equivalent mutants across all 12 subjects. Then it presents detailed analysis on the killed mutants,

discusses the redundancy in web mutation operators, and finally concludes by responding to the research questions.

Overview of mutants generated and killed

Table 5.24 summarizes the number of mutants generated, the number of equivalent mutants, the number of mutants killed, and the number of tests created by subjects. Figure 5.14 provides a visual summary of non-equivalent mutants of each mutation type.

Table 5.24: Summary of web mutants generated and killed

Subjects	**Mutants**	**Equivalent**	**Killed**	**Tests**
S1	27	0	27	26
S2	8	0	8	8
S3	18	0	18	17
S4	3	0	3	3
S5	115	16	99	65
S6	103	3	100	99
S7	34	0	34	30
S8	11	0	11	10
S9	5	0	5	5
S10	205	15	190	140
S11	9	0	9	9
S12	4	0	4	4
Total	**542**	**34**	**508**	**416**

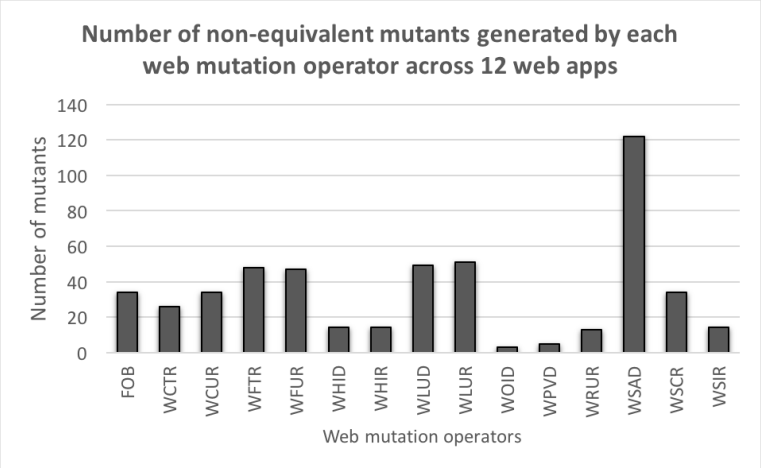

Figure 5.14: Non-equivalent web mutants generated

All 15 web mutation operators were applied to 12 subject web apps, generating a total of 542 mutants. Tests were designed and killed 508 mutants. With extensive hand analysis, this experiment determined that the other 34 mutants were equivalent (6.3%). Twenty-two equivalent mutants were of type WSAD, two were of type WSIR, four were of type WHID, four were of type WHIR, one was of type WLUD, and one was of type WLUR. All equivalent mutants were removed after being identified.

Analysis of equivalent mutants

Six WSAD equivalent mutants (one in subject S6 and five in subject S10) mutated attributes of sessions that the web apps never accessed. Hence, removing the attribute setting statements had no affect on the subjects' behaviors. The other 16 equivalent WSAD mutants, in subjects S5 and S10, mutated attributes that were reset being being accessed. This suggested that the developers might have unnecessarily set the session attributes. Straightforward static analysis could have found those mutants to be equivalent, and might even indicate unnecessary code or attributes that are not needed.

The two equivalent WSIR mutants were due to a session initialization in components of subject S6. These mutants modified a session initialization behavior (specified as a `boolean` parameter of the `getSession()` method) from `request.getSession(false)` to `request.getSession(true)`. Instead of **not** creating a new session if none exists, the mutated code created a new session if none exists. However, because these components were included in another component, the session object was managed by its container. As a result, these WSIR mutants had no impact on the subject's behavior. This suggests that the WSIR mutant is useful only when the mutated code is **not** included in another component.

The four equivalent WHID mutants and four equivalent WHIR mutants were in subject S10. Three of these equivalent WHID mutants and three of these equivalent WHIR mutants changed values that were either not used or were being deleted from the database. The others changed hidden form fields of non-keys of records that were to be sorted. Therefore, replacing or removing values of these hidden form fields did not affect the subject's behavior.

The equivalent WLUD and WLUR mutants in subject S6 involved changes to links to CSS files. As presentation checking was out of scope for this research, these mutants were excluded from study.

Detailed analysis of killed mutants and redundancy

Table 5.25 displays data from running the tests on mutants. The upper half summarizes the number of mutants generated from each operator for each subject (as displayed in the columns below *Mutants*) along with the total numbers of mutants by subject by operator. For example, webMuJava generated one FOB mutant, seven WCTR mutants, and 27 total mutants for subject S1.

The bottom half of Table 5.25 presents data on killing mutants. The column *Tests* shows the number of tests needed to kill all mutants of each type. That is, across all 12 subject web apps, the total of 34 tests were created to kill all FOB mutants. This test set is called **test_FOB**, and is listed on the left. The *Killed mutants* columns give the number of mutants that were killed by the test set that killed all of the mutants of the type on the left. For example, 24 tests were needed to kill all 26 of the WCTR mutants. Those same tests killed 12 WRUR mutants and 8 WSAD mutants. For several operators, the number of tests is smaller than the number of mutants killed because some tests killed more than one mutant. That is, the numbers on the diagonal are at least as big as the numbers on the left under *Tests*.

To obtain these numbers, for each subject, this experiment executed each test set on all mutants and recorded the number of killed mutants of each type. This was a total of 39,847 executions. Table 5.26 shows the overall redundancy of each operator based on the formula in Subsection 5.5.2. The overall redundancy of each operator is based on cumulative numbers of killed mutants of all 12 subjects and is computed as $\sum_{1 \leq i \leq 12} m_i / \sum_{1 \leq i \leq 12} M_i$ where m_i denotes the number of mutants of the i^{th} operator killed by tests designed specifically to kill mutants of the j^{th} operator, and M_i denotes the total number of mutants of the i^{th} operator. The diagonal cells are 1, indicating the tests adequate to kill mutants created from

Table 5.25: Number of mutants generated and killed

Subject	Mutants															**Total**
	FOB	WCTR	WCUR	WFTR	WFUR	WHID	WHIR	WLUD	WLUR	WOID	WPVD	WRUR	WSAD	WSCR	WSIR	
S1	1	7	0	5	5	0	0	0	0	0	0	3	5	0	1	27
S2	1	0	0	1	1	0	0	2	3	0	0	0	0	0	0	8
S3	1	0	0	1	1	2	2	1	1	0	0	0	6	3	0	18
S4	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	3
S5	2	3	3	4	3	0	0	1	1	0	2	1	76	0	3	99
S6	4	9	0	12	12	5	5	9	9	1	0	9	17	0	8	100
S7	3	6	0	7	7	0	0	0	0	0	0	0	11	0	0	34
S8	1	0	0	2	2	0	0	2	2	0	0	0	2	0	0	11
S9	1	0	0	1	1	0	0	0	0	2	0	0	0	0	0	5
S10	16	0	31	12	12	7	7	32	32	0	3	0	5	31	2	190
S11	2	0	0	1	1	0	0	2	3	0	0	0	0	0	0	9
S12	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	4
Total	34	26	34	48	47	14	14	49	51	3	5	13	122	34	14	508

