

Leveraging User Session Data to Support Web Application Testing*

Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, Marc Fisher II

Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, Nebraska
{elbaum, grother, skarre, mfisher}@cse.unl.edu

January 17, 2005

Abstract

Web applications are vital components of the global information infrastructure, and it is important to ensure their dependability. Many techniques and tools for validating web applications have been created, but few of these have addressed the need to test web application functionality, and none have attempted to leverage data gathered in the operation of web applications to assist with testing. In this paper we present several techniques for using user session data gathered as users operate web applications to help test those applications from a functional standpoint. We report results of an experiment comparing these new techniques to existing white-box techniques for creating test cases for web applications, assessing both the adequacy of the generated test cases and their ability to detect faults on a point-of-sale web application. Our results show that user session data can be used to produce test suites more effective overall than those produced by the white-box techniques considered; however, the faults detected by the two classes of techniques differ, suggesting that the techniques are complementary.

Keywords: software testing, test data generation, web applications, empirical studies.

*Portions of this paper have been previously presented in [31].

1 Introduction

Web applications are one of the fastest growing classes of software systems in use today. These applications support a wide range of activities including business functions such as product sale and distribution, scientific activities such as information sharing and proposal review, and medical activities such as expert-system based diagnoses. It is important that web applications be dependable, but recent reports indicate that in practice they often are not. For example, one study of web application integrity found that 29 of 40 leading e-commerce sites [25] and 28 of 41 government sites [24] exhibited some type of failure when exercised by a “first-time user”.¹

Several tools for validating web applications have been created, but most of these focus on protocol conformance, load testing, link checking, and various static analyses (we discuss these further in Section 2). Such tools address problems of availability, navigability, and performance faced initially by deployed web-applications; however, they do not directly assist in detecting the failures in meeting functional requirements that have been found to dominate in mature deployed web applications [24]. To date, tools that do support functional validation do so only by supporting capture-replay: the recording of tester input sequences for use in regression testing.

Recently, a few more formal approaches for testing the functional requirements of web applications have been proposed [7, 19, 30]. In essence, these are “white-box” testing approaches, building system models from inspection of code and identifying test requirements from those models. Early studies have shown that these approaches can facilitate the construction of “adequate” (by some criteria) test suites; however, the approaches can also be costly, due to the human effort required to generate test cases that meet the identified test requirements.

The search for a generalizable and practical approach to the functional testing of web applications is complicated by several characteristics of those applications. First, web application usage can change rapidly. For example, a web site can be caught by a search engine and suddenly re-

¹For further information, see www.keynote.com/solutions/performance_indices/business_index/business_40.html and www.keynote.com/solutions/performance_indices/government_index/government_40.html.

ceive hundreds of thousands of hits per day rather than just dozens [21]. In such cases, test suites designed with particular usage profiles in mind may be inappropriate. Second, web applications typically undergo maintenance at a faster rate than other systems; this maintenance often consists of small incremental changes [15]. To accommodate such changes cost-effectively, testing approaches should be automatable and test suites should be adaptable. Finally, web applications typically involve complex, multi-tiered, heterogeneous architectures including web, application, and database servers, and clients acting as interpreters. Testing approaches must be able to handle the various diverse components in these architectures. The foregoing characteristics are not unique to web applications, but they are particularly prevalent, and their effects on testing are particularly acute, in this paradigm. Unfortunately, although the recently proposed techniques [7, 19, 30] partially address the third characteristic, the first two characteristics have not yet been addressed.

In this article we propose a testing approach that addresses these issues by using data captured during user sessions to create test cases, potentially reducing the effort involved when test engineers are required to generate test cases. We describe three stand-alone variants of this approach, and two hybrid variants that combine the approach with white-box functional testing techniques. We report results of an empirical study comparing our techniques with two implementations of the testing approach proposed by Ricca and Tonella in [30]. Unlike previous studies of web application testing techniques, however, our study assesses the fault detection effectiveness of the approaches. Our results show that user session data can be used to produce test suites more effective overall than those produced by the white-box techniques considered; however, the faults detected by the two classes of techniques differ, suggesting that the techniques are complementary.

In the next section of this article we describe the characteristics of the class of web applications that we are considering, and review related work. Section 3 describes Ricca and Tonella’s technique, and presents our new approach and its variants. Section 4 presents the design and results of our empirical study. Section 5 discusses additional issues relevant to the use of user-session data in testing web applications, and Section 6 summarizes and discusses future work.

2 Background and Related Work

2.1 Web Applications

A *web application* can be differentiated from a *web site* based on the “ability of a user to affect the state of the business logic on the server” [6]. In other words, requests made to a web application go beyond navigational requests, including some form of data that needs further decomposition and analysis to be served. Figure 1 shows how a simple web application operates. A user (client) sends a request through a web browser. The web server responds by delivering content to the client. This content generally takes the form of some markup language (e.g., HTML) that is later interpreted by the browser to render a web page at the user site. For example, if a request consists of just a URL (Uniform Resource Locator – a web site address), the server may just fetch a static web page.

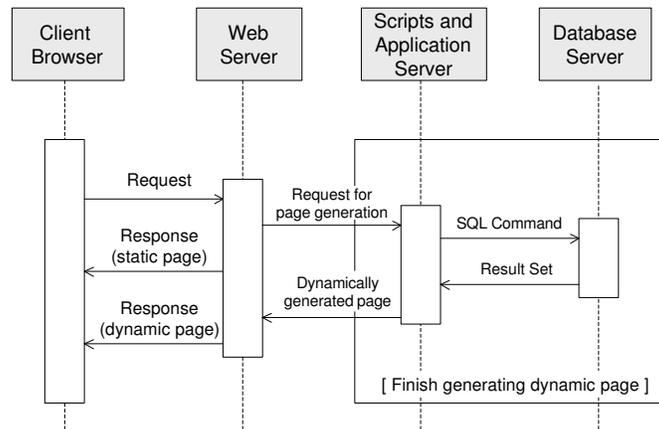


Figure 1: Sequence diagram of a web application.

Other requests are more complicated and require additional infrastructure which leads to more complex classes of web applications [34]. For example, in an e-commerce site, a request might include both a URL and data provided by the user. Users provide data primarily through forms consisting of input fields (textboxes, checkboxes, selection lists) rendered in a web page. This information is translated into a set of name-value pairs (input fields’ names and their values) and becomes part of the request. Although the web server receives this request, further elements are

needed to process it. First, a group of scripts and perhaps an application server may parse the request, query a database server to retrieve required information, and then employ formatting scripts to generate HTML code addressing the request. This newly generated code, created at run time based on a user's input, is called a dynamic web page.

In this context, the application server, database, and scripts collaborate to assemble a response that fits the request. Although requests can be more complex in practice, this example shows that there are multiple and varied technical components behind the web server. It is also important to note that scripts such as those just referred to are changed frequently [22], and the technologies supporting them change often, as evident in the frequent appearance of new standards for web protocols for data exchange and processing (e.g., XML, XSL, SOAP and CCS [18]).

2.2 Related Work on Validating Web Applications

The testing of web applications has been led by industry, where work has been oriented toward validation of non-functional requirements. Techniques proposed to date range from markup text language validators and link checkers to various load testing and performance measurement tools.²

The variety and quantity of tools for testing functional requirements of web applications is much more limited [7]. The most common class of functional testing tools provide infrastructure to support the capture and replay of specific usage scenarios [10, 14, 28]. Test engineers execute such usage scenarios, and the tools record events and translate them into scripts that can be replayed later for functional and regression testing. Another class of functional testing tools generates test cases by combining web site path exploration algorithms with tester-provided inputs [22, 26]. A prototype framework integrating these features is presented in [35]. Also, Lee and Offut present an approach for testing the data exchange process in web applications using XML [17].

