A Case Study of 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. 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 correct 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 [3, 13].

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 of capture/replay include reproducing failures caused by user input [12] and prioritizing bug fixes based on commonly accessed code [9]. 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 user 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. [7] 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 [14]. In addition to comparing user-session-based and model-based test cases, Elbaum et al. [7] 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 believe we can explain this result; because Elbaum et al. manipulated user sessions without accounting for application state, the application behavior was affected.

During our studies of web application testing with various applications, we found limitations in user sessions as test cases stemming from users sharing application state. Specifically, existing user-session-based testing techniques *ignore multi-user interactions* and, thus, *lose 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 test cases are more realistic than traditional user sessions and their replay covers different code and exposes different faults. 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 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

TAVWEB06 July 2006, Portland, Maine, USA.

Captured Log		Deployed Behavior
User1	index.jsp	Enter bookstore site
User2	addBook.jsp	Add Steve Martin's *Pure Drivel* to database
User1	search.jsp	List Steve Martin's books
User1	buy.jsp	Buy *Pure Drivel*

Figure 1: Multi-user interaction in the deployed behavior of a bookstore application

Replayed Log		Replayed Behavior
User1	index.jsp	Enter bookstore site
User1	search.jsp	List Steve Martin's books (does not include *Pure Drivel*)
User1	buy.jsp	**ERROR:** Attempt to buy *Pure Drivel*, a non-existent title in database
User2	addBook.jsp	Add Steve Martin's *Pure Drivel* to database

Figure 2: Replayed behavior of a bookstore application with user sessions as test cases, losing multi-user interactions

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 VS. CAPTURED LOG REPLAY

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. [7]. Each test case is a user session, where a *user session* is a collection of user requests in the form of a request type (GET or POST), the resource name, 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 session's state to maintain. 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 a user session contains requests for a single user, user-session partitioning loses multi-user interactions that occurred during deployment. A user can 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 application, as in Figure 1. In user-session-based testing, the example captured log is partitioned into two test cases that represent User1 and User2. The test case derived from User1 is replayed first. The emulated User1 will attempt to purchase a non-existent book and will execute error code, as illustrated by Figure 2.

Captured Log		Deployed Behavior
User1	search.jsp	Search for Steve Martin's *Pure Drivel*
User2	buy.jsp?order=0001	Buy *Pure Drivel*
User2	viewOrder.jsp?order=0001	View order details
User1	buy.jsp?order=0002	Buy *Pure Drivel*

Figure 3: Dynamic parameters and multi-user interaction in the deployed behavior of a bookstore application

Replayed Log		Replayed Behavior
User1	search.jsp	Search for Steve Martin's *Pure Drivel*
User1	buy.jsp?order=0001	Buy *Pure Drivel*
User2	buy.jsp?order=0002	Buy *Pure Drivel*
User2	viewOrder.jsp?order=0001	**ERROR:** Incorrect order number

Figure 4: Replayed behavior of a bookstore application with user sessions as test cases, losing multi-user interactions and mismatched dynamic parameters

Suppose instead that we replay logs exactly as captured. Our test case is the entire captured log and successfully exercises the code to purchase the book, with application behavior as in Figure 1. While this replay strategy may be more intuitive than replaying user sessions, it introduces some complexity into the replay 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 request. 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 of out-of-order execution on dynamically generated parameter data can be seen in order numbers from our bookstore application. Turning to a similar bookstore example in Figure 3, 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., "order=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 (Figure 3), her order number is 0001, and User1's order number is 0002. The order numbers are recorded as request parameters in the log. When replaying the user sessions as test cases (Figure 4), User2 is assigned the order number second (0002), but the assigned value (0002) will not correspond to the order number encoded in the HTTP request to view the order (0001). Unless we modify the parameter values in 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 (Figure 3).[1]

As illustrated in these two examples, replaying in log order captures an application's deployed behavior more closely than replaying user sessions. However, replaying in log order may not exactly duplicate the deployed behavior because of a number of factors, such as differences in the timing of incoming requests, the configuration of the system (e.g., not sending email in the testing environment), or nondeterminism of the application itself. Furthermore, if we encounter a bug during replay, it may be 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 in similar ways, 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 both types of test cases will exercise the application in different ways.* The primary advantages of partitioning into user sessions is that each test case isolates a single 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 the disadvantages of both approaches.