	Tests	**Killed mutants**															**Total**
test_FOB	34	34	0	0	0	0	0	0	0	0	0	0	0	0	0	0	34
test_WCTR	24	0	26	0	0	0	0	0	0	0	0	0	12	8	0	0	46
test_WCUR	31	0	0	34	0	0	0	0	0	0	0	0	0	3	0	0	37
test_WFTR	47	0	2	13	48	46	9	9	0	0	0	3	0	0	0	0	130
test_WFUR	47	0	2	13	41	47	12	12	1	1	0	3	0	0	0	0	132
test_WHID	10	0	0	0	0	10	14	14	1	1	0	0	0	0	0	0	40
test_WHIR	11	0	0	0	0	11	14	14	1	1	1	0	0	2	2	0	46
test_WLUD	49	0	0	0	1	2	2	2	49	49	0	0	0	0	0	0	105
test_WLUR	51	0	0	0	2	3	2	2	49	51	0	0	0	0	0	0	109
test_WOID	3	0	0	0	0	1	0	0	0	0	3	0	0	0	0	0	4
test_WPVD	2	0	0	0	0	1	0	0	0	0	0	5	0	0	0	0	6
test_WRUR	13	0	12	0	0	0	0	0	0	0	0	0	13	8	0	0	33
test_WSAD	57	0	9	0	6	8	4	0	0	0	0	0	9	122	0	0	158
test_WSCR	23	0	0	0	7	8	6	5	0	0	0	3	0	2	34	0	65
test_WSIR	14	3	2	9	4	4	1	1	6	6	0	0	4	9	0	14	63

operator j. Figure 5.15 provides a visual summary of overall redundancy of web mutation operators.

Some types of mutants are not generated for some subject web apps because they lack the features being mutated. Some features, such as hidden form fields and readonly inputs, are used quite rarely.

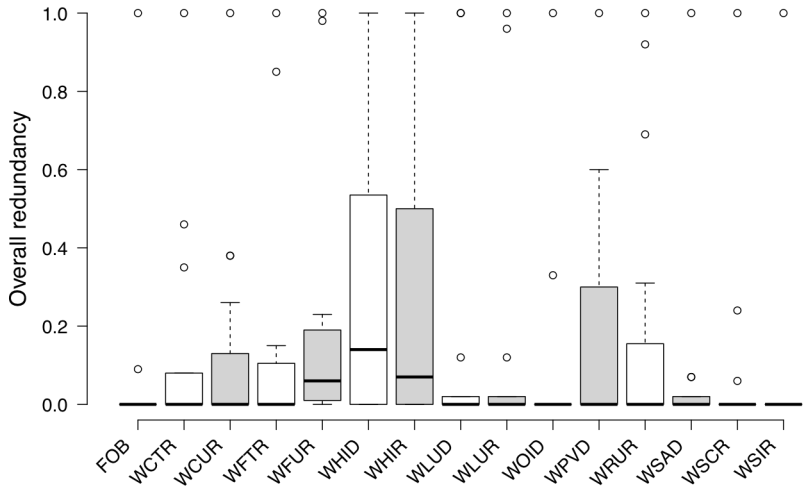

Figure 5.15: Overall redundancy of web mutation operators

To analyze the overlap between web mutation operators, this experiment considers how effective each set of tests kills other types of mutants for each subject. The higher the percentage of mutants killed by tests designed for other types of mutants, the more likely the operator that generates them is redundant.

The average redundancy is obtained from an average of $\sum_{1 \leq i \leq j} \frac{m_i}{M_i}$, where j is the number of subjects containing the type of mutants being considered. That is, for subject S_k, if there exists tests T_j and mutants M_i, tests T_j are executed on M_i and the number of mutants killed by the tests (m_i) is recorded. The effectiveness of tests T_j on mutants M_i for subject S_k is then computed, reflecting the redundancy of the mutation operator i. This experiment computes the effectiveness for all subjects to obtain an average effectiveness of tests T_j on mutants M_i, giving the average redundancy of the operator i. Otherwise, for

Table 5.26: Overall redundancy of web mutation operators

	Killed Mutants														
	FOB	WCTR	WCUR	WFTR	WFUR	WHID	WHIR	WLUD	WLUR	WOID	WPVD	WRUR	WSAD	WSCR	WSIR
test_FOB	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
test_WCTR	0	1	0	0	0	0	0	0	0	0	0	0.92	0.07	0	0
test_WCUR	0	0	1	0	0	0	0	0	0	0	0	0	0.03	0	0
test_WFTR	0	0.08	0.38	1	0.98	0.50	0.50	0	0	0	0.60	0	0	0	0
test_WFUR	0	0.08	0.38	0.85	1	0.93	0.93	0.02	0.02	0	0.60	0	0	0	0
test_WHID	0	0	0	0	0.21	1	1	0.02	0.02	0	0	0	0	0	0
test_WHIR	0	0	0	0	0.23	1	1	0.02	0.02	0.33	0	0	0.02	0.06	0
test_WLUD	0	0	0	0.02	0.04	0.14	0.14	1	0.96	0	0	0	0	0	0
test_WLUR	0	0	0	0.04	0.06	0.14	0.14	1	1	0	0	0	0	0	0
test_WOID	0	0	0	0	0.02	0	0	0	0	1	0	0	0	0	0
test_WPVD	0	0	0	0	0.04	0	0	0	0	0	1	0	0	0	0
test_WRUR	0	0.46	0	0	0	0	0	0	0	0	0	1	0.07	0	0
test_WSAD	0	0.35	0	0.13	0.17	0.29	0	0	0	0	0	0.69	1	0	0
test_WSCR	0	0	0	0.15	0.17	0.43	0.36	0	0	0	0.60	0	0.02	1	0
test_WSIR	0.09	0.08	0.27	0.08	0.09	0.07	0.07	0.12	0.12	0	0	0.31	0.07	0	1

all subjects, if neither T_j nor M_i exists, the average effectiveness of tests T_j on mutants M_i is recorded as unable to determine if the given test set is effective at killing this type of mutants (n/a). The average redundancy of each operator is presented in Table 5.27 and its visual summary is displayed in Figure 5.16.

RQ7: How frequently can web mutants of one type be killed by tests generated specifically to kill other types of web mutants?

The experimental results show that some types of mutants were often killed by tests that were not designed specifically to kill them.