Recently, three more formal techniques have been proposed to facilitate testing of functional requirements in web applications. These techniques employ forms of model-based testing, but can

²For a comprehensive list of tools see <http://www.softwareqatest.com/qatweb1.html>, and for a discussion of web application testing problems from an industry perspective see [12, 29].

be classified as “white-box” since they rely on information gathered from the web application code to generate the models on which they base their testing. Liu et al. [19] propose WebTestModel, which treats each web application component as an object and generates test cases based on data flow between objects. Ricca and Tonella [30] propose a model based on the Unified Modeling Language (UML) to enable web application evolution analysis and test case generation. Di Lucca et al. [7] propose a similar model that considers both unit testing of individual web pages and integration testing of collaboration between pages, and provides specific strategies for testing client and server pages. In essence, these techniques extend traditional path-based test generation and control or data flow adequacy assessment to the web application domain; the second and third also build on popular UML modelling capabilities.

The effectiveness of the foregoing techniques has been evaluated only in terms of ability to achieve coverage adequacy; in our search of the literature we find no reports on studies assessing the effectiveness of the techniques in terms of ability to reveal faults.

In [31] we presented an alternative technique for testing web applications based on user scenarios. In this article we proceed beyond that work in several ways. Where [31] considered two basic techniques and one hybrid, this article adds a more powerful basic technique and a new hybrid technique following a different approach. This article provides extended interpretation of the empirical results obtained in studying the proposed techniques, including analyses of the effectiveness of the techniques, analysis of results for individual faults, and analysis of the impact of the techniques on perceived reliability. This article also includes extended analysis of three issues of particular importance to the cost-effectiveness of web application testing techniques — the effects of application state and non-deterministic execution, and the management of an ever growing test suite — and presents empirical results obtained through studying some approaches for addressing these issues.

3 Web-Application Testing Techniques

In this section we provide details about the techniques that we investigate. Section 3.1 describes our implementations of Ricca and Tonella’s [30] white-box testing approach. Section 3.2 describes three techniques for testing functional requirements of web applications based on user session data. Section 3.3 presents two approaches that integrate the white-box and user-session approaches.

3.1 Ricca and Tonella’s Approach

Conceptually, Ricca and Tonella’s [30] approach creates a model in which nodes represent web objects (web pages, forms, frames), and edges represent relationships among those objects (include, submit, split, link). For example, Figure 2 displays a section of the model representing an application for on-line book purchasing, following the representation used in [30]. The model has one entry node (BookDetail) and two exit nodes (BookDetail and ShoppingCart). The BookDetail node is dynamically generated in response to a request to browse a particular book. When rendered by the browser, this page contains information about the book and (through edges e1 and e4) two forms: one to add the book to the shopping cart and one to rate the book. Both forms collect user input. If the rating form is submitted (e5), a new BookDetail page is generated with an updated rating. If a valid book quantity is submitted (e2), the shopping cart is updated and a corresponding dynamically generated page is sent to the browser. Otherwise, BookDetail is sent again (e3).

To generate test requirements and cases, a path expression to match the graph is generated following a procedure suggested by Beizer [2]. The path expression corresponding to the example in Figure 2 is $(e1e3 + e4e5) * (e1e2 + e1e3 + e4e5)$, where “*” indicates zero or more occurrences of the immediately preceding edge(s) and “+” indicates an alternative. The path expression is then used to generate test requirements by identifying the set of linearly independent paths³ that com-

³A linearly independent path is a path, through a graph, that includes at least one edge that has not been traversed previously (in a given set of paths under construction) and a set of linearly independent paths together ensures that each edge in the graph has been included in at least one test case [27].

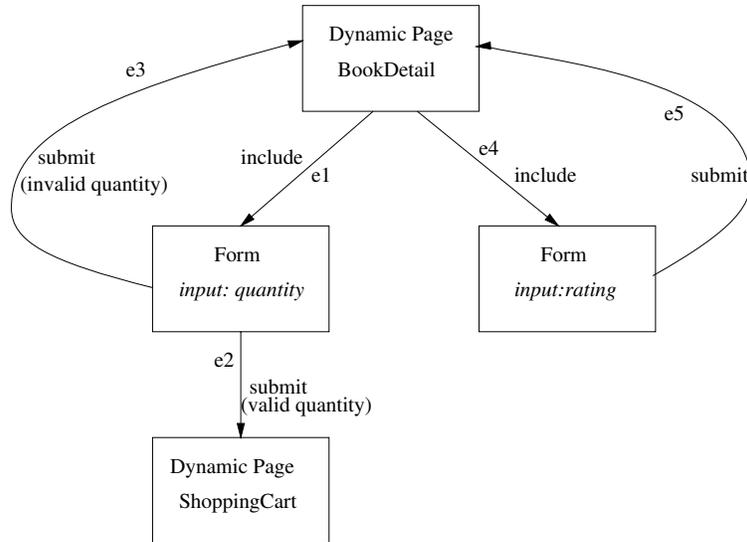


Figure 2: Simplified model for an e-commerce application.

prise it, and applying heuristics to minimize the number of requirements generated. The preceding path expression for Figure 2, for example, could yield the following test requirements: $e1e3$, $e1e2$, and $e4e5$. A test engineer then creates test cases to cover these requirements; here, test cases are sequences of web pages to be visited together with their corresponding name-value pairs. We consider two implementations of this approach.

WB-1: complete test requirements with ad-hoc selection of inputs. Our first white-box technique attempts to match the methodology presented in [30]. We generate test requirements from path expressions following the procedure just outlined, but we make the following assumptions about the process where [30] omits details: (1) we test only linearly independent paths; (2) we exercise forms that are included in multiple web pages but perform the same functionality independent of context (e.g, provide search capability) from only one source; and (3) we ignore circular links representing edges to the same page (included just to facilitate navigation within a page). After test requirements are generated from path expressions, we fill in the relevant forms so that the test cases can be executed.

WB-2: complete test requirements with formalized selection of inputs. Our second white-box implementation relaxes some of the assumptions established for WB-1 and incorporates a more elaborate approach for input value selection. In contrast to WB-1, WB-2 uses boundary values as inputs, and utilizes a strategy for combining inputs inspired by the “each condition/all conditions” strategy [3]. The test suite that results consists of a set of test cases in which for each form, each input variable is considered in isolation (all the other variables are set to the empty string), plus one test case in which all variables have values assigned. For the test cases that consider just one variable, the values are selected based on the boundary conditions for such variables. For the test case that includes all variables at once, one random combination of values is selected. The objective behind this strategy is to add formalism to the process of inputting data into the forms, as recommended in one of the examples in [30] and presented in [7].

3.2 User-session Based Techniques

One limiting factor in the use of white box web application testing techniques such as Ricca and Tonella’s is the cost of finding inputs that exercise the system as desired. Selection of such inputs is complex and must be accomplished manually [30]. User-session based techniques can help with this problem by transparently collecting user interactions (clients’ requests) in the form of URLs and name-value pairs, and then applying strategies to these to generate test cases.

Because normal web application operation consists of receiving and processing requests, and because a web application runs in just one environment which the organization performing the testing controls, the collection of client request information can be accomplished easily. For example, with minimal configuration changes, the Apache web server can log all received get-requests [1]. A slightly more powerful but less transparent alternative that can capture all name-value pairs involves adding snippets of javascript to the delivered webpages so that all requests invoke a server-side logging script. Utilizing Java servlet filters is yet another alternative that enables the dynamic interception of requests and responses at the server side. Given a particular collection mecha-

nism, user-session based techniques require little additional infrastructure to collect the required data, limiting the impact on web application performance. Another advantage of collecting just the requests is that at that higher abstraction level, some of the complexities introduced by heterogeneous web application architectures are hidden. This lessens the dependencies of user-session based techniques on changes in web application components.

