Towards Automatically Creating Test Suites from Web Application Field Data

Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
{sprenkle, gibson, sampath, pollock}@cis.udel.edu

ABSTRACT

Creating effective test cases is a difficult problem, especially for web applications. To comprehensively test a web application's functionality, test cases must test complex application state dependencies and concurrent user interactions. Rather than creating test cases manually or from a static model, field data provides an inexpensive alternative to creating such sophisticated test cases. An existing approach to using field data in testing web applications is user-session-based testing. However, previous user-session-based testing approaches ignore state dependences from multi-user interactions. In this paper, we propose strategies for leveraging web application field data to automatically create test cases that test various levels of multi-user interaction and state dependencies. Results from our preliminary case study of a publicly deployed web application show that these test case creation mechanisms are a promising testing strategy for web applications.

1. INTRODUCTION

After deployment, web applications frequently undergo maintenance to fix bugs, add functionality, and improve performance. Thoroughly and efficiently testing web applications in a way that mimics user interactions is crucial to ensure existing application functionality has not been affected by maintenance changes. With the prevalent use of web applications to conduct daily business, even partial functionality loss can cost businesses millions of dollars per hour [10].

Capture/replay of field data is an approach to testing that emulates real usage. In web application testing, user requests, i.e., URLs and associated data, are captured and replayed. Aside from the primary advantage of ensuring that configuration and application code changes have not adversely affected the application's behavior, other benefits include reproducing failures caused by user input [12] and prioritizing bug fixes based on commonly accessed code [8]. For web applications, field data has the additional advantage of being cheap to obtain (compared with other application domains [12]) and very portable because the usage data is independent of the underlying implementation and server technologies.

User-session-based testing—a specific type of capture/replay—is a complementary approach to traditional testing. In user-session-based testing, a tester captures accesses during deployment to create user sessions, which are then replayed as test cases. Intuitively, a *user session* is a single user's interaction with the application. Elbaum et al. [6] first studied leveraging field data for user-session-based testing of web applications. They showed that user-session-based test cases were nearly as effective at exposing faults as those generated by Ricca and Tonella's more expensive model-based approaches [13]. In addition to comparing user-session-based and model-based test cases, Elbaum et al. [6] proposed a technique to generate additional test suites by splitting and merging user sessions. They found that the synthesized test suites were not as effective as user sessions. As a result of our recent work [15], we can explain this result because the authors manipulated user sessions without accounting for application state, which affects application behavior.

Proceedings of MASPLAS'06 Mid-Atlantic Student Workshop on Programming Languages and Systems Rutgers University, April 29, 2006.

During our studies of web application testing with various applications, we found limitations in user sessions as test cases; the limitations stem from users sharing application state. Specifically, existing user-session-based testing techniques *ignore multi-user interactions* and, thus, *lose necessary application state dependencies*. In this paper, we propose leveraging field data beyond user sessions to generate test cases, which can *expose different application behaviors*. By expanding test cases to include multi-user interactions and accounting for state dependences during replay, the test cases more closely represent actual application usage. Our more realistic test cases and replay will cover different code and expose different faults than traditional user sessions. In addition, the proposed test case creation mechanisms allow testers to tailor testing to meet their goals and localize bugs.

The main contributions of this paper for web applications are

- a presentation of the limitations of user sessions in representing actual application usage,
- three new automatic test case generation and replay strategies: partition by fixed time blocks, partition by server inactivity threshold, and augmented user sessions, and
- a case study of the test case generation strategies for a publicly deployed web application.

We describe user-session-based testing and its limitations in Section 2. In Section 3, we present strategies for automatically generating test cases from field data. Section 4 presents the methodology for our case study, and in Section 5, we describe and analyze the results of our study. Finally, in Section 6, we present our conclusions and directions for future research.

2. USER-SESSION-BASED TESTING

We first present the limitations of user-session-based testing and describe why current user-session-based testing techniques may not be sufficient for testing web applications.

Suppose that we partition a captured log into user sessions and replay those user sessions, as proposed by Elbaum et al. [6]. Each test case is a user session, where a *user session* is a collection of user requests in the form of URL and name-value pairs. To generate user sessions, we use session cookies as identifiers when the cookies are available in the log. Otherwise, we say that a user session begins when a request from a new IP address reaches the server and ends when the user leaves the web site or the session times out. In our work, we consider a 45-minute gap between two requests from a user equivalent to a session timing out.

By partitioning the log into user sessions, we have created test cases that are easy to replay because there is only one user's session state to maintain, we have isolated each user's actions from other users, and each test case represents a single user's use of the application, i.e., an application use case. Test cases are replayed sequentially, ordered by the timestamp of their first requests. If a test case exposes a fault, debugging will focus on a single user's requests—rather than the entire log—to locate the fault.