On average, almost all WFUR mutants were killed by WFTR-adequate tests (test_WFTR in Table 5.27). Both WFUR mutants and WFTR mutants are created by modifying `<form>` tags. It seems logical that tests adequate to kill either group should also be effective at killing another group. However, this experiment observed that test_WFUR misses WFTR mutants when a blank form or a form that does not allow data entry is submitted. Test_WFTR misses WFUR mutants when the mutated URL[12] causes a similar HTML response. Note that to determine whether a test distinguishes a mutant from the original app, the HTML responses of the mutant and of the original app are compared.

Almost all WHID and WHIR mutants were killed by test_WFUR. Since WHID and WHIR mutants manipulate the subject's hidden inputs, form submissions that use these hidden inputs will affect the subject's behavior.

The test set test_WLUR kills all WLUD mutants. The WLUD and the WLUR operators mutate `<a>` tags, so it is reasonable to expect tests that exercise the `<a>` tag will distinguish WLUD and WLUR mutants from the original app. In the experiment, however, test_WLUD missed two WLUR mutants. There was no WLUD mutant for the `<a>` tags of these two WLUR mutants, and thus no corresponding tests. This is because applying the WLUD operator to these particular `<a>` tags would have created equivalent mutants, and thus were

[12]The mutation tool *webMuJava* extracts all URLs used in the subject web app and statistically records their frequency of reference. The URL that is used the most frequently is used for URL replacement. If the most frequently used URL is the same as the original, the second most frequently used URL is used.

Table 5.27: Average redundancy of web mutation operators

	Killed Mutants														
	FOB	WCTR	WCUR	WFTR	WFUR	WHID	WHIR	WLUD	WLUR	WOID	WPVD	WRUR	WSAD	WSCR	WSIR
test_FOB	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
test_WCTR	0	1	0	0	0	0	0	0	0	0	0	0.67	0.12	*n/a*	0
test_WCUR	0	0	1	0	0	0	0	0	0	*n/a*	0	0	0.30	0	0
test_WFTR	0	0.07	0.21	1	0.99	0.33	0.33	0	0	0	0.50	0	0	0	0
test_WFUR	0	0.07	0.21	0.65	1	0.93	0.93	0.14	0.14	0	0.50	0	0	0	0
test_WHID	0	0	0	0	0.58	1	1	0.33	0.33	1	0	0	0	0	0
test_WHIR	0	0	0	0	0.61	1	1	0.33	0.33	1	0	0	0.11	0.33	0
test_WLUD	0	0	0	0.14	0.29	0.33	0.33	1	0.91	0	0	0	0	0	0
test_WLUR	0	0	0	0.29	0.43	0.33	0.33	1	1	0	0	0	0	0	0
test_WOID	0	0	*n/a*	0	0.50	0	0	0	0	1	*n/a*	0	0	*n/a*	0
test_WPVD	0	0	0	0	0.21	0	0	0	0	*n/a*	1	0	0	0	0
test_WRUR	0	0.48	0	0	0	0	0	0	0	0	0	1	0.16	*n/a*	0
test_WSAD	0	0.25	0	0.16	0.19	0.27	0	0	0	0	0	0.33	1	0	0
test_WSCR	0	*n/a*	0	0.29	0.79	0.79	0.54	0	0	*n/a*	1	*n/a*	0.17	1	0
test_WSIR	0.05	0.07	0.15	0.20	0.20	0.10	0.10	0.06	0.06	0	0	0.15	0.20	0	1

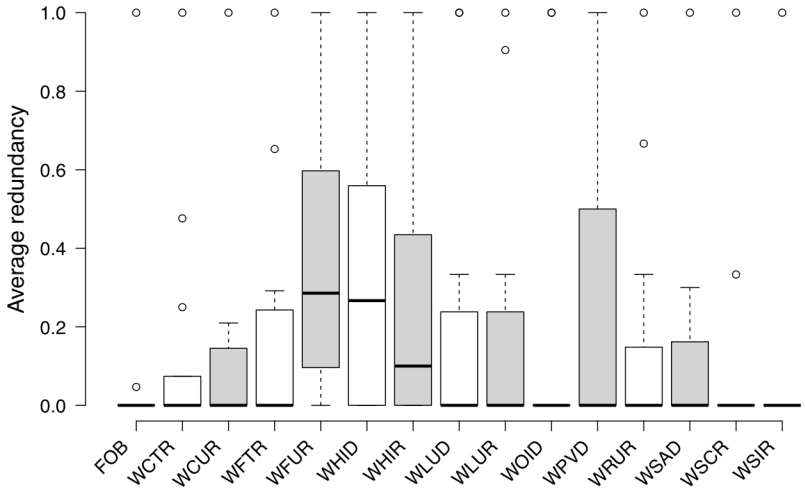

Figure 5.16: Average redundancy of web mutation operators

not generated.

RQ8: Which types of web mutants are seldom killed by tests designed to kill other types of web mutants?

The experimental results show that no tests adequate for other types of mutants kill WSIR mutants. One possible reason is that WSIR mutants manipulate the existence of the session object that maintains information about the user and client. Tests must invalidate the session to kill WSIR mutants. For this experiment, other types of mutants can be killed without exercising the validity of the session. The experimental result suggests that WSIR mutants are important to improve the quality of tests.

The other types of mutants were sometimes killed by tests designed to kill other types of mutants. However, the effectiveness varies.

A very small number of FOB mutants were killed by test_WSIR. The investigation revealed that the test_WSIR tests that killed FOB mutants imitate clicking of the browser back button after intentionally terminating the session. This experiment infers that other tests did not kill FOB mutants because the other tests do not exercise the browser `back` button.

While it is possible to kill FOB mutants with tests designed for other types of mutants, testers may have overlooked some browser features. Most test cases do not verify if the app under test behaves properly when the back button is used. Therefore, using FOB mutants can potentially improve the quality of tests.

RQ9: Which types of web mutants (and thus the operators that create them) can be excluded from the testing process without significantly reducing fault detection?

On average, test_WFTR was 99% effective at killing WFUR mutants, while test_WFUR was only 65% effective at killing WFTR mutants. This suggests an overlap between WFTR mutants and WFUR mutants. Therefore, this experiment concludes that WFUR mutants can be excluded without significantly reducing fault detection capability.

Test_WHID was 100% effective at killing WHIR mutants and test_WHIR was 100% effective at killing WHID mutants. This implies a strong overlap between WHID and WHIR mutants. To determine which mutants can be removed with minimal loss in fault detection capability, this experiment considered their effectiveness at killing other types of mutants. It is important to note that because the WHID and WHIR operators mutate relatively rarely used elements, only 14 WHID and 14 WHI mutants were generated on three subjects. On average, test_WHIR was more effective at killing mutants of other types than test_WHID was. Hence, WHIR mutants are stronger than WHID mutants. This experiment concludes that WHID mutants can be excluded.

WLUD and WLUR mutants also overlapped. Tests adequate to kill WLUD mutants were 91% effective at killing WLUR mutants, and tests adequate to kill WLUR mutants were 100% effective at killing WLUD mutants. In addition, test_WLUD was less effective at killing other types of mutants than test_WLUR was. This suggests that WLUD mutants can be excluded.