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 web applications [1, 5, 10, 11, 14]. 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 5 to illustrate the differences between test case generation strategies. Figure 5 (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 5 (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.

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 in which 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 short 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 Algorithm 1.

Algorithm 1 Fixed-Time Block

Input: log $L = (r_0, r_1, ..., r_n)$, sorted by timestamp, and a time interval *timeint*
Output: test cases $C = (c_0, c_1, ..., c_m)$

```
select the first request r_0 in the captured log L
add request r_0 to test case c_0
t ← timestamp of request r_0
j ← 0
for i = 1 to n do
  t_i ← timestamp of request r_i in log L
  if (t_i − t) > timeint then
    j ← j + 1
    create new test case c_j
    t ← t + timeint {Difference with Server Inactivity}
  end if
  add r_i to test case c_j
end for
```

For the example in Figure 5 (a), we partition the test suite in time blocks of *timeint* equal to 10 minutes to create the four test cases shown in Figure 5 (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**.

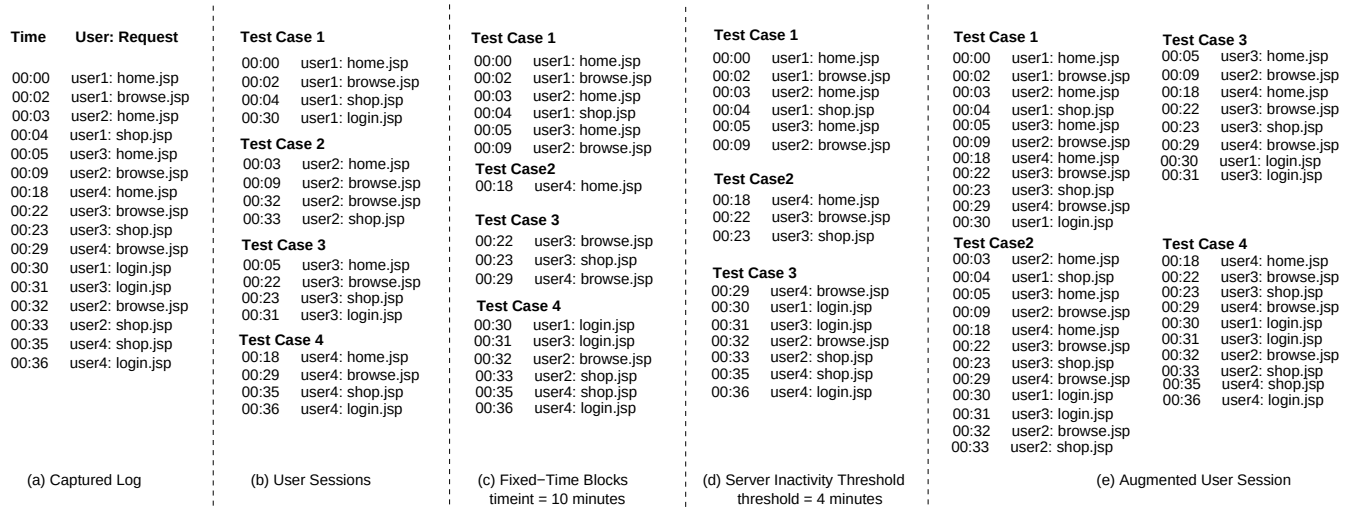

Figure 5: Examples of Generating Test Cases from Field Data

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.