Because the user session contains the requests of a single user, user-session partitioning loses multi-user interactions that occurred during deployment. A user may affect the shared application state—and thus the behavior of other users. For example, consider the scenario where two users are simultaneously accessing a bookstore. The following sequence of events appear in the captured log:

User1:	search for Steve Martin's *Pure Drivel*
User2:	buy last copy of *Pure Drivel*
User1:	attempt to buy *Pure Drivel*
User1:	since book is on backorder and the user wants to buy a book as a gift, search for other Steve Martin books
User1:	buy *Shopgirl*

In user-session-based testing, the above log is partitioned into two test cases that represent User1 and User2. The test case derived from User1 replays first and will find *Pure Drivel* is still available but will still purchase *Shopgirl*. The test suite may not execute the special code that handles books on backorder.

Suppose instead that we replay logs exactly as captured. Our test case is the entire captured log and exercises the backorder application code. While this replay strategy may be more intuitive than replaying user sessions, it introduces some complexity into the implementation. One could replay each request as if it originated from one user, but that implementation also does not accurately reflect the captured behavior. Instead, we need to identify the requesting user for each request (similar to parsing the user sessions) and make the request on behalf of the user while maintaining session state for each user. When we replay the requests in log order, the results more closely emulate the captured behavior of the application.

Besides the direct influence of users on the behavior of others, users also affect the dynamically generated parameter data, which in turn affects replay. A common approach to maintain state across a user's requests is to pass data as a parameter in the URL. When the parameter values are dynamically generated and depend on the order users access the site, replaying the user requests out of log order—as in user-session based testing—will not represent actual usage. However, out-of-order execution may cover error code or less frequently used code.

An example of the effect on dynamically generated parameter data can be seen in order numbers from our bookstore application. Returning to our bookstore example, the application may assign an order number to each user. Instead of looking up the customer's order in the database, the order number is passed as a parameter, e.g., "orderno=XXXX", thus acting as a cache. The application can then use the passed order number to process the purchase request. Since User2 purchases a book first, her order number is 0001, and User1's order number is 0002. The order numbers are recorded as URL parameters in the log. When replaying the user sessions as test cases, User1 is assigned the order number first (0001), but the assigned value (0001) will not correspond to the order number encoded in the URL (0002). Unless we fix the user sessions before replay, which may be difficult or cumbersome with large test suites, the user-session-based test suite will exercise the error code that handles the mismatched order numbers, instead of the code handling correct order submissions. Replaying the captured log will exercise the application as during deployment in this example.[1]

As we illustrated in these two examples, replaying in log order captures an application's deployed behavior closely. However, the replay may not duplicate the deployed behavior because of differences in the timing of incoming requests, the configuration of the system (e.g., not sending email in the testing environment), etc. Furthermore, if we encounter a bug during replay, it is more difficult to identify the bug's root cause because of the log's size and the interaction between multiple users. Since deployed applications can log gigabytes of accesses in a day and the logs are likely repetitive because users access the application similarly, we need to reduce the size of the suite, thus reducing test case redundancy and improving test efficiency.

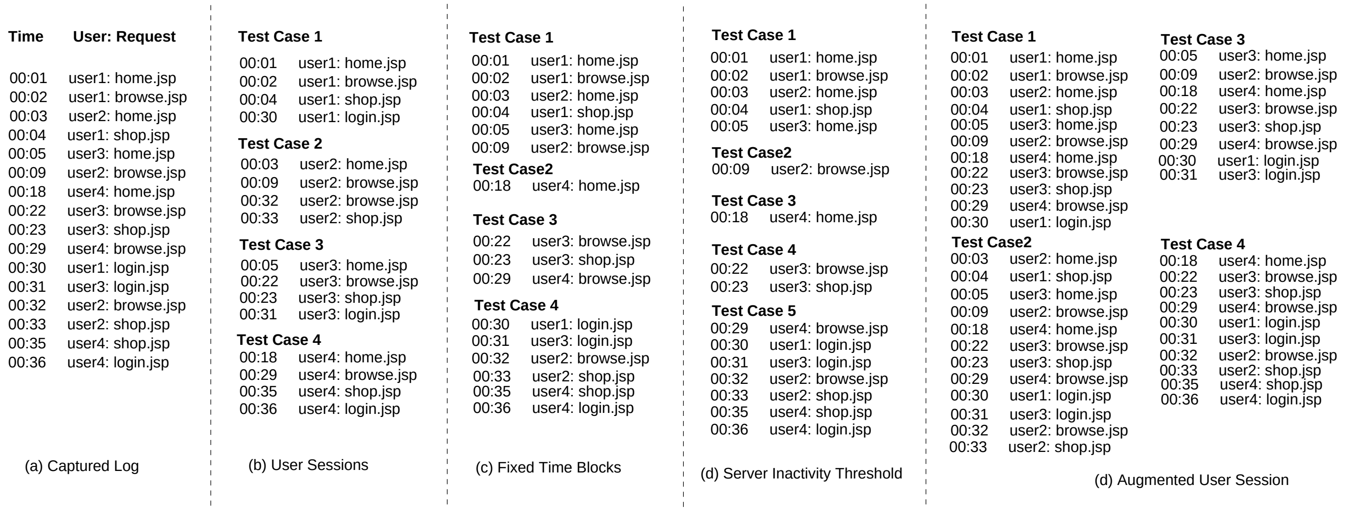
In summary, *a tester probably wants both types of test cases—test cases that expose faults with respect to one user and faults caused by interaction between users—because the test cases will exercise the application in different ways.* The primary advantages of partitioning into user sessions is that test cases represent the application's use cases and isolate the user's requests from other users' requests, which facilitates debugging. However, user-session partitioning does not emulate multi-user interactions or handle dynamically generated parameter data. Alternatively, replaying the recorded log captures multi-user interactions but makes it harder to locate bugs. In the next section, we describe our strategies for generating test cases that address these disadvantages.