Additionally, the experimental results indicate that WOID can be excluded. As shown in Table 5.25, only three WOID mutants were generated on two subjects. This implies

that not many web apps contain a readonly input. 100% of WOID mutants were killed by test_WHID and test_WHIR. Although this could indicate that WOID mutants can be excluded, the savings would be small.

Similar to WOID mutants, in this experiment, only one subject had WPVD mutants, all of which were killed by tests adequate to kill WSCR mutants. Again, it might be safe to exclude WPVD mutants, but the savings would be small.

Another interesting pair of mutant types that might overlap are WCTR and WRUR mutants. Test_WCTR, on average, killed 67% of WRUR mutants and test_WRUR killed 48% of WCTR mutants. Although this is significant overlap, again, the numbers are relatively small.

In conclusion, the experiment recommends excluding the WFUR, WHID, and WLUD operators. Because the numbers of WOID, WPVD WCTR, and WRUR mutants generated are small and the savings would be minimal, the experimental results are not strong enough to recommend removing the operators that create them.

5.5.4 Threats to Validity

As in all software engineering studies, this study has limitations that could have influenced the experimental results. Some of these limitations were minimized or avoided while some may be unavoidable.

Internal validity: The quality of tests may vary depending upon the testers' experience. This experiment relies on tests manually created by only one person. The results may differ with different tests. Another potential threat is that some of the computation and analysis such as identifying equivalent mutants was performed by hand and hence may introduce human errors.

External validity: Though the subjects used in this experiment contained a variety of web component interactions, it is impossible to guarantee that they are representative. The results may differ with other web apps. This is a threat that is prevalent in almost all software engineering studies. Technologically, this experiment was limited to asynchronous

Java-based web apps.

Construct validity: The experiment assumed that webMuJava worked correctly.

Chapter 6: Conclusions and Future Work

This chapter revisits the research problems and the RQs and draws on the findings to verify the research hypothesis (Section 6.1). The chapter then restates the contributions (Section 6.2). Finally, the chapter concludes with future research directions (Section 6.3).

6.1 Research Problem and RQs Revisited

Web apps continue to have widespread failures. Traditional software testing techniques are insufficient for testing web apps due to the nature of web app technologies. Improperly implementing and testing the communications between web components is a common source of faults in web apps. While many new technologies have been created to develop and enhance the functionality of web apps, they introduce new challenges in testing. This research investigated and classified seven challenges in testing web apps and categorized web faults. Based on the fault model, the research designed web mutation testing with the goal to improve the quality of web apps and reduce the cost of testing web apps. The research hypothesis

> ***Mutation testing can be used to reveal more web interaction faults than existing testing techniques can in a cost-effective manner.***

was validated with nine RQs. Four experiments were conducted to answer these RQs.

The first experiment (Section 5.2) verified whether web mutation testing can help improve the quality of tests developed with traditional testing criteria (addressing RQ1 and RQ2). The experiment evaluated the usefulness of web mutation operators based on the number of killed mutants of each operator.

- **RQ1**: How well do tests designed for traditional testing criteria kill web mutants?
  - The experiment revealed that none of the traditional tests were able to kill a high percentage of web mutants. The mutation scores ranged from 17% to 75%, with a mean of only 47%. The wide range of mutation scores suggested that some testers produced much higher quality test sets than others.
- **RQ2**: Can hand-designed tests kill web mutants?
  - The web mutation-based tests killed all the mutants generated in the experiment. Therefore, to answer this question, hand-designed tests can kill web mutants. Designing tests with web mutation testing can improve the quality of tests.

The second experiment (Section 5.3) examined the applicability of web mutation testing to detecting web faults (addressing RQ3 and RQ4). The experiment evaluated the mutation scores from executing the web mutation-based tests on hand-seeded faults. The experiment also identified the characteristics of faults that were detected.

- **RQ3**: How well do tests designed for web mutation testing reveal web faults?
  - The web mutation-based tests detected 68% to 100% of hand-seeded faults, with an average fault detection of 85%.
- **RQ4**: What kinds of web faults are detected by web mutation testing?
  - Web mutation testing detected various kinds of hand-seeded faults. The majority of faults detected were faults due to incorrect relational operators and incorrect arithmetic operation. These kinds of faults directly impact the main functionality of web apps. The web mutation-based tests that verified the apps' functionality could easily detect these faults. Many other detected faults involved incorrect conditional operators and accessing non-existent form elements.

The third experiment (Section 5.4) studied whether web mutation testing and traditional Java mutation testing are complementary (addressing RQ5 and RQ6). The experiment

executed tests designed with web mutation testing on Java mutants and executed tests designed with traditional Java mutation testing on web mutants, and then analyzed the number of killed mutants.

- **RQ5**: How well do tests designed for web mutants kill traditional Java mutants and tests designed for traditional Java mutants kill web mutants?
  - The mutation scores of running the web mutation-based tests on Java mutants ranged from 38% to 87%, with a mean of 66%. The percentages of Java mutants of each operator killed by the web mutation-based tests ranged from 26% to 98%. The findings excluded the percentages of killed AODS and AODU mutants because too few mutants were created (only one AODS mutant and one AODU mutant).
  - The mutation scores of running the Java mutation-based tests on web mutants ranged from 0% to 67%, with a mean of 41%. The percentages of web mutants of each operator killed by the Java mutation-based tests ranged from 0% to 100%.
  - The wide range of mutation scores suggested that some kinds of mutants were easily detected while some were not.
- **RQ6**: How much do web mutants and traditional Java mutants overlap?
  - The experiment revealed an overlap between web mutants and Java mutants, especially the mutants that had direct impact on the main functionality of the subjects under test. However, some kinds of web mutants were left undetected. The experimental results suggested that Java mutants helped design tests that checked all form inputs and verified individual web components whereas web mutants helped design tests that verified interactions between web components. Using both Java mutants and web mutants can improve the quality of tests.

While powerful, one major concern when applying mutation analysis is cost, and a major factor in cost is the number of mutants. Every additional mutant increases both computational and human cost. The last experiment (Section 5.5) concentrated on decreasing the testing cost by reducing the number of mutants generated (addressing RQ7, RQ8, and RQ9). For each subject web app, the experiment designed 15 sets of tests adequate to kill 15 types of mutants (i.e., 15 mutation operators), executed them on all mutants, and computed an average effectiveness of each set of tests on each type of mutants. The effectiveness of the test sets were used to analyze redundancy among web mutation operators.