Given a set of URL and name-value pairs collected from user sessions, there are many techniques by which test cases could be generated. One family of techniques re-uses user session data directly. In this context, the simplest technique involves sequentially replaying individual user sessions. A second technique involves replaying a mixture of interactions from several users. A third technique involves mixing regular user requests with requests that are likely to be problematic (e.g., navigating backward and forward while submitting a form). We consider each of these techniques in our empirical study. Other issues and approaches for web application testing, involving indirect use of user session data, incorporation of web application state in testing, non-determinism in web applications, and test suite management, are explored in Section 5.

The key questions to be addressed for user session based approaches involve whether they can in fact be cost-effective, and what cost-benefits tradeoffs exist between different approaches. To address these questions, in this work we study three specific user-session based techniques: (1) a technique that directly reuses entire sessions, (2) a technique that replays a mixture of sessions, and (3) a technique that replays sessions with some targeted modifications.

We now present these techniques. In the following, let $U = \{u_1, u_2, \dots, u_m\}$ be a set of user sessions, with u_i consisting of n requests r_1, r_2, \dots, r_n , where each r_i consists of *url*[*name* – *value*]*. For simplicity, we define a user session as beginning when a request from a new IP address reaches the server and ending when the user leaves the web site or the session times out.⁴

⁴Web applications such as the one employed in our study utilize more advanced mechanisms to track user sessions, such as the incorporation of cookies into the flowing html.

US-1: direct reuse of user sessions. Our simplest technique, US-1, transforms each $u_i \in U$ into a test case by formatting each of its associated requests, r_1 to r_n , into an *http* request that can be sent to a web server. The resulting test suite contains m test cases, one for each user session.

In a sense, US-1 is analogous to a constrained version of a capture-replay tool (e.g, Rational Robot [28]) in which we capture just the URL and name-value pairs that occur throughout a session. In contrast to approaches that capture user events at the client site, however, which can become complicated as the number of users grow, our approach captures just the URL and name-value pairs that are the result of a sequence of the user’s events, at the server site. This lessens privacy problems caused by the more intensive instrumentation used by some capture-replay tools.

US-2: combining different user-sessions. US-2 generates new user sessions based on the pool of collected data, creating test cases that contain requests belonging to different users. US-2 is meant to expose error conditions caused when conflicting data is provided by different users. US-2 generates test cases as follows:

- select an unused session u_a from U ;
- copy requests r_1 through r_i from u_a , where i is a random number greater than 1 but smaller than n , into the test case;
- randomly select session u_b from U , where $b \neq a$, and search for any r_j in u_b with the same URL as r_i ;
- if an r_j with the same URL as r_i is not found in u_b , select another session u_b ; if there is not a viable u_b , then reuse u_a directly as a test case (as in US-1);
- if an r_j with the same URL as r_i is found in u_b , then add all the requests following r_j from u_b into the test case after r_i ;
- mark u_a “used”, and repeat the process until no more unused sessions are available in U .

US-3: reusing user sessions with form modifications. Our third technique, US-3, builds on the first technique by replaying *modified* user sessions. The modifications focus on the input forms through which the users can alter web application behavior. To maintain tester’s effort to a minimum, we favored an automatic and inexpensive mechanism to modify forms’ inputs. This mechanism performs random deletion of characters in the string values associated with the name-value pairs, generating variations on the users’ inputs that may lead to the exploration of new scenarios (e.g, changing one character in a login or password sequence leads to the incorrect login scenario). The test cases are generated as follow:

- select an unused session u_a from U ;
- randomly select an unused request r_i from u_a ; if there are no more unused r_i in u_a , then reuse u_a directly as a test case (as in US-1);
- if r_i does not contain at least one name-value pair, mark r_i as used and repeat previous step;
- if r_i has one or more name-value pairs, then modify the name-value pairs:
 - create one test case for each name-value pair by deleting a random character in the value string,
 - create one test case by modifying the values of all the pairs at once by deleting a random character in each value string;
- mark u_a “used” and repeat the process until no more unused sessions are available in U .

3.3 Integrating User Session and White Box Techniques

The foregoing techniques are strictly user-session-based. It seems possible that hybrid techniques that combine user-session-based approaches with structured white-box web application testing techniques might be cost-effective, by potentially reducing the human effort needed to select inputs to cover white-box requirements while also incorporating representative user behavior. We

thus also consider two hybrid techniques. In the following, let $Q = \{q_1, q_2, \dots, q_o\}$ be a set of testing requirements identified by the technique WB-2.

HYB-1: partially satisfying testing requirements with user session data. The HYB-1 approach attempts to match equivalent user-session sequences with the testing requirements in Q . Although there is no guarantee that a set of collected inputs will cover all testing requirements, HYB-1 attempts to reduce the tester's input generation efforts by using the collected user inputs to cover as many testing requirements as possible, as follows:

- select an unused testing requirement q_i from Q ;
- translate q_i into a URL sequence $urlSeq_{q_i}$;
- identify sessions u_{match} in U that contain $urlSeq_{q_i}$. If no sessions are identified, mark q_i used and return to step 1;
- from each u_{match} , extract the name-value pairs for sequences of requests matching $urlSeq_{q_i}$;
- complete $urlSeq_{q_i}$ with collected sequences of name-value pairs to transform q_i into a set of executable test cases;
- mark q_i used and repeat the process until no more unused requirements are available in Q .

HYB-2: satisfying testing requirements with user session data and tester input. HYB-2 enhances HYB-1 by complementing user session data with tester input suggestions geared toward achieving coverage of all testing requirements. The first two steps of the process, selection and translation, are identical to the first two steps used by HYB-1. In the third step, however, each testing requirement q_i for which no relevant user session data is found is placed on a list of unsatisfied requirements. After the rest of the test case generation process is performed, unsatisfied requirements are provided to the tester for completion.

4 Empirical Study

To investigate user-session-based and hybrid techniques we performed a controlled experiment in which we applied the techniques to a web application, together with the white-box techniques, with the aim of answering the following research questions:

RQ1. How effective are the techniques? This question concerns the performance of the WB, US, and HYB techniques, in terms of the coverage and fault-detection they provide.

RQ2. Does technique appropriateness vary with fault type? This question concerns the degree of similarity between the techniques in terms of the fault detection capabilities they provide.

RQ3. What relationship exists between the number of user sessions and the effectiveness of the test suites generated based on those sessions' interactions? This question concerns the possibility of manipulating the cost-benefits of user-session based test suites through session selection.

4.1 Variables and Metrics

Our independent variable is testing technique; Table 1 summarizes the techniques considered.

We measured two dependent variables:

- Coverage: percentage of functions (subroutines) and basic blocks (single entry, single-exit sequence of instructions) covered in the code responsible for generating the web pages and accessing the database.
- Fault detection effectiveness: percentage of faults detected by the testing technique. (Section 4.2.2 provides further details on the faults utilized in the experiment).

Label	Description	Type
WB-1	Simplest Ricca and Tonella implementation [30]	White box
WB-2	WB-1 with boundary values	White box
US-1	Direct reuse of user sessions	User-session
US-2	Combining different user sessions	User-session
US-3	Reusing user sessions with form modifications	User-session
HYB-1	Partially satisfying testing requirements with user session data	Hybrid
HYB-2	Satisfying testing requirements with user session data and tester input	Hybrid

Table 1: Web application testing techniques.

4.2 Experiment Setting

4.2.1 E-commerce site

As an experiment setting we used the free and open-source online bookstore available at gotocode.com. The online bookstore’s functionalities can be divided into two groups: customer activities and administration activities. In this study we focus on functionalities that are accessible to the customer rather than to the administrator because the user data we collected was aimed at customer activities. Figure 3 provides a screenshot of the application. Using this web application, customers can search, browse, register, operate a shopping cart, and purchase books on-line, in a manner similar to that used in other similar popular sites on the web. Customer functionality is implemented through Perl scripts and modules to handle data and dynamic generation of HTML pages, Mysql to manage database accesses, a database structure composed of seven tables tracking books, transactions, and other data objects, and Javascript and cookies to provide identification and personalization functionality. The application code that provides the customer functionality through the generation of dynamic pages consists of 9 Perl files and 3 Perl modules that include 67 functions (called subroutines or sub in the Perl programming language) and 399 blocks. The application also includes a directory containing images (e.g., logos, book covers). An Apache Web Server hosts the application.