3. STRATEGIES FOR GENERATING MULTI-USER TEST CASES

A test strategy is an algorithm or heuristic used to create test cases from a representation, an implementation, or a test model [2]. Techniques have been proposed to generate test cases from static models of a web application [1, 9, 4, 11, 13]. In this section, we present three different strategies to create test cases that capture *multi-user interactions from field data* for web applications. The test cases are expected to (1) represent logical user sessions so as to target functionality and usage patterns as experienced by application users, (2) be cost-effective to replay and manageable in terms of overhead involved in maintaining and executing each test case, and (3) be effective in terms of program coverage and fault detection capabilities.

Throughout this section, we will use Figure 1 to illustrate the differences between test case generation strategies. Figure 1 (a) is the captured web server log, with users *user1, user2, user3,* and *user4* accessing the application. Partitioning by user sessions, as described in the previous section, creates the test cases shown in Figure 1 (b). In the remainder of the paper, we will refer to this approach as User Sessions.

[1] Handling nondeterministic parameter values is an area of future research. One approach is to modify the application under test to generate the values deterministically and test the nondeterministic code independently.



```
Time    User: Request

00:01   user1: home.jsp
00:02   user1: browse.jsp
00:03   user2: home.jsp
00:04   user1: shop.jsp
00:05   user3: home.jsp
00:09   user2: browse.jsp
00:18   user4: home.jsp
00:22   user3: browse.jsp
00:23   user3: shop.jsp
00:29   user4: browse.jsp
00:30   user1: login.jsp
00:31   user3: login.jsp
00:32   user2: browse.jsp
00:33   user2: shop.jsp
00:35   user4: shop.jsp
00:36   user4: login.jsp
```

(a) Captured Log

```
Test Case 1
00:01   user1: home.jsp
00:02   user1: browse.jsp
00:04   user1: shop.jsp
00:30   user1: login.jsp
Test Case 2
00:03   user2: home.jsp
00:09   user2: browse.jsp
00:32   user2: browse.jsp
00:33   user2: shop.jsp
Test Case 3
00:05   user3: home.jsp
00:22   user3: browse.jsp
00:23   user3: shop.jsp
00:31   user3: login.jsp
Test Case 4
00:18   user4: home.jsp
00:29   user4: browse.jsp
00:35   user4: shop.jsp
00:36   user4: login.jsp
```

(b) User Sessions

```
Test Case 1
00:01   user1: home.jsp
00:02   user1: browse.jsp
00:03   user2: home.jsp
00:04   user1: shop.jsp
00:05   user3: home.jsp
00:09   user2: browse.jsp
Test Case2
00:18   user4: home.jsp

Test Case 3
00:22   user3: browse.jsp
00:23   user3: shop.jsp
00:29   user4: browse.jsp
Test Case 4
00:30   user1: login.jsp
00:31   user3: login.jsp
00:32   user2: browse.jsp
00:33   user2: shop.jsp
00:35   user4: shop.jsp
00:36   user4: login.jsp
```

(c) Fixed Time Blocks

```
Test Case 1
00:01   user1: home.jsp
00:02   user1: browse.jsp
00:03   user2: home.jsp
00:04   user1: shop.jsp
00:05   user3: home.jsp
Test Case2
00:09   user2: browse.jsp

Test Case 3
00:18   user4: home.jsp

Test Case 4
00:22   user3: browse.jsp
00:23   user3: shop.jsp
Test Case 5
00:29   user4: browse.jsp
00:30   user1: login.jsp
00:31   user3: login.jsp
00:32   user2: browse.jsp
00:33   user2: shop.jsp
00:35   user4: shop.jsp
00:36   user4: login.jsp
```