- **RQ7**: How frequently can web mutants of one type be killed by tests generated specifically to kill other types of web mutants?
  - The overall redundancy of web mutation operators ranged from 0% to 25%, with a mean of only 9%. The other tests did not kill WSIR mutants while WSIR-adequate tests killed several other types of mutants. This is encouraging because it indicates that WSIR mutants may be particularly strong.
- **RQ8**: Which types of web mutants are seldom killed by tests designed to kill other types of web mutants?
  - No tests adequate for other types of mutants killed WSIR mutants. WSIR-adequate tests, the only tests of all fifteen kinds of web mutants, killed very few FOB mutants (5%). The other types of mutants were sometimes killed by tests designed to kill other types of mutants but the redundancy varied.
- **RQ9**: Which types of web mutants (and thus the operators that create them) can be excluded from the testing process without significantly reducing fault detection?
  - The experimental results strongly indicated that three mutation operators were largely redundant and could be removed with minimal loss in the fault detection capability: WFUR, WHID, and WLUD.

In conclusion, since the design of web mutation operators was based on interaction faults derived according to the seven challenges, the findings confirmed that tests generated for web mutation testing can reveal more interaction faults than existing testing techniques can.

6.2 Summary of Contributions

With the ultimate goal to improve the quality of web apps and reduce the testing cost by using effective web mutation operators for test case design and generation, the global contribution of this research is the testing criterion for web apps, specifically a set of web mutation operators. The specific contributions are listed below:

- Classified challenges in testing web apps
- Modeled faults in web apps to ensure interaction fault coverage
- Defined a set of web mutation operators
- Implemented a web mutation testing tool
- Experimentally evaluated the effectiveness of web mutation testing to improve the quality of tests designed with traditional testing criteria
- Experimentally examined the applicability of web mutation testing to detect web faults
- Experimentally investigated an overlap between web mutation testing and traditional Java mutation testing, and identified whether they can be complementary
- Experimentally analyzed the redundancy in web mutation operators to provide recommendation for cost reduction

This research retrofitted web-specific mutation operators into the Java mutation testing system, hence developing a web mutation testing tool. Most of these implementation ideas can be transferred into other mutation testing tools with slight modification.

Throughout this dissertation, the design details of web mutation operators were discussed in terms of J2EE-based web apps. The underlying concepts of the operators can be applied to other web development languages and frameworks with slight modifications.

The list below shows the publications based on this dissertation:

- Upsorn Praphamontripong and Jeff Offutt. Finding Redundancy in Web Mutation Operators. 13^{th} IEEE Workshop on Mutation Analysis. Tokyo, Japan, April 2017.
- Upsorn Praphamontripong, Jeff Offutt, Lin Deng, and JingJing Gu. An Experimental Evaluation of Web Mutation Operators. 11^{th} IEEE Workshop on Mutation Analysis. Chicago IL, April 2016.
- Upsorn Praphamontripong. Web Mutation Testing. The Ph.D. Symposium of 5^{th} IEEE International Conference on Software Testing, Verification and Validation. Montreal, Quebec, Canada. April 2012.
- Upsorn Praphamontripong and Jeff Offutt. Applying Mutation Testing to Web Applications. 6^{th} Workshop on Mutation Analysis, Paris, France, April 2010.

The list below shows my other publications:

- Jeff Offutt, Vasileios Papadimitriou, and Upsorn Praphamontripong. A Case Study on Bypass Testing of Web Applications. Springer's Empirical Software Engineering, 19(1):69-104, 2014.
- Garrett Kent Kaminski, Upsorn Praphamontripong, Paul Ammann, and Jeff Offutt. A Logic Mutation Approach to Selective Mutation for Programs and Queries. Information and Software Technology, 53(10):1137-1152, 2011.
- Garrett Kent Kaminski, Upsorn Praphamontripong, Paul Ammann, Jeff Offutt. An Evaluation of the Minimal-MUMCUT Logic Criterion and Prime Path Coverage. Software Engineering Research and Practice, 205-211, 2010.

- Nan Li, Upsorn Praphamontripong and Jeff Offutt. An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-uses and Prime Path Coverage. 5^{th} Workshop on Mutation Analysis, Denver, Colorado, April 2009.

- Upsorn Praphamontripong, Swapna Gokhale, Aniruddha Gokhale, and Jeff Gray. An Analytical Approach to Performance Analysis of an Asynchronous Web Server. Simulation: Transactions of the Society for Modeling and Simulation, 83(8):571-586, August 2007.

- Upsorn Praphamontripong, Swapna Gokhale, Aniruddha Gokhale, and Jeff Gray. Performance Analysis of a Middleware Demultiplexing Pattern. 40^{th} Hawaiian International Conference on System Sciences (HICSS), Big Island, Hawaii, January 2007.

- Upsorn Praphamontripong, Swapna Gokhale, Aniruddha Gokhale, and Jeff Gray. Performance Analysis of an Asynchronous Web Server. 30^{th} Annual International Computer Software and Applications Conference, September 2006.

- Swapna Gokhale, Aniruddha Gokhale, Jeff Gray, Paul Vandal, Upsorn Praphamontripong. Performance Analysis of the Reactor Pattern in Network Services. 5^{th} Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems, Rhodes Island, Greece, April 2006.

- Arundhati Kogekar, Dimple Kaul, Aniruddha Gokhale, Paul Vandal, Upsorn Praphamontripong, Swapna Gokhale, Jing Zhang, Yuehua Lin, Jeff Gray. Model-driven Generative Techniques for Scalable Performability Analysis of Distributed Systems. Next Generation Software Workshop, held at IPDPS, Rhodes Island, Greece, April 2006.

- Upsorn Praphamontripong and Gongzhu Hu. XML-Based Software Component Retrieval with Partial and Reference Matching. IEEE International Conference on Information Reuse and Integration, Las Vegas, Nevada, November 2004.

6.3 Future Research Directions

This dissertation believes that web mutation testing can be complementary to other testing techniques and can be further turned and augmented to target other web-specific features and technologies. In addition to the web mutation testing concept, although functional, the web mutation testing tool is subject to further improvement. The research described in this document can be continued in several directions:

- Web apps are built with multiple technologies, both synchronous and asynchronous. This requires more mutation operators and complicates tool building. This research plans to expand the web mutation operator set to test for asynchronous faults (based on JavaScript and AJAX) and other programming languages such as PHP. Several ideas from the current set of operators such as URL manipulation and parameter mismatch may be applied to other frameworks; for instance, calling an unintended function in JavaScript, swapping parameters, mis-typing parameters, or dropping parameters in JavaScript function calls.

- The experiment described in Section 5.3 evaluated fault detection ability of web mutation testing. Future work should examine the correlation between the kinds of faults and the operators, thus being useful when substituting real faults with mutants in software testing research.

  The fault study can be further analyzed to understand the severity of faults. Fault severity information can be useful for maintenance; i.e., deciding which faults should be fixed immediately and which can be postponed. Future work should include metrics to determine the correlation between the severity of faults and the effectiveness of the mutation operators. This information can also be useful when incorporating into an automated software repair system.