We populated the database with information on 100 books (e.g., title, authors, short description, category, price, rating). We adapted the look of the site so that the registration procedure was



Figure 3: Screenshot of e-commerce site used in our experiment.

expedited and logins were minimized, which made the navigation process similar to that used on commercial sites. Finally, to capture the data required by user-session techniques, we modified the Perl scripts that generated web pages and we added a server-side request-data logging daemon. The script modification consisted in the incorporation of Javascript event handlers (e.g., onClick, onLoad, onUnload) in the web page source. When these handlers were triggered by user request activity at the client-side, we proceeded to record the request data (e.g., time stamp, IP, userId, sessionId, http request) through the logging daemon. (Section 4.3 discusses additional instrumentation required for experimentation and assessment).

4.2.2 Fault seeding

We wished to evaluate the performance of web testing techniques with respect to the detection of faults. Faults were not available with our subject application; thus, to obtain them, we followed a fault seeding procedure similar to one defined and employed in previous studies of testing techniques [9, 13, 39]. We recruited two graduate students of computer science, each with at least two years of programming experience, and instructed them to insert faults that were as realistic as possible based on their experience. To direct their efforts we provided a tool that randomly selected an approximate location in which to seed a fault. We also provided the following list of fault types to consider (adapted from the fault classification in [23]):

- Scripting faults. This includes faults associated with variables, such as definitions, deletions, or changes in values, and faults associated with control flow, such as addition of new blocks, redefinitions of execution conditions, removal of blocks, changes in execution order, and addition or removal of function calls.
- Forms faults. This includes addition, deletion, or modification of a forms' name or predefined values for a name. In our target site, such faults were seeded in the sections of the scripts that dynamically generated the html code.
- Database query faults. This consists of the modification of a query expression, which could affect type of operation, table to access, fields within a table, or search key or record values.

The graduate students assigned to the task seeded a total of 50 potential faults. Nine of these potential faults were discarded: four were in unused sections of code and five had no impact on the behavior of the application (e.g., the value of a variable was changed after its last usage). Thus, 41 faults were retained for use in experimentation.

4.2.3 Test suite creation

We used the techniques described in Section 3 to create test suites. One of the authors led the development of test cases for the WB techniques, spending approximately 75 hours creating and refining the representation model of the web application and identifying the inputs needed to meet the adequacy criteria. (To avoid a potential source of bias, this activity was completed prior to any examination of the particular faults that had been inserted into the application.)

User-session based and hybrid techniques require user session data; thus, to create them, we needed to obtain a pool of users and encourage them to use the e-bookstore in a manner typical of users of this type of site. Users navigate such sites to browse and perhaps purchase books if their content and price are appropriate for their needs and budget. We wished to provide the context and incentive for users to interact with our application under similar circumstances.

We assembled a list of candidate study participants containing students from the Department of Computer Science and Engineering at University of Nebraska-Lincoln. We emailed participation requirements and incentives to these candidates, and 73 chose to participate; these participants had an average age of 24, and 94 percent had on-line buying experience. We asked these participants to complete three tasks. First, they completed an on-line form providing demographic data. Second, they selected two computer science courses, based on descriptions of four courses that we provided. Finally, they used the e-bookstore site to select the most appropriate book(s) for the courses they had selected. For this final step, we made the e-commerce site available to the participants for two weeks. For the first and last steps we asked the participants to employ the Microsoft Explorer browser to avoid potential compatibility problems which were not the target of this study.

To select books, participants needed to search and browse until they found those books that they considered most appropriate. We provided no definition of “appropriateness”, so that it meant different things to different participants, which we hoped would lead to a variety of activities. However, we did inform the participants of an incentive so that they would take the task completion

Metric	WB-1	WB-2	US-1	US-2	US-3	HYB-1	HYB-2
Test Suite Size	28	64	85	84	407	1004	1089
Requests	99	241	1975	1919	2742	1428	1397

Table 2: Test suite characterization.

seriously: on completion of the experiment, the five participants who selected the most appropriate books for the courses would each receive a ten dollar gift certificate. Further directions specified that, if more than five participants selected the most appropriate books, the amount spent would be evaluated, and ties would be broken by considering the time spent on the web site. Again, the objective was to recreate the conditions that motivate user behavior on similar web applications.

Data from the user sessions was logged, with 99 total sessions obtained. Fourteen of these sessions could not be used because, despite instructions, participants accessed the web site using browsers other than those required. We used the 85 remaining sessions to construct user-session based test suites of the seven varieties listed in Table 1. Table 2 characterizes the test suites thus created, along with the white-box test suites, in terms of numbers of test cases and total numbers of requests contained in the suite. The WB test suites are much smaller than US or HYB test suites, but this is as expected.

4.3 Data Collection and Technique Assessment

To enable data collection and technique assessment we developed the infrastructure depicted in Figure 4. User requests and responses are collected as well as a snapshot of the database at the end of each transaction. Once the user data has been collected, the test case generation and assessment process begins. To support the US and HYB techniques, user sessions are extracted from the collected data; then, these sessions are processed as indicated in Sections 3.2 and 3.3 to generate test suites. Note that, unlike the process for generating WB suites, the process for generating US suites does not require a test engineer’s input if such infrastructure is available, while HYB techniques do require some degree of engineer effort in the integration of the testing approaches.

Given the test suites created by the foregoing process, we executed each on the original web application with no faults seeded, saving output and database state, thus treating this version as our test oracle. The test suite execution was emulated through the automatic transformation of each test case into a series of http requests sent directly to the web application (no browser was involved in this process). To measure function and block coverage, we instrumented the application, using snippets of additional Perl code, so that each time a function or block was executed a counter associated with that function or block would be incremented. We then re-executed the suites to measure coverage. Finally, to evaluate the fault-detection effectiveness of our test suites, we activated each of the seeded faults individually and executed each test case, comparing relevant outputs to determine whether that test case revealed that fault.

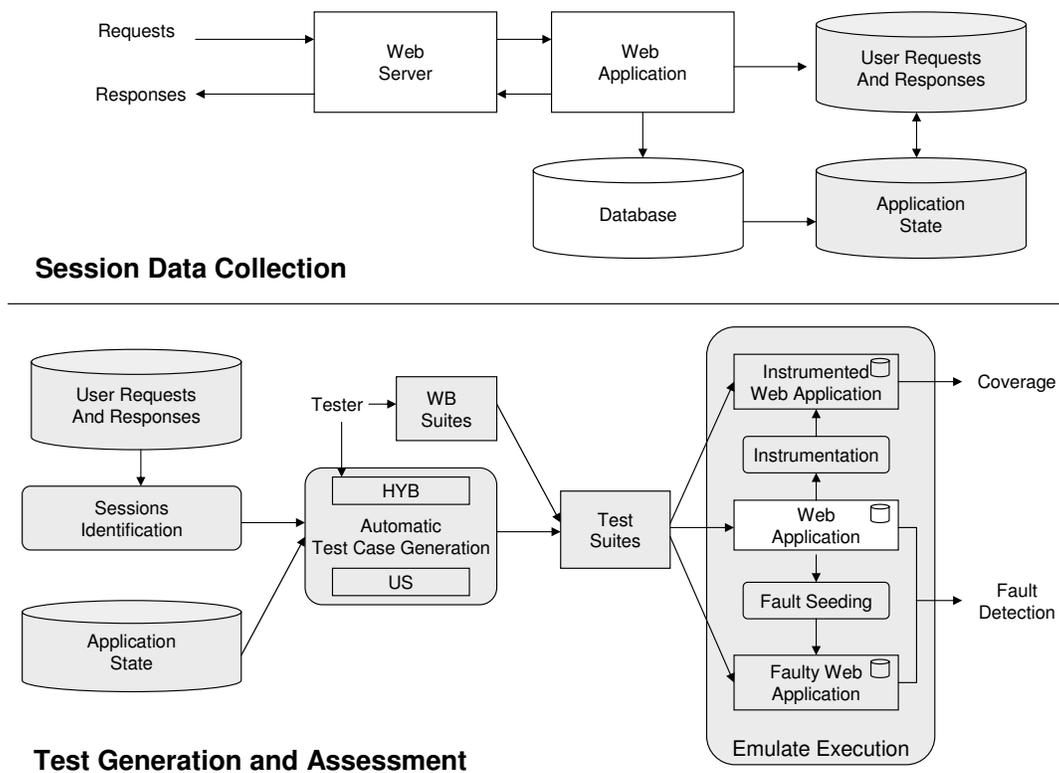


Figure 4: Experiment infrastructure for data collection.