(d) Server Inactivity Threshold

```
Test Case 1                      Test Case 3
00:01   user1: home.jsp          00:05   user3: home.jsp
00:02   user1: browse.jsp        00:09   user2: browse.jsp
00:03   user2: home.jsp          00:18   user4: home.jsp
00:04   user1: shop.jsp          00:22   user3: browse.jsp
00:05   user3: home.jsp          00:23   user3: shop.jsp
00:09   user2: browse.jsp        00:29   user4: browse.jsp
00:18   user4: home.jsp          00:30   user1: login.jsp
00:22   user3: browse.jsp        00:31   user3: login.jsp
00:23   user3: shop.jsp
00:29   user4: browse.jsp
00:30   user1: login.jsp
Test Case2                       Test Case 4
00:03   user2: home.jsp          00:18   user4: home.jsp
00:04   user1: shop.jsp          00:22   user3: browse.jsp
00:05   user3: home.jsp          00:23   user3: shop.jsp
00:09   user2: browse.jsp        00:29   user4: browse.jsp
00:18   user4: home.jsp          00:30   user1: login.jsp
00:22   user3: browse.jsp        00:31   user3: login.jsp
00:23   user3: shop.jsp          00:32   user2: browse.jsp
00:29   user4: browse.jsp        00:33   user2: shop.jsp
00:30   user1: login.jsp         00:35   user4: shop.jsp
00:31   user3: login.jsp         00:36   user4: login.jsp
00:32   user2: browse.jsp
00:33   user2: shop.jsp
```

(d) Augmented User Session

Figure 1: Examples of Generating Test Cases from Field Data

3.1 Time-Based Approaches

We propose partitioning the web server log based on time intervals. The intuition behind creating such a test case is that the test case represents a snapshot of application activity, from the server's (rather than a single user's) perspective. The time-based approaches are highly dependent on the application characteristics and typical usage patterns of the application and are appropriate for applications whose usage patterns change depending on the time of day or year. For example, an application that manages a conference may have distinct periods of activity for submissions, camera-ready submissions, and registration, with bursts of activity during certain times of the day or as deadlines approach. Another example is a frequently used bookstore application that has long sequences of requests arriving from the same user within small time intervals. On the other hand, a course manager application where instructors post grades and students view their grades online would have shorter bursts of activity intermingled with long, inactive periods. In the course management application, small inactive periods probably suggest students viewing their grades after the instructor assigned grades; larger inactivity periods could occur between assignments.

3.1.1 Fixed-Time Blocks

We first propose simply partitioning the web server log into fixed time blocks (FIXED-TIME BLOCK). Partitioning by FIXED-TIME BLOCK addresses the disadvantages that arise from partitioning by USER SESSIONS, while creating smaller, multi-user test cases that are easier to debug than the full log. The straightforward algorithm to generate test cases is shown in Figure 2.

For the example in Figure 1 (a), we partition the test suite in time blocks of $timeint = 10$ minutes to create the four test cases shown in Figure 1 (c). All the requests in each test case are less than ten minutes apart. The last request in **Test Case 1** and the first request in **Test Case 2** are separated by more than 10 minutes, hence the two requests are assigned to separate test cases. Any interaction between *user1* and *user2* that affects the state of the system in the first 10 minutes during actual usage is captured by **Test Case 1**.

The primary disadvantage of FIXED-TIME BLOCK is the strict partitioning of the log into fixed-time-length test cases, which means that a logical user session may be split across multiple test cases. Since the chosen fixed length will determine the number and size of test cases, the length must be chosen wisely. If not selected appropriately, the fixed-time blocks are likely to partition the web server log into numerous small test cases or create large—possibly redundant—test cases containing many requests. Both of these scenarios are undesirable because they contribute to the overhead of maintaining and executing a large suite of test cases or a smaller suite of large test cases.

```
Input: log $L = (r_0, r_1, ..., r_n)$, sorted by timestamp of request $r$
Output: test cases $C = (c_0, c_1, ..., c_m)$
select a fixed time interval $timeint$
select the first request $r_0$ in the captured log $L$
add request $r_0$ to test case $c_j, j = 0$
let $t$ be the the time stamp of request $r_0$
for each request $r_i, i > 0$ in log $L$
    note the time stamp $t_i$ of request $r_i$
    if difference between $t_i$ and $t > timeint$
        increment j
        create new test case $c_j$
        let $t$ be the the time stamp of request $r_i$
    add $r_i$ to test case $c_j$
```

Figure 2: FIXED-TIME BLOCK Algorithm

3.1.2 Server Inactivity Threshold

To decrease the number of logical user sessions split across multiple test cases while still maintaining a high level of multi-user interaction, we propose partitioning based on server inactivity periods. We hypothesize that application usage patterns change after a period of inactivity. To determine a suitable server inactivity period, we apply statistical analysis on the captured log's periods of inactivity to determine a suitable server inactivity interval for the application.