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 in test cases, we propose partitioning based on server inactivity periods (SERVER INACTIVITY). Ideally, SERVER INACTIVITY will preserve more logical user sessions than FIXED-TIME BLOCK, while still maintaining a high level of multi-user interactions.

Algorithm 2 shows the algorithm for generating test cases. Using the partitioning algorithm with a server inactivity period of 4 minutes, the captured log in Figure 5 (a) is partitioned into five test cases (Figure 5 (d)). 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 into individual user sessions as with USER SESSIONS.

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 INACTIVITY can create large test cases with many requests per test case. The overhead of maintaining and executing such large test cases may not be practical, especially in the case of heavy, continuous application usage.

Algorithm 2 SERVER INACTIVITY

Input: log $L = (r_0, r_1, ..., r_n)$, sorted by timestamp, and an inactivity threshold $threshold$
Output: test cases $C = (c_0, c_1, ..., c_m)$

select the first request r_0 in the captured log L
add request r_0 to test case c_0
$t \leftarrow$ timestamp of request r_0
$j \leftarrow 0$
for $i = 1$ to n **do**
  $t_i \leftarrow$ timestamp of request r_i in log L
  if $(t_i - t) > threshold$ **then**
    $j \leftarrow j + 1$
    create new test case c_j
  end if
  $t \leftarrow$ timestamp of request r_i {Difference with FIXED-TIME BLOCK}
  add r_i to test case c_j
end for

3.2 Augmented User Sessions

To address the disadvantage of splitting logical user sessions across test cases 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 (as compared to the captured log) to ease in debugging and testing. The algorithm for generating test cases is shown in Algorithm 3.

The captured log in Figure 5 (a) is converted into AUGMENTED USER SESSIONS test cases as shown in Figure 5 (e). **Test Case 1** contains all the requests made by *user1* and any other requests made during the time that *user1* was using the application. The advantage of AUGMENTED USER SESSIONS is that the test cases capture unbroken sets of logical user sessions, while still capturing multi-user interaction that may affect the state of the system. As shown in Figure 5 (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. By augmenting the user sessions, the number of requests in the suite increased from 16 requests for USER SESSIONS to 41 requests in AUGMENTED USER SESSIONS. Generating these test cases costs

Algorithm 3 AUGMENTED USER SESSIONS

Input: log $L = (r_0, r_1, ..., r_n)$, sorted by timestamp
Output: test cases $C = (c_0, c_1, ..., c_m)$

```
create the set of user sessions U from the captured log L,
as described in Section 2
for all user sessions u_i ∈ U do
  create an empty test case c_i
  f_i ← timestamp of first request in u_i
  l_i ← timestamp of last request in u_i
end for
for all requests r in the captured log L do
  t ← timestamp of r
  for all test cases c_i ∈ C do
    if f_i ≤ t ≤ l_i then
      add r to c_i
    end if
  end for
end for
```

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; Table 1 summarizes the qualitative tradeoffs between the test-case generation techniques. 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; determining 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 initial test case generation cost is a less important consideration than the ability of test cases to model user behavior and the expense of executing the test cases. We compare the actual costs to generate and replay the test cases in Section 5.2.

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 associated 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 [8], 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 emulate replaying each request as the appropriate user, e.g., maintain state for each user, as described in Section 2. For more details about our framework, refer to [15].

The USER SESSIONS, FIXED-TIME BLOCK, and SERVER INACTIVITY algorithms are implemented as presented. For AUGMENTED USER SESSIONS, a single logged request may appear in multiple test cases. If we replayed the complete AUGMENTED USER SESSIONS test suite sequentially, we would replay portions of logical user sessions multiple times. Replaying the complete AUGMENTED USER SESSIONS test suite while maintaining the requests' log order and not duplicating requests is equivalent to replaying the log. For our coverage studies, we replay the log instead of replaying the equivalent AUGMENTED USER SESSIONS test suite. We feel it is reasonable to expect a tester to replay the log rather than the AUGMENTED USER SESSIONS test cases when replaying the whole suite. The advantages of using AUGMENTED USER SESSIONS as test cases are more pronounced in maintenance tasks or test suite reduction scenarios, which we plan to evaluate in future work.