  Moreover, the correlation may be useful in software reliability engineering research, which analyzes the factors that lead to software failure and estimates the mean time to failures. Using the correlation that signifies the severity impacts and the effectiveness

of the operators, the estimate can be done via a series of simulations.

- The experiment described in Section 5.4 generated Java mutants using only method-level mutation operators. Future work should include class-level mutation operators.

- The experiment described in Section 5.5 raised questions about the WOID, WPVD, WCTR, and WRUR operators. While not strong enough to be definitive, the results indicate that these operators at least create many redundant operators. Although it may not be possible to exclude the operators completely, the "personalized," or "tailored" mutation approach of Kurtz et al. may help further reduce the cost of web mutation [52].

- The current web mutation testing tool relies on string comparison to determine whether mutants are killed. Identifying killed mutants can be improved with better heuristics.

- Heuristic approaches to automatically identify equivalent mutants should be implemented in the mutation testing system.

- The `failOnReload` mutation operator, designed after the completion of the research experiments, needs to be validated.

- The current web mutation testing tool is semi-automated. While mutant generation and execution are automated, tests must be designed by hand as sequences of HTTP requests and are automated in Java, HtmlUnit, JWebUnit, and Selenium. For future work, the concepts of neural networks and machine learning may be incorporated to train and recognize certain sequences of HTTP requests and input constraints, assisting in automated test case generation of web mutation testing system.

- The web mutation testing tool currently creates a different source file for each mutated component. A more efficient approach would be to use program schema [104].

Bibliography

Bibliography

[1] *HtmlUnit*. [Online] http://htmlunit.sourceforge.net/, last access February 2017.

[2] *Selenium*. [Online] http://www.seleniumhq.org/, last access February 2017.

[3] *JWebUnit*, 2015. [Online] https://jwebunit.github.io/jwebunit/, last access February 2017.

[4] R. Abraham and M. Erwig. Mutation operators for spreadsheets. *IEEE Transactions on Software Engineering*, 35(1):94–108, Jan 2009.

[5] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *7th IEEE International Conference on Software Testing, Verification, and Validation (ICST 2014)*, pages 21–30, Cleveland, OH, 2014.

[6] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, November 2016. 2nd Edition, ISBN 978-1107172012.

[7] Anneliese A. Andrews, Jeff Offutt, Curtis Dyreson, Christopher J. Mallery, Kshamta Jerath, and Roger Alexander. Scalability issues with using FSMWeb to test web applications. *Information and Software Technology*, 52(1):52–66, January 2010.

[8] Anneliese Amschler Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with FSMs. *Journal of Software and Systems Modeling*, 4(3):326–345, 2005.

[9] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments. In *Proceeding of the International Conference on Software Engineering (ICSE 2005)*, pages 402–411, St. Louis, MO, May 2005.

[10] Laura Batchelor. Bank of America explains website outage, October 2011. [Online] http://money.cnn.com/2011/10/06/news/companies/bank_of_america_website/, last access February 2017.

[11] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In *Proceeding of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 81–88, Washington, DC, 2000. IEEE Computer Society.

[12] Clint Boulton. Google suffers first gmail outage of 2011, February 2011. [Online] http://www.eweek.com/c/a/Messaging-and-Collaboration/Google-Suffers-First-Gmail-Outage-of-2011-850632, last access February 2017.

[13] Stefano Ceri, Florian Daniel, and Federico M. Facca. Modeling web applications reacting to user behaviors. *Computer Networks*, 50(10):1533–1546, 2006.

[14] Kelly Clay. Amazon.com goes down, loses $66,240 per minute, August 2013. [Online] http://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/, last access February 2017.

[15] Alan Cooper and Robert Reimann. *Designing for the Web, About Face 2.0: The Essentials of Interaction Design*. Wiley Publishing, 2003.

[16] U. S. Customs and Border Protection. Ace secure data portal to enhance border security and efficiency. [Online] http://www.cbp.gov/trade/automated, last access February 2017.

[17] Howard Dahdah. Amazon S3 systems failure downs web 2.0 sites, July 2008. [Online] http://www.computerworld.com.au/article/253840, last access February 2017.

[18] Máarcio E. Delamaro, José C. Maldonado, and Aditya P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.

[19] Márcio Eduardo Delamaro, Jeff Offutt, and Paul Ammann. Designing deletion mutation operators. In *7th International Conference on Software Testing, Verification and Validation, (ICST 2014)*, pages 11–20, Cleveland, OH, March 2014.

[20] M.E. Delamaro, Lin Deng, V.H. Serapilha Durelli, Nan Li, and J. Offutt. Experimental evaluation of SDL and one-op mutation for C. In *7th International Conference on Software Testing, Verification and Validation (ICST 2014)*, pages 203–212, March 2014.

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

[22] Richard A. DeMillo and Jeff Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[23] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt. Towards mutation analysis of android apps. In *8th Workshop on Mutation Analysis (Mutation 2015)*, pages 1–10, Graz, Austria, April 2015.

[24] Lin Deng, Jeff Offutt, and Nan Li. Empirical evaluation of the statement deletion mutation operator. In *6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, pages 80–93, Luxembourg, March 2013.

[25] Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, September 2006.

[26] Kinga Dobolyi. *An Exploration of User-Visible Errors in Web-based Applications to Improve Web-based Applications*. PhD thesis, University of Virginia, 2010.

[27] Stacey Ecott. Fault-based testing of web applications. [Online] http://dreuarchive.cra.org/2005/Ecott/paper.pdf, last access February 2017.

[28] Sebastian Elbaum, Kalyan-Ram Chilakamarri, Marc Fisher, II, and Gregg Rothermel. Web application characterization through directed requests. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis (WODA 2006)*, pages 49–56, New York, NY, 2006. ACM.

[29] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering*, pages 49–59, Portland OR, 2003.

[30] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, and Marc Fisher II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, March 2005.

[31] Sandra Camargo Pinto Ferraz Fabbri, José C. Maldonado, Paulo Cesar Masiero, Márcio E. Delamaro, and E. Wong. Mutation testing applied to validate specifications based on Petri nets. In *Proceedings of the IFIP TC6 8th International Conference on Formal Description Techniques VIII*, pages 329–337, London, UK, 1996. Chapman & Hall, Ltd.

[32] Seth Fiegerman. Yahoo says data stolen from 1 billion accounts, December 2016. [Online] http://money.cnn.com/2016/12/14/technology/yahoo-breach-billion-users/index.html?iid=EL, last access February 2017.

[33] Jon Fingas. Dropbox goes down following problem with routine maintenance, January 2014. [Online] http://www.engadget.com/2014/01/10/dropbox-goes-down-following-problem-with-routine-maintenance/, last access February 2017.