Figure 3 shows the algorithms for generating test cases and for determining a reasonable threshold. Using the partitioning algorithm with a server inactivity period of 4 minutes, the captured log in Figure 1 (a) is partitioned into five test cases (Figure 1 (d)). It should be noted that a tester only needs to select an appropriate threshold once—potentially for many different applications—and therefore this approach is cheaper than continually splitting the logs

```
**Input**: log $L = (r_0, r_1, ..., r_n)$, sorted by timestamp of request $r_i$
**Output**: test cases $C = (c_0, c_1, ..., c_m)$
$To\ partition\ log\ by\ server\ inactivity$ :
    select server inactivity threshold $threshold$
    select the first request $r_0$ in the captured log $L$
    add request $r_0$ to test case $c_j, j = 0$
    let $t$ be the the time stamp of request $r_0$
    for each request $r_i$ in log $L$
        note the time stamp $t_i$ of request $r_i$
        if difference between $t_i$ and $t > threshold$
            increment j
            create new test case $c_j$
        let $t$ be the the time stamp of request $r_i$
        add $r_i$ to test case $c_j$
$Tofind\ inactivity\ threshold$ :
    partition log into user sessions as described in Section 2 to
        create set of user sessions $U$
    compute time difference, $server\_inactivity_{ij}$, between
        consecutive user sessions $u_i$ and $u_j$ in $U$
    analyze statistically all $server\_inactivity_{ij}$
        to determine a reasonable threshold
```

Figure 3: Server Inactivity Threshold Algorithm

```
**Input**: log $L = (r_0, r_1, ..., r_n)$, sorted by timestamp of request $r_i$
**Output**: test cases $C = (c_0, c_1, ..., c_m)$
create the set of user sessions $U$ from the captured log $L$,
        as defined in Section 2
for each user session $u_i$ in the set $U$
    save the timestamp of the first $f_i$ and last $l_i$ requests in $u_i$
for each request $r_i$ in the captured log $L$
    for each user session $u_j$ in the set $U$
        let $t_i$ be the timestamp of $r_i$
        if $f_j \leq t_i \leq l_j$
            add $r_i$ to $u_j$
```

Figure 4: Augmented User Sessions Algorithm

into individual user sessions as with User Sessions.

Replaying the test cases created by Server Inactivity partitioning covers distinct application usage patterns and captures multi-user interactions. Although Server Inactivity captures more logical user sessions than Fixed-Time Block, some logical user sessions will still be split across multiple test cases. In addition, because of either poor threshold choice or heavy application usage, Server Inactivitycan create large test cases with many requests per test case. The overhead of maintaining and executing such large test cases may not practical, especially in the case of heavy, continuous application usage.

3.2 Augmented User Sessions

To address the disadvantage of breaking up the logical user sessions in the other multi-user interaction approaches, we present a third approach: Augmented User Sessions. This approach has the advantages of replaying the whole captured log while providing reasonably-sized test cases to ease in debugging and testing. The algorithm for generating test cases is in Figure 4.

The captured log in Figure 1 (a) is converted into test cases by Augmented User Sessions as shown in Figure 1 (e). **Test Case 1** contains all the requests made by *user1* and any other requests made at the same time that *user1* was using the application. The advantage of using Augmented User Sessions is that the test cases capture unbroken groups of logical user sessions, while still capturing multi-user interaction that may affect the state of the system. As shown in Figure 1 (e), a request may be contained in multiple test cases and the size of the test cases might become large and difficult to manage for some applications. Generating these test cases costs more than the other techniques (including User Sessions) because we need to logically divide the user sessions before augmenting them.

3.3 Summary of Tradeoffs

In this section, we presented three alternatives to user sessions as test cases and discussed the benefits and limitations of each technique informally; a qualitative summary of the tradeoffs between the test-case generation techniques is in Table 1. Our proposed approaches address the limitation of User Sessions, namely that user sessions as test cases fail to capture multi-user interactions. However, the proposed approaches have their own benefits and drawbacks. Fixed-Time Block and Server Inactivity require the tester to choose an appropriate time length or threshold and the resulting test cases may break logical user sessions across multiple test cases, unlike Augmented User Sessions. Splitting logical user sessions will cause Fixed-Time Block and Server Inactivity to cover more error code than Augmented User Sessions; which technique covers more code in practice is the subject of our case study in the next section.

In terms of test case generation cost, the worst case time for each generation algorithm is $O(n)$, where n is the number of requests in the log. In practice, the algorithm for Augmented User Sessions is more expensive because it requires calculating user session boundaries in addition to augmenting the test cases. However, the ability of test cases to model user behavior and the expense of executing the test cases are more important considerations than the initial test case generation cost.

3.4 Implementation

The capture/replay system is part of our web application testing framework [15]. Our system for capture/replay consists of three components: logging, test-case generation, and replay. Our logging tool records user requests, including URL data and cookies. The log is the input to our suite of test-case generation tools, which implements the test case generation algorithms. We implemented customized replay tools using HTTPClient [7], which handles get and post requests, file uploading, and maintaining the client's session state. For test cases that contain requests from multiple users, the requests are tagged with time and session information so that the tool can maintain the state for each user, as described in Section 2. For more details about our framework, refer to [15].