4.4 Results

We now present the result of our study, considering each of our research questions in turn.

4.4.1 RQ1: On fault-detection effectiveness

Table 3 presents the fault-detection effectiveness results measured for the test suites generated by each technique considered. For each technique, the table presents block coverage, function coverage, and faults detected in absolute values and percentages for the test suite of that type. Overall, the test suites generated by each technique covered over 64% of the blocks and 96% of the functions in the web application, and detected between 54% and 63% of the seeded faults.

Metric	WB-1		WB-2		US-1		US-2		US-3		HYB-1		HYB-2	
	abs	%	abs	%	abs	%								
Block Coverage	263	66	306	76	263	66	255	64	288	72	260	65	270	68
Function Coverage	65	97	66	99	65	97	64	96	65	97	65	97	64	99
Faults Detected	22	54	24	58	23	56	23	56	26	63	23	56	23	56

Table 3: Summary data on fault-detection effectiveness.

WB-2 provided the greatest code coverage with 76% and 99% of the blocks and functions covered, respectively. US-3 was second best, covering 72% of the blocks and 97% of the functions. The remaining techniques achieved at most 66% and 97% coverage of blocks and functions, respectively. To our surprise, US-1 provided greater coverage than US-2, suggesting that either the procedure we used to combine sessions is inadequate, or combining sessions is not advantageous.

US-3 provided the greatest fault detection capabilities with 63% of the faults detected, followed by WB-2 with 58% of the faults detected. Further analysis revealed that even though WB-2 exercised 90% of the faulty statements (the most of all techniques), it exposed only 24 faults which indicates that either exercising those faults did not corrupt the application data state or it did not propagate to an output to become detectable. The other techniques detected fewer than 57% of the faults, and the WB-1 technique provided the least fault detection. The HYB techniques also failed to provide gains, performing similar to the basic US-1 technique.

It is worth observing that, although the difference in fault-detection effectiveness between the best US and WB techniques was only 2%, the results obtained from the US techniques were based on a small pool of user sessions, and might be expected to become more effective (subject to bounding effects) as additional sessions are collected.

4.4.2 RQ2: On technique appropriateness

To better understand the differences between techniques, we also analyzed results for differences and commonalities. Table 4 presents a detailed comparison of the most powerful white box and user-session based techniques, WB-2 and US-3. The first row of the table lists the blocks covered, functions covered, and faults detected in common by the techniques. The second and third rows present the same information focusing on blocks, functions, or faults uniquely associated with one technique and not the other. The fourth row shows the result of using both techniques.

Technique Combination	Blocks		Functions		Faults	
	abs	%	abs	%	abs	%
$(WB-2 \cap US-3)$	273	68	65	97	23	54
$(WB-2 - US-3)$	32	8	1	2	2	5
$(US-3 - WB-2)$	14	4	0	0	4	9
$(WB-2 \cup US-3)$	319	80	66	99	29	67

Table 4: Detailed comparison of WB-2 and US-3.

The table shows that the blocks covered by WB-2 and US-3 are not identical: 32 blocks were covered only by WB-2, while 14 others were covered only by US-3. Also, two of the faults found by WB-2 were not found by US-3, and four faults were found only by US-3. Similar differences occurred with the other techniques. This data supports the claim that the approaches we considered perform differently in terms of coverage and fault detection. Further, despite the unimpressive performance of the HYB techniques that we implemented, the last row of Table 4 shows that when test cases from both techniques (representing both approaches) are combined, the resulting coverage and fault detection capabilities are better than those observed for either technique singly.

Further analysis revealed that there are some faults that user-session based approaches rarely

captured. These faults involve particular name-value pairs that cannot be generated using the forms available through the web pages. For example, when supplying a book evaluation through the application, five levels (from 1 to 5 stars) are available through the web site. The only way to construct a request with evaluation values outside that range is to generate the request outside the rendered page. This scenario is not likely to originate with regular users of the application, but it could occur if the site were detected by a search robot that performs that type of request. It is likely that this type of request would need to be generated by design instead of by user activity. In addition, although we could not identify a certain type of fault that WB techniques were likely to miss, we observed that strategies such as that followed by WB-2 helped lessen the impact of tester input choices on the effectiveness of WB techniques.

Analyzing the potential exposure of a fault by a user provides another perspective on the differences between these approaches – a perspective that reflects the software’s reliability as perceived by the client. By design, user-session based techniques are more likely to detect faults that are, were, or might be exposed by users in the course of their normal operations. User-session based techniques are more representative of the type of inputs the web application will receive when deployed because they actually constitute real inputs. In contrast, WB approaches are independent of potential user behavior, generating test cases based exclusively on a coverage adequacy criterion.

To further investigate this conjecture, we ranked the faults in our application based on the number of sessions in which they were exposed (we employed the sessions from US-1 to approximate the behavior observed during the clients’ sessions). Then, we determined how many of those faults were exposed by the white box testing techniques. Table 5 shows the results. As the table shows, the faults that occurred most frequently (ranked 1-5) were exposed in over 98% of the sessions, impacting users most of the time. White box techniques exposed all five of these faults. The next few tiers in the table show, however, that the correlation between white box technique fault exposure and fault frequency was somewhat variable, and both WB techniques missed one or two faults that impacted 81% of the user sessions and two or three that impacted a third of the sessions. This

	Faults ranked in tiers				
	1-5	6-10	11-15	16-20	Rest
Average sessions affected	98% (83)	81% (69)	33% (28)	1% (1)	0% (0)
WB-1 detects	100% (5)	80% (4)	60% (3)	40% (2)	40% (16)
WB-2 detects	100% (5)	60% (3)	40% (2)	40% (2)	50% (20)

Table 5: Fault-detection effectiveness of WB techniques across fault frequency classes

variability and potential impact reflects the lack of connection between the test cases generated by white box techniques and the way in which the software was really employed by users.

4.4.3 RQ3: On numbers of user-sessions versus test suite effectiveness

A detailed analysis of our data revealed that certain techniques could have performed better given some adjustments. For example, WB-2 could have discovered one more fault if the “each condition/all condition” strategy, used to combine inputs, was replaced by a more powerful strategy such as “all variants”, which would require the execution of all potential combinations of designated values (in our case valid string and empty string) for the variables in a form [3]. Still, such an approach would imply significant additional human participation, and potentially lead to scalability problems.

US-1, on the other hand, could have detected five additional faults, increasing its fault detection effectiveness to 68% (better than any other technique) if the users had exercised specific additional functionality. Two of these additional faults were not discovered by WB-2 and required a special combination of input values to be exposed. For example, one fault could have been exposed if a user had attempted to access the shopping cart prior to completing the registration procedure. The three other faults that could have been discovered by US-1, and that were captured by WB-2, required erroneous inputs that did not appear in the collected user sessions. For example, the registration procedure required a password and a confirmation for that password. A fault was exposed when these inputs did not match. WB-2 detected that fault but US-1 did not, because no user session exhibited that behavior.