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 partitions or processes 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 [6]. 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 Table 2. The application characteristics include both the JSP and Java code. From the log characteristics in Table 2, 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.

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 timeout; likely to split user sessions across test cases
Server Inactivity	Multi-user interaction; variable-sized test cases	Requires smart threshold; can 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

Source Code Characteristics				Log Characteristics					
Classes	Methods	NCLOC	Statements	Tot URLs	Distinct URLs	25th %	Median Gap	75th %	Avg Gap
355	1,534	61,720	27,136	16,275	443	4 s	24 s	74s	13 mins

Table 2: Subject Application Characteristics

To address the previously stated research goals, we evaluated the generated test cases in terms of effectiveness and efficiency. We used three dependent variables as our measures: program coverage, cost of generation, and replay cost for each test 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 [4] to collect statement, condition, and branch coverage.

We implemented each test case generation technique in Java. We executed each test case generation 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 time intervals of one minute, one hour, and six hours based on the log characteristics in Table 2 and our intuitions about what lengths of time would capture effective test cases for DSpace. We chose an inactivity threshold of 25 minutes for Server Inactivity after analyzing the inactivity gaps in two very different web applications, DSpace and a conference registration/submission manager. We found that 75% of the server inactivity periods were greater than 25 minutes. Therefore, an inactivity period of 25 minutes will not separate 25% of the logical user sessions into their own test cases but will correctly separate 75% of the user sessions. To replay the test cases, we used our customized Java replay tools. Before replaying each test suite, we initialized the state to the application state before we started collecting user accesses. We measured 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 3 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 3, the statement coverage for most of the test suites was comparable (within 400 statements), covering around 65% of the code. We do not expect 100% coverage because using field data to generate 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 before we started logging user requests.

Augmented User Sessions (implemented as Log replay) covers the most statements. Because Augmented User Sessions replays 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. During Augmented User Sessions replay, the observed application behavior did not match deployed behavior, but we do not have a clear explanation for the behavior yet. Factors not seen in the logs, such as fixes to the deployed code, network outages, or unexpected user behavior, may explain the discrepancies. Inevitably, because of DSpace's dependence on state, User Sessions will execute error code (often redundant in our application) instead of the correct behavior. We attribute the much smaller coverage of the hour and minute test suites to splitting the 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 quantify the differences between the suites, we compared the statements covered by User Sessions with the other suites; the results are shown in Table 4. The first column is the combined number of statements that the suites cover, and the second column is the number of covered statements in common among suites generated by User Sessions and the alternate suite. The third column is a measure of how different the suites are: the larger the number, the more different the suites. The fourth column is the number of statements User Sessions covers but the alternative suite (A) does not cover. Lastly, the fifth column 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

Suite		Number of Test Cases	Statements Covered	% Statements Covered	Generation Time (s)	Total No. of Requests	Requests/Test Case (Med/Avg)	Suite Replay Cost (mins)
User Sessions		1,342	17,536	64.6	2	16,275	2/12	76
Fixed-Time Block	Minute	8,447	12,270	45.2	2	16,275	1/2	216
	Hour	1,769	15,713	57.9	3	16,275	2/9	102
	6-hour	508	17,674	65.1	4	16,275	24/32	73
Server Inactivity		1,814	17,745	65.4	2	16,275	10/9	75
Augmented User Sessions (Generation)		1,342	N/A	N/A	8	184,656	10/138	N/A
Captured Log (Replay)		1	17,866	65.7	3	16,275	16,275/16,275	52

Table 3: Comparison of Test Suites

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

Table 4: Comparison of Statement Coverage of Alternative Test Case Generation Techniques (A) with User Sessions (US)

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 when the publication identifier is not valid.