Except for Augmented User Sessions, the User Sessions, Fixed-Time Block, and Server Inactivity algorithms are implemented exactly as presented. However, for Augmented User Sessions, a single logged request may appear in multiple test cases—effectively replaying portions of logical user sessions multiple times. However, for our coverage studies, since replaying the full test suite is equivalent to replaying the log, we have slightly modified the implementation so that the same request is not replayed multiple times; the implementation change will manifest itself in the time results.

4. CASE STUDY

To evaluate the differences between the testing strategies, we applied the strategies to the captured log of a deployed web application. Each test case generation technique is partitioning or pro-

Test Suite	Benefits	Drawbacks
User Sessions	Represent logical user sessions	No multi-user interaction
Fixed-Time Block	Multi-user interaction; variable-sized test cases	Requires smart time out; likely to split user sessions across test cases
Server Inactivity	Multi-user interaction; variable-sized test cases	Requires smart threshold; may split user sessions across test cases
Augmented User Sessions	Represent logical user sessions; Multi-user interaction	Larger test cases than User Sessions; higher cost to generate test cases

Table 1: Qualitative Comparison of Techniques

Classes	Methods	NCLOC	Statements
355	1534	61720	27136

Table 2: Subject Application Characteristics

Tot URLs	Distinct URLs	25th %	Median Gap	75th %	Avg Gap
16275	443	4 s	24 s	74s	13 mins

Table 3: Log Characteristics

cessing the original server log in different ways. We would like to compare the resultant types of test cases in terms of

1. Effectiveness – program coverage
2. Efficiency – the cost of test suite generation and replay

4.1 Subject Application

Our research group developed a customized web application for maintaining a digital publications library based on DSpace, an open-source digital repository system [5]. The application automatically generates sorted publications pages from a database that research group members maintain through a web application interface. A user can create dynamic views of publications by searching with different criteria. DSpace is written in Java Servlets and JSPs that deliver HTML content to the user and uses a PostGreSQL database and a filestore backend. We collected field data after publicizing our digital library in August 2005.

DSpace's application and log characteristics from August 2005 through February 2006 are in Tables 2 and 3, respectively. The application characteristics include both the JSP and Java code. From the log characteristics in Table 3, we see that the time gap between most consecutive requests is very short—less than a minute difference between them.

4.2 Methodology

4.2.1 Variables and Measures

Our experiment involves one independent variable: the test case generation technique. The test case generation techniques examined are User Sessions, Fixed-Time Block, Server Inactivity, and Augmented User Sessions.

To answer the previously stated research goals, we evaluated generated test cases in terms of efficiency and effectiveness. We used three dependent variables as our measures: program coverage, cost of generation, and replay cost for suite.

4.2.2 Experiment Design

We performed our experiments within the experimental framework described in our previous work [15]. We use Cenqua's Clover [3] to collect statement, condition, and branch coverage.

We implemented each test case generation technique in Java. We executed each technique on DSpace's captured access logs to generate the suite of test cases. Because the test suites generated by Fixed-Time Block and Server Inactivity depend on the chosen length of time or inactivity threshold, we generate three test suites for Fixed-Time Block, using timeouts of one minute, one hour, and six hours based on the log characteristics in Table 3 and our intuitions about what lengths of time would capture effective test cases. We chose an inactivity threshold of 25 minutes for Server Inactivity after statistically analyzing the inactivity gaps in two very different web applications, a conference registration and submission manager as well as DSpace. We found that 75% of the server inactivity periods were greater than 25 minutes. Therefore, an inactivity period of 25 minutes will merge 25% of the logical user sessions in test cases. To replay the test cases, we used our customized Java replay tools. We collected generation and replay costs as well as coverage for each suite.

4.3 Threats to Validity

One threat to validity of our experiments is our lack of large captured logs—logs that contain millions of accesses, rather than thousands. Because we conducted our case study on one application with its own unique usage characteristics, we cannot generalize our results to all web applications; however, we believe our application is complex enough and we collected a large enough log to be able to evaluate some of the differences between the techniques. In addition, the machines we used to run experiments were not dedicated to our experiments; other users, other experiments, backups, and network activity may affect the timing results.

5. RESULTS AND ANALYSIS

In this section, we present the results of our case study as well as our analysis of these results. Table 4 summarizes the results for test suite size, number of statements covered, generation time, and replay time for all generated suites.

5.1 Program Coverage Effectiveness

From Table 4, the statement coverage for most of the test suites was comparable (within 400 statements), covering around 63% of the code. We do not expect 100% coverage because using field data as test cases will only cover the code that users access. Analyzing our coverage reports, we found that most of the uncovered code was in alternative classes that our configuration did not use. The remaining uncovered code was administrative functionality and classes used to initialize the system because we started logging user sessions after this phase.