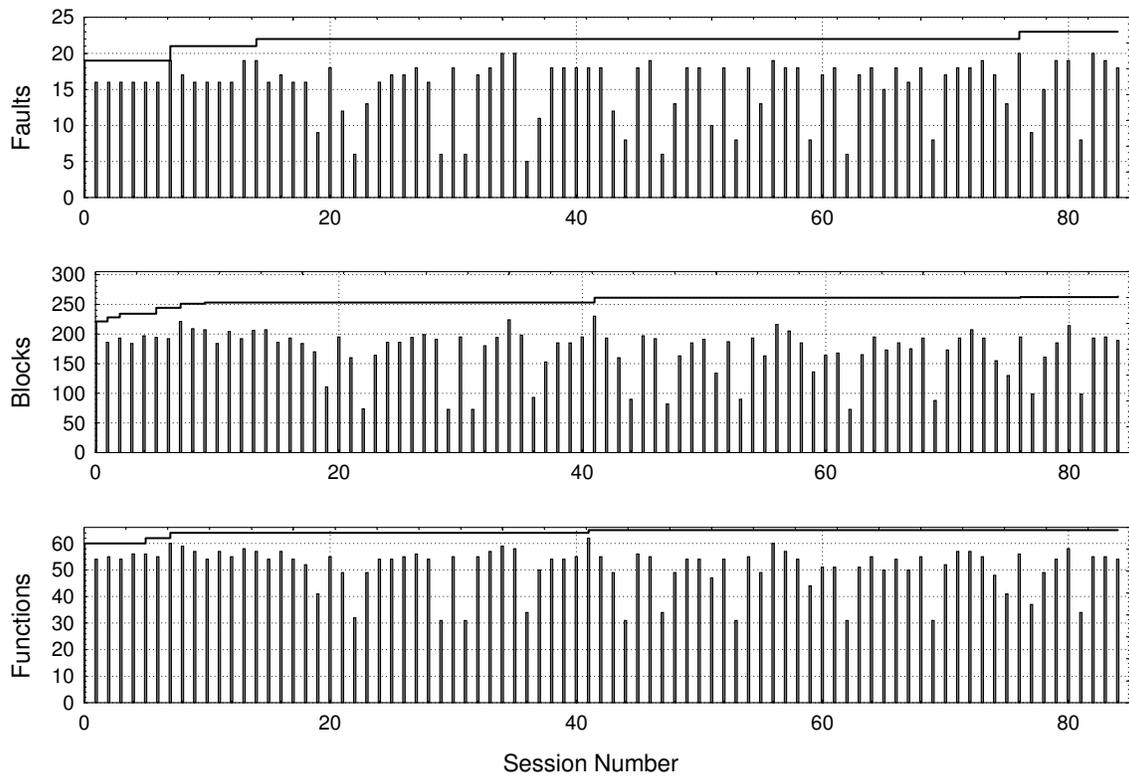


Figure 5: Relationship between number of user sessions collected and effectiveness of US-1.

In this study, the effectiveness of user-session based techniques improved as the number of collected sessions increased. Figure 5 shows the relationship between the number of user sessions collected for and employed by US-1, and its effectiveness under the various metrics considered. The x-axis presents the individual sessions from 1 to 85 (corresponding one to one with the test cases generated by US-1), the bars represent the US-1 test case value for the corresponding y-axis variable, and the line represents the cumulative count of unique faults exposed or blocks and functions covered, as additional sessions exercised the application.

The figure shows that test cases generated from user sessions varied in terms of coverage and fault exposure, and that the cumulative increases in the dependent variables were slower over time. Within the ranges observed, a positive trend in the cumulative counts suggests that user-session based techniques might continue to improve as additional sessions (and potentially additional user

behaviors) are collected; however, this trend would likely also be tempered by the costs of collecting and analyzing additional data, and as the number of detectable faults remaining in the system is reduced. Of course, with testing techniques generally, fault-detection effectiveness is expected to increase with increases in testing activity; however, where user-session based techniques are concerned, these effectiveness gains can potentially be achieved at somewhat less cost due to the reduction in tester participation required by those techniques. The costs of experimentation, however, constrained our ability to collect further sessions and data needed to investigate the relative tradeoffs; further studies on commercial sites with thousands of sessions per day would provide the opportunity to further analyze such trends.

4.5 Threats to Validity

This study, like any other, has some limitations. In this section we identify the primary limitations that could have influenced the results and we explain how we tried to control unwanted sources of variation. Some of these limitations are unavoidable consequences of the decision to use controlled experimentation; however, the advantage of control is additional certainty regarding causality.

First, we required an infrastructure with which to reproduce a web application that resembled as closely as possible those found in the real world. To control for potential threats caused by the lack of representativeness of the web application, we carefully adapted an existing e-commerce application, populating its database and configuring the site shipping and personalization attributes. In spite of this effort, the resulting web site included just a subset of the technologies used in the field, and our results can be claimed to generalize only to web applications using similar technologies. Additional web applications must be studied to overcome these threats to external validity.

Second, our user-session based techniques employ user interactions collected from users interacting with our e-commerce site. User navigation and buying patterns were not our focus, but we wished to reproduce the activities users might perform in this type of site. The instructions for participants included a task and an incentive to make the experience more realistic for a set of potential customers, but are still just an approximation of reality and threaten external validity

because of representativeness, and internal validity because they could have constituted a nuisance variable that affected the results. Similarly, the input selection that drives the WB techniques is influenced by the testers's ability and intuition for input selection. We limited the impact of this threat by providing a common strategy for all testers to use in selecting input values.

Third, we needed to seed faults in the application so that we could evaluate the testing techniques. Although naturally occurring faults are preferred, obtaining a web application with a large number of known faults was not feasible. As a consequence, we opted for a fault seeding process similar to those commonly used in previous research to measure testing techniques' fault-detection effectiveness. The risk of seeding faults that are not representative of the faults found in web applications is still a threat to external validity. For example, we did not simulate the existence of faults on the client side (e.g., javascript faults). In a related internal threat to validity, during the fault detection effectiveness measurement phase of the experiment each fault was activated individually. Although this constitutes the simplest of fault-detection scenarios (e.g., multiple faults present simultaneously can lead to more complex scenarios that include fault masking), given that this is the first study that quantifies fault detection effectiveness we felt that this limitation was acceptable.

Fourth, the metric set that we used does not consider all possible important metrics. For example, we did not consider any cost measure that would allow us analyze the cost-effectiveness of the introduced approaches; e.g., factoring in human costs associated with testing. Further studies are needed to determine under what circumstances each testing approach is appropriate from a cost-benefit perspective. Still, for the aspects of effectiveness that we intended to capture, we did employ reasonable constructs that have been utilized before in several similar empirical contexts.

Finally, we modified existing instrumentation tools and developed others to capture user interactions and to transform those interactions into test cases. For the WB techniques, we made some assumptions when the specification of Ricca and Tonella's approach was unclear. The implementation and assumptions we made are a threat to internal validity. We reduced that threat by inspecting the tools and performing multiple runs when possible.

5 Additional Considerations for Web Application Testing

5.1 Web Application State

When a specific user request is made to a web application, the outcome of that request may depend on factors not completely captured in the URL and name-value pairs alone; for example, an airline reservation request may function differently at different times depending on the pool of available seats. Further, the ability to execute subsequent test cases may depend on the system state achieved by preceding test cases. If the initial system state is known and can be instantiated, the simple approach of replaying user sessions in their entirety is not affected by application state. The use of more complex approaches such as intermixed or parallel replay, however, might often be affected by state. In such cases, one approach for employing user-session data is to periodically take snapshots of the state values (or a subset of those values) that potentially affect web application response. (This approach is similar to an approach suggested for testing database applications [5], an area of research that has close ties to the issues involved in testing web application state). Associating such snapshots with specific requests, or sequences of requests, increases the likelihood of being able to reproduce portions of user sessions, at the cost of resources and infrastructure.