For DSpace, replaying both User Sessions and Augmented User Sessions yields the largest number of statements executed and requires only two hours to replay both types of test cases.

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. We automatically generated each test suite in a few seconds, as shown in Table 3. As expected, generating Augmented User Sessions took the longest amount of time because the number of requests in the suite is 10 times larger than the other suites. Augmented User Sessions's larger size is an indication of how much the user sessions overlap. With larger logs, we might see a greater distinction between the techniques in terms of generation cost. Table 3 also shows the median and average number of requests per test case. On average, test suites generated by Augmented User Sessions are ten times larger than User Sessions.

The cost of replaying each test suite 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 executing our Java replay tool 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 (the log) so that we did not replay the same request multiple times and b) the suite executes less error code; depending on the error code executed, the server can return an empty response, which causes the replay tool to repeat the request.

5.3 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 (because of splitting the logical user sessions).

To maximize DSpace coverage, each test case should maintain logical user sessions within test case boundaries and also 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 appropriate inactivity threshold.

We found that 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 multiple test cases in the suite. We believe that Fixed-Time Block (Minute or Hour) will create suites where later test cases improve coverage of earlier test cases only marginally. In the future, we are interested in studying the marginal increase in coverage by the test cases created from the different test case generation techniques.

Replaying an appropriate reduced suite may be sufficient to address the issue of redundant test cases in the generated suites. After removing the redundant test cases from each test suite, we believe that we will be left with different types of test cases in each suite. For example, reducing User Sessions results in a reduced suite that represents different usage patterns of the application with respect to the user, whereas reducing test suites created by Fixed-Time Block or Server Inactivity will likely select test cases from different application usage cycles with respect to the server. 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 compare their cost-effectiveness.

We are 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 the best time-based techniques 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. 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.

Depending on a tester's goals, the tester may or may not want to mimic deployed behavior because the various behaviors will exercise different application code. From our preliminary results, Augmented User Sessions best imitates the deployed application behavior. Fixed-Time Block and Server Inactivity can imitate deployed application behavior if the time interval/threshold is selected appropriately. However, small time intervals or 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 in its initial configuration 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 ends a 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 relative accuracy between replaying Server Inactivity versus 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 the current user-session-based testing technique. 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.

Based on our subject application, we found Server Inactivity and Augmented User Sessions to be the most effective test case generation mechanisms in terms of code coverage. However, there are generation and replay cost tradeoffs for these two techniques: Augmented User Sessions test cases are more expensive to generate than Server Inactivity (by 5 seconds), but optimized replay for Augmented User Sessions cases is faster than Server Inactivity replay (by 27 minutes).

We have not yet fully compared our approaches to the current user-session-based testing technique. 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. Our fault detection experiments can follow a similar design to those presented in our previous work [16], where we compare the suites based on their abilities to detect seeded faults.

Since 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 how easily 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] M. Blumenstyk. Web Application Development - Bridging the Gap between QA and Development. <http://www.stickyminds.com>, 2002.

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

[5] G. DiLucca, A. Fasolino, F. Faralli, and U. D. Carlini. Testing web applications. In *International Conference on Software Maintenance*, pages 310–319, Washington, DC, USA, October 2002. IEEE Computer Society.

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

[7] 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. IEEE Computer Society, 2003.

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

[9] 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, New York, NY, USA, 2003. ACM Press.

[10] 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.

[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)*, New York, NY, USA, May 2005. ACM Press.

[13] S. Pertet and P. Narsimhan. Causes of failures in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University, December 2005.

[14] F. Ricca and P. Tonella. Analysis and testing of web applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[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, New York, NY, USA, November 2005. ACM Press.

[16] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, and A. Souter. An empirical comparison of test suite reduction techniques for user-session-based testing of web applications. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 587–596, September 2005.