Augmented User Sessions covers the most statements. Because Augmented User Sessions replays the log similarly to the deployed execution with interacting users, the emulated users primarily maintain the appropriate state and the application behaves as expected, covering the most code. We did observe requests that did not match deployed behavior, which we believe was caused by a change in the code. Inevitably, because of DSpace's dependence on state, User Sessions will execute error code (often redundant in our study) instead of the correct behavior. We attribute the much smaller coverage of the hourly and minute test suites to splitting the

Metric	User Session	Hourly	Minute	6-hour	Server Inactivity	Augmented User Sessions (Log)
Number of Test Cases	1342	1769	8447	508	1814	1342 (1)
Statements Covered	17536	15713	12270	17674	17745	17866
Generation Time Per Suite (s)	9	11	52	5	14	16 (1)
Suite Replay Cost (mins)	76	102	216	73	75	52

Table 4: Comparison of Test Suites

Comparison	Hourly	Minute	6-hour	Server Inactivity	Augmented User Sessions
US ∪ A	17630	17638	17731	17947	17971
US ∩ A	15585	12141	17445	17300	17363
(US ∪ A) - (US ∩ A)	2045	5497	286	647	608
US - A	1934	5381	74	219	156
A - US	111	116	212	428	452

Table 5: Comparison of Statement Coverage of Alternative Test Case Generation Techniques (A) with User Sessions (US)

logical user session across test cases and thus losing the session's state. These test suites execute error code more frequently than the other test suites.

To help quantify the differences between the suites, we compare the statement coverage of User Sessions with the other suites, shown in Table 5. The first row is the combined number of statements that the suites cover, and the second row is the number of statements in common that both suites cover. The third row is a measure of how different the two suites are: the larger the total, the more different the suites. The fourth row is the number of statements User Sessions covers but the alternative suite (A) does not cover. Lastly, the fifth row is the number of statements that the alternative suite covers (A) that User Sessions (US) does not.

User Sessions executed some code that the alternative suites did not and vice versa. The best alternative techniques (Server Inactivity and Augmented User Sessions) executed over 400 statements that User Sessions missed. We attribute most of the code unique to User Sessions to statements that handle state inconsistencies caused by the User Sessions's loss of multi-user interactions. For example, when replaying User Sessions, some of the publications were not uploaded or were not approved for inclusion in the digital library because of inconsistencies between the publication identifier in the URL and the publication identifier that the executing server expected. Instead of executing the expected code, User Sessions covers error code in a servlet that handles displaying publications because the publication id is not valid.

For DSpace, replaying both User Sessions and Augmented User Sessions yields the largest number of statements executed—in only two hours.

5.2 Test Suite Generation & Replay Costs

Test suite generation consists of two costs: parsing the server log and generating suites. The time required for parsing the server log is dependent on the number of requests—a constant across all our techniques except for Augmented User Sessions. We automatically generated each test suite in less than a minute, as shown in Table 4. Creating test cases seems to dominate the cost of generating suites because the cost closely matches the number of generated test cases.

The cost of replaying the test suites was on the order of an hour. The time to replay the multi-user capturing test cases was in general higher because of the cost of maintaining the state for multiple users and replaying each request as the appropriate user. In addition, test suites with more test cases will take more time to replay due to the increased overhead of starting our Java replay tool once for every test case.

We attribute Augmented User Sessions's faster replay time to two factors: a) we preprocessed the requests into one large test case so that we did not replay the same URL multiple times and b) the suite executes less error code; depending on the error code executed, the server will return an empty response, which causes the replay tool to repeat the request.

5.3 Preliminary Analysis

Our case study revealed interesting results: even though each test suite replayed the same requests, the order in which requests were replayed and how session state was maintained affected the application's behavior. While all test suites executed the same 12K statements, each test suite covered additional unique code: User Sessions executed error code because of out-of-order replay, while Fixed-Time Block and, to a lesser extent, Server Inactivity executed error code because session state was not maintained appropriately (due to split in the logical user sessions).

To maximize DSpace coverage, test cases should maintain logical user sessions and capture multi-user interactions. In our case study, Augmented User Sessions covered the most statements. While not explicit, Server Inactivity maintained user sessions in our experiment, as a side effect of using an accurate inactivity threshold. Combining test suites from different generation strategies provides more coverage than using the test suites from each technique in isolation.

The unique code covered by the suites other than Augmented User Sessions may not be worth their replay cost because the same code is executed frequently by the multiple test cases in the suite. In the future, we are interested in studying the marginal increase in coverage by the test cases from the different test case generation techniques. We believe that Fixed-Time Block (Minute or Hourly) will create suites where latter test cases improve coverage of earlier test cases only marginally.

Choosing and replaying an appropriate reduced suite may be sufficient to address the issue of multiple test cases covering the same code. After removing the redundant test cases from each test suite, we believe that we will be left with different types of test cases. For example, as we have shown in previous work [14], reducing User Sessions results in a reduced suite that represents the set of different use cases of the application. Reducing a test suite created by Fixed-Time Block or Server Inactivity will likely select test cases from different application usage cycles. Reducing suites generated by Augmented User Sessions creates test cases that contain unique logical user sessions and multi-user interactions. In the future, we plan to reduce the suites from the different test case generation strategies and evaluate their cost-effectiveness.