A second alternative is to ignore state when generating test cases. The resulting test cases may not precisely reproduce the user activity on which they are based, but they may still usefully distribute testing effort relative to one aspect of the users' operational profile (the aspect captured by the operation) in a manner not achieved by white-box testing.

From this second perspective, the process of using user session data to generate test cases is related to the notion of partitioning the input domain of an application under test in the hopes of being able to effectively sample from the resulting partitions [38]. In this context, the potential usefulness of user-session based testing techniques, like the potential usefulness of white-box testing techniques, need not rest solely on being able to exactly reproduce a particular user session. Rather, that usefulness resides in the fact that user session data can be used to provide effective partitioning heuristics, together with input data that can be transformed into test cases related to

the resulting partitions.

To consider the possible effects of application state on the empirical results reported in Section 4, we repeated the experiments described in that section, using versions of each of our US and HYB techniques that save and utilize state. In these versions of our techniques, we saved the initial and terminal program states captured in snapshots of the application’s database; we then executed the techniques from appropriate initial states, and compared both application output and final states to detect whether fault detection had occurred.

The results of this effort revealed no difference in fault detection between techniques that considered state information and those that did not. We found this surprising, because we can easily manufacture user session scenarios under which a state characterization *would* allow detection of faulty program behavior in our web application. For example, had there been a fault in updating the book inventory after a sale, using a database snapshot that captures database state after a purchase allows detection of that fault, where just observing the response to the user request would not show any difference. Clearly, the existence of a fault that affects web application state is a necessary condition for state information to add value to our user-session based techniques. It is also important to note that, even in the presence of such a fault, it would be necessary for the collected sessions to include the scenario in which the user exercised the web application in such a way that the fault would be exposed.

5.2 Non-determinism in Web Applications

In a non-deterministic software system, identical sets of inputs can generate different outputs, exercising different sequences of events or code components. Such uncertainty can make the testing process more difficult and expensive.

In web applications we find at least two sources of non-determinism. First, multiple users can navigate through sequences of pages, arbitrarily interleaving their requests. The order in which these requests are made can affect the web application behavior. For example, if multiple users

attempt to place a reservation for a hotel room, then the order in which those reservations are placed could determine which user gets which room. Second, separate processes or threads are often spawned to handle client requests. These processes often access shared resources generating further non-determinism. For example, in the environment supported by ColdFusion [20], certain types of processes can share machine, application, client, and even session data.

A wide variety of approaches for testing traditional, non-deterministic software systems have been proposed (e.g., [4, 16, 32, 33, 36, 40]). A fundamental notion behind this work is that two overall classes of testing approaches are possible: those that sample over non-deterministic runs, and those that attempt to create specific deterministic runs. In the terminology of [4], where the goal is to exercise synchronization events (SYN-sequences), the first class of solutions are called “multiple execution” or “non-deterministic” and involve executing the program repeatedly over the same inputs in the hope of exercising a reasonable percentage of the possible synchronization events. A complementary strategy can also instrument the program with random waits to increase the probability that different executions will exercise different SYN-sequences. However, neither case can guarantee that even a reasonable subset of the SYN-sequences is exercised. The second class of solutions utilizes deterministic replay of a chosen set of SYN-sequences. This approach requires specific tool support and its effectiveness is dependent on the tester’s selection of SYN-sequences. Analogous approaches apply to testing focusing on other structural adequacy criteria.

Both of these overall classes of approaches could be applied to the testing of web applications. One simple user-session based test generation method that considers non-determinism for web applications could interleave requests from different user sessions; such interleaving could be performed in a multiple execution fashion, or using deterministic replay. This method would support testing for non-deterministic behavior with relatively minor extensions to the methods introduced in Section 3.2. For example, client requests for a book order could be randomly or deterministically shuffled to exercise several scenarios (e.g., interleave requests from client A before the purchase, during the credit card transaction, and before the inventory is updated due to client B transaction).

Still, these interleaved requests are sequential, failing to fully address simultaneously running processes accessing shared databases or data structures. To test multiple simultaneously executing web application processes, we need to be able to deterministically execute SYN-sequences or other coverage targets employing capture and replay tools. Investigation of such an approach is a possible avenue for future work.

5.3 Managing Evolving Test Suites

The increasing numbers of user sessions that can be gathered for web applications over time empowers user-session based testing techniques, but it also poses challenges for test suite maintenance and forces increased generation of oracle values. The test suite maintenance problem is caused by somewhat “equivalent” user sessions leading to redundant test cases, and ultimately, to test suites whose execution is not cost-effective. The oracle problem [37] appears when a tester needs to determine what the expected output is in response to a request, and to effectively compare results to expected results.

One approach that may help reduce the costs of maintaining test suites and examining testing results relative to large sets of user sessions involves using existing techniques to reduce test suite size while maintaining coverage by removing redundant test cases from the test suite. To evaluate the effectiveness of this approach, we applied it to the user session data gathered in our study.

We considered two reduction mechanisms. The first mechanism adapts the test suite reduction technique of Harrold et al. [11] and applies it off-line to the test cases generated by US-1. The basic idea behind this technique is to attempt to select (heuristically) the smallest possible subset of the test suite that maintains the coverage achieved by the entire test suite. When applied to the test cases generated with US-1 at a functional coverage level, this technique reduced test suite size by 98%, at the cost of missing three faults detected that would have been detected by the unreduced test suite (20 were detected). When applied to these test cases at a level of page-to-page transition coverage, this technique reduced test suite size by 79%, at the cost of missing two faults. Finally,

when applied to these test cases at a level of block coverage, this technique reduced test suite size by 93%, at the cost of missing only one fault.

A second mechanism for reducing test execution and auditing costs is based on cluster analysis, a method for finding groups in a population of objects. We used cluster analysis to define groups of test cases that had similar coverage patterns, using a hierarchical agglomerative approach and euclidian distance as our measure of similarity (following a procedure similar to the one employed by Dickinson et al. [8]). A reduced test suite was then generated by randomly selecting one test case from each cluster. As expected, a smaller number of clusters provided greater test reduction at a cost of fault detection effectiveness. For example, when defining just four clusters the test suite size was reduced by 98%, missing three more faults than the US-1 approach. On the other hand, when the number of clusters was the same as the number of US-1 test cases, then there was no reduction and the results are equivalent to running US-1 test cases.

These results suggest that applying reduction and clustering to user-session techniques could be helpful for handling the large number of requests that could be gathered from commercial e-commerce sites. As defined, however, the reduction and clustering techniques we considered function only on complete data sets. New techniques will need to be developed to incrementally handle the collection and processing of data as it arrives. We conjecture that certain adaptations could make the existing approaches work incrementally. For example, the clustering approach could be modified such that once the clusters are established, new sessions are considered only if they do not fit in an existing cluster. Still, this conjecture must be validated with further studies of scalability and performance.

6 Conclusion

We have presented a new approach for testing web applications and several techniques for implementing that approach. These new techniques differ from existing techniques in that they leverage captured user behavior to generate test cases. We have presented the results of a controlled experi-

ment that suggest that this approach's effectiveness may be complementary to that of more formal white box testing approaches.

The user-session based testing techniques that we have presented have several additional potential advantages over other techniques. First, because these techniques utilize user requests as a basis for generating test cases, they are less dependent on the complex and fast changing technology underlying web applications, which is one of the major limitations of white box approaches designed to work with a subset of the available protocols. Second, the level of human effort involved in capturing URL and name-value pairs is relatively small as these are already processed by web applications. This is not the case with testing approaches that require a high degree of participation by test engineers. Third, in user-session based approaches, each user is a potential provider of test data: this implies the potential for an economy of scale in which additional users provide additional inputs for use in test generation. The potential power of the techniques resides in the number and representativeness of the URL and name-value pairs collected, and the possibility of their use in generating more powerful test suites. Finally, unlike traditional capture and replay approaches, user-session based techniques automatically capture authentic user interactions for use in deriving test cases, as opposed to interactions, created by testers, that may not represent real field usage.