[34] Kevin Granville. 9 recent cyberattacks against big businesses, February 2015. [Online] http://www.nytimes.com/interactive/2015/02/05/technology/recent-cyberattacks.html, last access February 2017.

[35] Yuepu Guo and Sreedevi Sampath. Web application fault classification – An exploratory study. In *2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2008)*, pages 303–305, 2008.

[36] William Halfond and Alessandro Orso. Automated identification of parameter mismatches in web applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 181–191, Atlanta, GA, 2008. ACM.

[37] William G. J. Halfond and Alessandro Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE 2007)*, pages 145–154, New York, NY, 2007. ACM.

[38] Matthew 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 February 2017.

[39] Shan-Shan Hou, Lu Zhang, Tao Xie, Hong Mei, and Jia su Sun. Applying interface-contract mutation in regression testing of component- based software. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, ICSM 2007, pages 174–183, Paris, France, October 2007. IEEE.

[40] Rick Hower. Web site test tools and site management tools, 2002. [Online] http://www.softwareqatest.com/qatweb1.html, last access February 2017.

[41] Raj Jain. *The Art of Computer Systems Perfornamce Analysis: Techniques for Experimental Design Measurement, Simulation, and Modeling.* John Wiley & Sons, Canada, 1991. ISBN 0-471-50336-3.

[42] Yue Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sept 2011.

[43] R. Just, G.M. Kapfhammer, and F. Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *5th International Conference on Software Testing, Verification and Validation (ICST 2012)*, pages 720–725, Montréal, Canada, April 2012.

[44] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the International Conference on Automated Software Engineering (ASE 2011)*, pages 612–615, November 9-11 2011.

[45] 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.

[46] Garrett Kent Kaminski, Upsorn Praphamontripong, Paul Ammann, and Jeff Offutt. A logic mutation approach to selective mutation for programs and queries. *Information and Software Technology*, 53(10):1137–1152, 2011.

[47] Gary Kaminski, Paul Ammann, and Jeff Offutt. Improving logic-based testing. *Journal of Systems and Software*, 86(8):2002–2012, August 2013.

[48] Sunwoo Kim, John A. Clark, and John A. McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proceedings of NET.ObjectDays*, pages 9–12, 2000.

[49] D. Kung, C. H. Liu, and P. Hsia. An object-oriented Web test model for testing Web applications. In *Proceeding of IEEE 24th Annual International Computer Software and Applications Conference (COMPSAC 2000)*, pages 537–542, Taipei, Taiwan, October 2000.

[50] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and Lin Deng. Mutant subsumption graphs. In *10th Workshop on Mutation Analysis (Mutation 2014)*, pages 176–185, Cleveland, OH, March 2014. IEEE Computer Society.

[51] Bob Kurtz, Paul Ammann, and Jeff Offutt. Static analysis of mutant subsumption. In *11th Workshop on Mutation Analysis (Mutation 2015)*, April 2015.

[52] Bob Kurtz, Paul Ammann, Jeff Offutt, Marcio E. Delamaro, Mariet Kurtz, and Nida Gökće. Analyzing the validity of selective mutation with dominator mutants. In *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Seattle Washington, USA, November 2016.

[53] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, pages 3–13, Zurich, Switzerland, 2012. IEEE Press.

[54] Suet Chun Lee and Jeff Offutt. Generating test cases for XML-based Web component interactions using mutation analysis. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 200–209, Hong Kong China, November 2001. IEEE Computer Society Press.

[55] Jin-Hua Li, Geng-Xin Dai, and Huan-Huan Li. Mutation analysis for testing finite state machines. In *2nd International Symposium on Electronic Commerce and Security (ISECS 2009)*, volume 1, pages 620–624, May 2009.

[56] Nuo Li, Tao Xie, Maozhong Jin, and Chao Liu. Perturbation-based user-input-validation testing of web applications. *Journal of System Software*, 83(11):2263–2274, November 2010.

[57] Zhao Li and Jeff Tian. Testing the suitability of markov chains as web usage models. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications (COMPSAC 2003)*, pages 356–361, Washington, DC, 2003. IEEE Computer Society.

[58] Andrew Lipsman. Weekly online holiday retail sales in billions. [Online] http://www.comscore.com/Insights/Data-Mine/Weekly-Online-Holiday-Retail-Sales-in-Billions, last access February 2017.

[59] 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.

[60] Chien-Hung Liu. Data flow analysis and testing of JSP-based web applications. *Information and Software Technology*, 48(12):1137–1147, 2006.

[61] G. Di Lucca, A. Fasolino, and F. Faralli. Testing web applications. In *Proceedings of the International Conference on Software Maintenance (ICSM 2002)*, ICSM '02, pages 310–319, Washington, DC, USA, 2002. IEEE Computer Society.

[62] Giuseppe Di Lucca and Massimiliano Di 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.

[63] Yu-Seung Ma, Yong-Rae Kwon, and Jeff 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.

[64] Yu-Seung Ma and Jeff Offutt. Description of method-level mutation operators for java, 2005. [Online] http://cs.gmu.edu/∼offutt/mujava/mutopsMethod.pdf, last access February 2017.

[65] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava : An automated class mutation system. *Wiley's Software Testing, Verification, and Reliability*, 15(2):97–133, June 2005.

[66] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. muJava home page, 2005. [Online] http://cs.gmu.edu/∼offutt/mujava/, last access February 2017.

[67] José Carlos Maldonado, Márcio Eduardo Delamaro, Sandra C. P. F. Fabbri, Adenilso da Silva Simão, Tatiana Sugeta, Auri Marcelo Rizzo Vincenzi, and Paulo Cesar Masiero. Proteum: A family of tools to support specification and program testing based on mutation. In W. Eric Wong, editor, *Mutation Testing for the New Century*, pages 113–116. Kluwer Academic Publishers, 2001.

[68] Nashat Mansour and Manal Houri. Testing web applications. *Information and Software Technology*, 48(1):31–42, January 2006.

[69] Alessandro Marchetto, Filippo Ricca, and Paolo Tonella. Empirical validation of a web fault taxonomy and its usage for fault seeding. In *9th IEEE International Workshop on Web Site Evolution (WSE 2007)*, pages 31–38, Washington, DC, USA, 2007. IEEE Computer Society.

[70] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th International Conference on the World Wide Web (WWW 2007)*, pages 667–676, New York, NY, 2007. ACM.

[71] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling Ajax by inferring user interface state changes. In *Proceedings of the 2008 Eighth International Conference on Web Engineering (ICWE 2008)*, pages 122–134, Washington, DC, USA, 2008. IEEE Computer Society.

[72] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of Ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[73] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient JavaScript mutation testing. In *6th International Conference on Software Testing, Verification and Validation (ICST)*, pages 74–83, March 2013.

[74] T. Mouelhi, Y. Le Traon, E. Abgrall, B. Baudry, and S. Gombault. Tailored shielding and bypass testing of web applications. In *4th International Conference on Software Testing, Verification and Validation (ICST 2011)*, pages 210–219, March 2011.