We were also interested in investigating whether it is more important to capture multi-user interactions or logical user sessions in the test cases. From our results, User Sessions, which does not capture multi-user interaction, covers less code than a time-based technique such as FIXED-TIME BLOCK (6-hour) or SERVER INACTIVITY. Both FIXED-TIME BLOCK (6-hour) and SERVER INACTIVITY maintain multi-user interaction at the cost of some loss in maintaining logical user sessions, although this was not the case for the other time-based techniques. For our subject application, it appears that it is more important to capture multi-user interactions than maintain logical user sessions. However, we do not expect this to hold true for web applications that do not have state dependencies caused by multi-user interaction.

From our preliminary results, AUGMENTED USER SESSIONS best imitates the deployed application behavior. Depending on the time interval/threshold selected FIXED-TIME BLOCK and SERVER INACTIVITY are likely to imitate deployed application behavior. However, small intervals of FIXED-TIME BLOCKor inappropriately selected inactivity threshold values will split logical user sessions to such an extent that the resulting loss in session state will make it impossible to mimic deployed behavior.

5.4 Observations

Beyond the results of our case study, we also informally observed different application behaviors by replaying the alternative test suites. We inadvertently performed load testing on our application. While we had no problem replaying USER SESSIONS on the application, our server and the application could not handle the high load created by replaying the other test suites, which generated many more sessions at a high rate. Since users do not have an explicit "end" or "close" request to end their session, the server typically throws out the session after some period of inactivity. The default timeout for the Resin web server is 30 minutes. In our system, we replay a month's worth of requests in about 10 minutes. The server must have enough resources to handle all of the requests, otherwise it does not have time to clean out sessions and must start dropping requests.

Replaying the test cases with multi-user interaction also exposed a known problem in DSpace's underlying text search engine with too many open files, which has been addressed in later versions of DSpace.

We also observed that, in a few cases, replaying the SERVER INACTIVITY test suite exhibited behavior similar to actual deployed application behavior that AUGMENTED USER SESSIONS replay failed to mimic. Upon inspection, we believe that something unusual happened—perhaps on the client-side or on the network—that is not captured in our log because the user was not behaving as expected. This was an interesting observation, contrary to our intuitions, but we cannot make any conclusive claims regarding the replay accuracy of SERVER INACTIVITY over the captured log.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented and evaluated several new approaches to generating test cases from field data that address the loss of multi-user interaction in current user-session-based testing techniques. In terms of generation cost, replay cost, and coverage, our techniques generated alternative suites that provide more multi-user interactions and have comparable coverage to user-session-based testing. However, user-session-based testing does execute some code not executed by our suites.

We have not yet fully compared our approaches to current user-session-based testing techniques. In the future, we plan to evaluate the approaches on multiple applications with different application and usage characteristics. We also will evaluate the test suites in terms of their relative abilities to expose faults. Because we are likely to generate large, redundant test suites from the field data, we also want to compare the reduced suites derived from each suite. Although difficult to evaluate, another metric to consider is the ease with which a user can locate a bug from a test case. Based on these experiments, we can then make recommendations to testers about appropriate test-case generation techniques, given their application and usage characteristics.

Acknowledgments

We thank Frank Zappaterrini for building one of the replay tools and Frank Zappaterrini and Madhu Nayak for customizing DSpace.

7. REFERENCES

[1] A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software Systems and Modeling*, 4(2), April 2005.

[2] R. Binder. *Testing Object-Oriented Systems*. Addison Wesley, 2000.

[3] Clover: Code coverage tool for Java. <http://www.cenqua.com/clover/>, 2006.

[4] G. DiLucca, A. Fasolino, F. Faralli, and U. D. Carlini. Testing web applications. In *International Conference on Software Maintenance*, page 310, October 2002.

[5] DSpace Federation. <http://www.dspace.org/>, 2006.

[6] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering*, pages 49–59. 2003.

[7] HTTPClient V0.3-3. <http://www.innovation.ch/java/HTTPClient/>, 2006.

[8] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 141–154, 2003.

[9] C.-H. Liu, D. C. Kung, and P. Hsia. Object-based data flow testing of web applications. In *First Asia-Pacific Conference on Quality Software*, 2000.

[10] Michal Blumenstyk. Web Application Development - Bridging the Gap between QA and Development. <http://www.stickyminds.com>, 2002.

[11] J. Offutt, Y. Wu, X. Du, and H. Huang. Bypass testing of web applications. In *IEEE 15th International Symposium on Software Reliability Engineering*, pages 187–197, Nov. 2004.

[12] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proceedings of the Third International Workshop on Dynamic Analysis (WODA)*, pages 1–7, May 2005.

[13] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, 2001.

[14] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. Souter. Analyzing clusters of web application user sessions. In *Proceedings of the Third International Workshop on Dynamic Analysis (WODA)*, May 2005.

[15] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and fault detection for web applications. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE05)*, pages 253–262, November 2005.