We do not suggest, however, that web application testing should rely *solely* on user session data; the abilities of other techniques to reveal other types of faults should also be leveraged, particularly prior to the collection of user-session data. User-session based techniques, however, can then be applied either in a system's beta testing phase to generate a baseline test suite based on interactions with friendly customers, or during subsequent maintenance to enhance a test suite that was originally generated by a more traditional method. The approach could also help test engineers monitor and improve test suite quality as the web application evolves, and as the application's usage proceeds beyond the bounds anticipated in earlier releases and earlier testing.

Our results suggest several directions for future work. First, the combination of traditional

testing techniques and user-session data seems to possess a potential that we have not been able to fully exploit. New methods might be able to successfully integrate these approaches. In addition, more complex techniques that consider other factors affecting the approach, such as the influences of state and non-determinism, must be explored. We have suggested approaches that can be taken to accommodate these factors, but additional study is needed.

Second, the analysis of user-session techniques suggests that using a large number of captured user sessions involves tradeoffs. Additional sessions may provide additional fault detection power, but a larger number of sessions also implies more test preparation and execution time. Techniques for filtering sessions, such as the reduction and clustering techniques we have suggested and obtained initial data on, will need to be further investigated.

Third, the applicability of the user-session test generation approach will certainly be affected by the efficiency of the data collection process. We need to be able to characterize the costs of US techniques and compare these with the costs of more traditional techniques. Further studies are also needed to determine under what types of loads user-session based approaches may be cost-effective. Given the observed asymptotic improvement in fault detection as the number of user sessions increase, studies will need to consider whether the approach should be applied to all or just a subset of the user-sessions.

Finally, we believe that there are many applications for user-session data that have not been fully explored in the domain of web applications. For example, a recent report by the Business Internet Group of San Francisco [25] states:

“The fact is that Web applications can never truly be tested to accommodate the scope of operational variables and user behaviors that a dynamic Web application encounters in production.”

This statement is likely correct, but our results also suggest that user-session data could be used to at least partly address this problem, allowing engineers to assess the appropriateness of an existing

test suite in the face of shifting operational profiles, and prioritize their validation activities based on the behavior exhibited by deployed applications. Through such approaches, we hope to be able to harness the power of user session data to improve the reliability of this important class of software applications.

Acknowledgments

This work was supported in part by the NSF Information Technology Research program under Awards CCR-0080898 and CCF-0347518 to University of Nebraska, Lincoln and CCR-0080900 to Oregon State University. The e-commerce application was obtained from gotocode.com. M. Hardojo and K. Le assisted in the fault seeding process. We especially thank the users who participated in the study, and the reviewers of the paper for their helpful comments.

References

- [1] Apache-Organization. Apache http server version 2.0 documentation. <http://httpd.apache.org/docs-2.0/>.
- [2] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.
- [3] R. Binder. *Testing Object-Oriented Systems*. Addison Wesley, Reading, MA, 2000.
- [4] R. Carver and K. C. Tai. Deterministic execution testing of concurrent Ada programs. In *Proceedings of the Conference on Tri-Ada*, pages 528–544, January 1989.
- [5] D. Chays, S. Dan, P. G. Frankl, F. Vokolos, and E. J. Weyuker. A framework for testing database applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 147–157, August 2000.
- [6] J. Conallen. *Building Web Applications with UML*. Addison-Wesley Publishing Company, Reading, MA, 2000.
- [7] G. Di Lucca, A. Fasolino, F. Faralli, and U. Carlini. Testing web applications. In *Proceedings of the International Conference on Software Maintenance*, pages 310–319, November 2002.

- [8] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the International Conference on Software Engineering*, pages 339–348, May 2001.
- [9] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.
- [10] Empirix. Web Testing Solutions. <http://www.empirix.com/Empirix/Web+Test+Monitoring/-Testing+Solutions/>.
- [11] M. Harrold, R. Gupta, and M. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.
- [12] E. Heatt and R. Mee. Going faster: Testing the web application. *IEEE Software*, pages 60–65, March 2002.
- [13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering*, pages 191–200, May 1994.
- [14] Software Research Inc. eValid. <http://www.soft.com/eValid/>.
- [15] E. Kirda, M. Jazayeri, C. Kerer, and M. Schranz. Experiences in Engineering Flexible Web Services. *IEEE MultiMedia*, 8(1):58–65, January 2001.
- [16] P. V. Koppol and K. C. Tai. An incremental approach to structural testing of concurrent software. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 14–23, January 1996.
- [17] S. Lee and J. Offutt. Generating test cases for XML-based web component interactions using mutation analysis. In *Proceedings of the 12th IEEE International Symposium on Software Reliability Engineering*, pages 200–209, November 2001.
- [18] T. Lee. World wide web consortium. <http://www.w3.org/>.

- [19] C. Liu, D. Kung, P. Hsia, and C. Hsu. Structural testing of web applications. In *Proceedings of the 11th IEEE International Symposium on Software Reliability Engineering*, pages 84–96, October 2000.
- [20] Developing ColdFusion MX Applications. Macromedia, Inc., San Francisco, CA, 2003, <http://www.macromedia.com/>.
- [21] S. Manley and M. Seltzer. Web facts and fantasy. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterrey, CA, 1997.
- [22] B. Michael, F. Juliana, and G Patrice. Veriweb:automatically testing dynamic web sites. In *Proceedings of 11th International WWW Conference*, Honolulu, HI, May 2002.
- [23] A. Nikora and J. Munson. Software evolution and the fault process. In *Proceedings of the Twenty Third Annual Software Engineering Workshop*, 1998.
- [24] Business Internet Group of San Francisco. The BIG-SF Report on Government Web Application Integrity. http://www.tealeaf.com/downloads/news/analyst_report/BIG-SF_Report_Gov_2003-05.pdf.
- [25] Business Internet Group of San Francisco. The Black Friday Report on Web Application Integrity. http://www.tealeaf.com/downloads/news/analyst_report/BIG-SF_BlackFridayReport.pdf.
- [26] Parasoft. WebKing. <http://www.parasoft.com/jsp/products>.
- [27] R. S. Pressman. *Software Engineering A Practitioner's Approach*. McGraw-Hill, 5 edition, 2001.
- [28] Rational-Corporation. Rational testing robot. <http://www.rational.com/products/robot/>.
- [29] Testing a website: Best practices. <http://www.reveregroup.com>.
- [30] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the International Conference on Software Engineering*, pages 25–34, May 2001.

- [31] S. Karre S. Elbaum and G. Rothermel. Improving web application testing with user session data. In *Proceedings of the International Conference on Software Engineering*, pages 49–59, May 2003.
- [32] K. C. Tai and R. H. Carver. A specification-based methodology for testing concurrent programs. In *Proceedings of the 5th European Software Engineering Conference*, pages 154–172, September 1995.
- [33] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, 1992.
- [34] S. Tilley and H. Shihong. Evaluating the Reverse Engineering Capabilities of Web Tools for Understanding Site Content and Structure: A Case Study. In *Proceedings of the International Conference on Software Engineering*, pages 514–523, May 2001.
- [35] J. Tzay, J. Huang, F. Wang, and W.C. Chu. Constructing an Object-Oriented Architecture for Web Application Testing. *Journal of Information Science and Engineering*, 18(1):59–84, January 2002.
- [36] S. Weiss. A formal framework for studying concurrent program testing. In *Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification*, pages 106–113, July 1988.
- [37] E. J. Weyuker. On testing non-testable programs. *The Computing Journal*, 15(4):465–470, 1982.
- [38] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.
- [39] W. Wong, J. Horgan, S. London, and A. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, pages 41–50, April 1995.
- [40] R. D. Yang and C. G. Chung. Path analysis testing of concurrent programs. *Information and Software Technology*, 34(1):43–56, 1992.