[75] K. Nishiura, Y. Maezawa, H. Washizaki, and S. Honiden. Mutation analysis for JavaScript web application testing. In *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 159–165, January 2013.

[76] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[77] A.J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *15th International Conference on Software Engineering*, pages 100–107, May 1993.

[78] Jeff Offutt. Scope and handling state in Java server pages. [Online] http://cs.gmu.edu/∼offutt/classes/642/slides/642Lec10b-JSP-stateHandling.pdf, last access February 2017.

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

[80] Jeff Offutt, Vasileios Papadimitriou, and Upsorn Praphamontripong. A case study on bypass testing of web applications. *Empirical Software Engineering*, 19(1):69–104, February 2014.

[81] Jeff Offutt and Roland Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, San Jose, CA, October 2000.

[82] Jeff Offutt and Ye Wu. Modeling presentation layers of web applications for testing. *Software and Systems Modeling*, 9(2):257–280, April 2010.

[83] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong 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.

[84] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Web application bypass testing. In *Proceedings of the 28th International Computer Software and Applications Conference, Workshop on Quality Assurance and Testing of Web-Based Applications (COMPSAC 2004)*, pages 106–109, Hong Kong, China, September 2004. IEEE Computer Society.

[85] Vasileios Papadimitriou. Automating bypass testing for Web applications. Master's thesis, George Mason University, 2006.

[86] Nicole Perlroth. Attacks on 6 banks frustrate customers, September 2012. [Online] http://www.nytimes.com/2012/10/01/business/cyberattacks-on-6-american-banks-frustrate-customers.html, last access February 2017.

[87] Soila Pertet and Priya Narasimhan. Causes of failure in web applications. Technical Report CMU-PDL-05-109, December 2005. [Online] http://repository.cmu.edu/, last access February 2017.

[88] Upsorn Praphamontripong and A. Jefferson Offutt. Applying mutation testing to web applications. In *6th Workshop on Mutation Analysis (Mutation 2010)*, pages 132–141, Paris, France, April 2010.

[89] Upsorn Praphamontripong and Jeff Offutt. Finding redundancy in web mutation operators. In *13th IEEE Workshop on Mutation Analysis (Mutation 2017)*, Tokyo, Japan, April 2017.

[90] Upsorn Praphamontripong, Jeff Offutt, Lin Deng, and JingJing Gu. An experimental evaluation of web mutation operators. In *11th IEEE Workshop on Mutation Analysis (Mutation 2016)*, pages 102–111, Chicago IL, April 2016.

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

[92] Filippo Ricca and Paolo Tonella. Testing processes of web applications. *Annals of Software Engineering*, 14(1-4):93–114, December 2002.

[93] Filippo Ricca and Paolo Tonella. Web testing: A roadmap for the empirical research. In *7th IEEE International Symposium on Web Site Evolution (WSE 2005)*, pages 63–70, 2005.

[94] Sreedevi Sampath, Renee C. Bryce, Gokulanand Viswanath, Vani Kandimalla, and A. Gunes Koru. Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 141–150, Washington, DC, USA, 2008. IEEE Computer Society.

[95] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, and Lori Pollock. Web application testing with customized test requirements – An experimental comparison study. In *Proceedings of International Symposium on Software Reliability Engineering*, pages 266–278. IEEE Computer Society, November 2006.

[96] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock, and Amie Souter Greenwald. Applying concept analysis to user-session-based testing of web applications. *IEEE Transactions on Software Engineering*, 33(10):643–658, October 2007.

[97] K. Seshadri, L. Liotta, R. Gopal, and T. Liotta. A wireless internet application for healthcare. In *Proceedings of the 14th IEEE Symposium on Computer-Based Medical Systems (CBMS 2001)*, pages 109–114. IEEE, 2001.

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

[99] Sara Sprenkle, Camille Cobb, and Lori Pollock. Leveraging user-privilege classification to customize usage-based statistical models of web applications. In *International Conference on Software Testing, Verification and Validation (ICST 2012)*. IEEE, April 2012.

[100] Internet World Stats. Internet usage statistics: World internet users and population stats. [Online] http://www.internetworldstats.com/stats.htm, last access February 2017.

[101] A. Tappenden, P. Beatty, J. Miller, A. Geras, and M. Smith. Agile security testing of web-based systems via HTTPUnit. In *Proceedings of the Agile Development Conference (ADC 2005)*, pages 24–29, Denver CO, July 2005.

[102] Paolo Tonella and Filippo Ricca. Statistical testing of web applications. *Journal of Software Maintenance and Evolution*, 16(1-2):103–127, January 2004.

[103] M.A.S. Turine, M.C.F. de Oliveira, and P.C. Masiero. A navigation-oriented hypertext model basd on statecharts. In *Proceedings of the 8th ACM Conference on Hypertext*, pages 102–111, 1997.

[104] Roland Untch, Jeff Offutt, and Mary Jean Harrold. Mutation analysis using program schemata. In *Proceedings of the 1993 International Symposium on Software Testing, and Analysis*, pages 139–148, Cambridge MA, June 1993.

[105] Roland H. Untch. On reduced neighborhood mutation analysis using a single mutagenic operator. In *ACM Southeast Regional Conference*, pages 19–21, Clemson SC, 2009.

[106] T. R. Weiss. Two-hour outage sidelines amazon.com, August 2006. [Online] http://www.computerworld.com/, last access February 2017.

[107] W. Eric Wong and Aditya P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, December 1995.

[108] Weichen Eric Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, West Lafayette, IN, 1993. [Online] http://docs.lib.purdue.edu/dissertations/AAI9420921/, last access February 2017.

[109] Wuzhi Xu, J. Offutt, and J. Luo. Testing web services by XML perturbation. In *16th IEEE International Symposium on Software Reliability Engineering*, pages 10 pp.–266, Nov 2005.

[110] E. Yourdon. *Byte Wars: The Impact of September 11 on Information Technology*. Prentice Hall, 2002.

Biography

Upsorn Praphamontripong is a Ph.D candidate of the Department of Computer Science of Volgenau School of Engineering at George Mason University. She is currently a full-time lecturer of the Computer Science Department at the University of Virginia. Praphamontripong received her M.S. in Computer Science from Central Michigan University in 2004 and her B.S. in Computer Science from Thammasat University in Thailand in 1997. At George Mason University, she involved in the Self-Paced Learning Increases Retention and Capacity (SPARC) project, which focuses on increasing the capacity and retention of overall students as well as expanding enrollment by women. Her advisor is Dr. Jeff Offutt. Praphamontripong's research interests include software engineering, software reliability engineering, software testing and maintenance, software usability, and performance analysis. She is also interested in seeking ways to increase capacity and retention in introductory programming courses.