Prioritizing User-session-based Test Cases for Web Applications Testing

Sreedevi Sampath[†], Renée C. Bryce[‡],
Gokulanand Viswanath[†], Vani Kandimalla[‡], A. Güneş Koru[†]
[†] Department of Information Systems
University of Maryland, Baltimore County
{sampath, gokul1, gkoru}@umbc.edu
[‡] School of Computer Science
University of Nevada Las Vegas
{reneebryce, kandimal}@cs.unlv.edu

Abstract

Web applications have rapidly become a critical part of business for many organizations. However, increased usage of web applications has not been reciprocated with corresponding increases in reliability. Unique characteristics, such as quick turnaround time, coupled with growing popularity motivate the need for efficient and effective web application testing strategies. In this paper, we propose several new test suite prioritization strategies for web applications and examine whether these strategies can improve the rate of fault detection for three web applications and their pre-existing test suites. We prioritize test suites by test lengths, frequency of appearance of request sequences, and systematic coverage of parameter-values and their interactions. Experimental results show that the proposed prioritization criteria often improve the rate of fault detection of the test suites when compared to random ordering of test cases. In general, the best prioritization metrics either (1) consider frequency of appearance of sequences of requests or (2) systematically cover combinations of parameter-values as early as possible.

1. Introduction

Web applications are a critical component of many businesses. Failures in this domain result in losses of millions of dollars to organizations [17, 21]. Web applications often have a large and dynamic user base that demands a high level of robustness and reliability. Most web applications must be available 24/7. This requires testers to fix bugs in an application and deploy a new version within a short time frame. In such situations, testers execute regression test suites to ensure that the new version of the application is working as expected. With quick turnaround time being critical for web applications, testers can benefit from test suites that can detect faults early in the test execution cycle.

Unlike other application domains, in web applications, logs of actual usage data are easily available to testers. Converting usage data into test cases is known as user-session-based testing [9, 26]. These user-sessions provide testers with information on exactly how users interact with a web application, making them ideal for test development. A user-session-based test case is a sequence of base requests and parameter-value pairs (also called name-value pairs). Cookies and the originating IP address information are used to convert requests in a web server log into a sequence of user-session-based test cases. Requests with parameter-values specified by a user may access back-end code or retreive/store data in the database, thus exercising complex interactions between application components. Further, user sessions identify the most frequently accessed parts of an application. This is important in testing because frequently accessed components of a system have significant impact on the user-perceived reliability of an application.

In [26], Sampath et al. reduce user-session-based test suites while maintaining their effectiveness. In this paper, we expand upon Sampath et al.'s [26] previous work. However, instead of further reducing the size of test suites, we prioritize user-session-based test suites with the goal of improving the rate of fault detection. Prioritizing test cases is particularly significant in user-session-based testing because limited resources make it difficult to execute all of the test cases that can accumulate for a frequently-used application in the *logged or originally recorded* order.

Several strategies exist to prioritize test cases for C programs [6, 7, 11, 24] and Java programs [4, 5, 11]. We expand upon previous work to develop criteria specifically for web application testing. In particular, we consider frequency of user requests and interactions of parameter-values in requests. The main contributions of this paper are: *(1) Strategies to prioritize user-session-based web application test suites, (2) Empirical evaluation of the strategies using user-session-based test suites, and (3) Guidance to testers based on the results of our empirical evaluation.*

Section 2 presents background in web application testing, user-session-based testing, and test case prioritization.

978-0-7695-3127-4/08 $25.00 © 2008 IEEE
DOI 10.1109/ICST.2008.42

Section 3 presents the prioritization metrics that we use to order user-session-based tests. Section 4 describes our subject applications and experimental methodology. We present and analyze the results in Section 5. Section 6 concludes.

2. Background

2.1. Web Applications

A web application is a set of web pages that are accessible through a browser over a network. A web page can be either static—in which case the content is the same for all users—or dynamic, such that content depends on user input. Web applications may include integration of numerous technologies; third-party reusable modules; a well-defined, layered architecture; dynamically generated pages with dynamic content; and extensions to an application framework. Large web-based software systems can require millions of lines of code, contain many interactions between objects, and involve significant interaction with users. Changes in user profiles and frequent maintenance changes also complicate automated testing [13].

While several approaches exist for model-based web application test case generation [1, 14, 16, 23], to our knowledge there is little work in the area of test case prioritization for web applications.

2.2. User-session-based Testing

Web applications often rapidly evolve. With each evolution, testers need to run regression tests. One source of regression tests for web-applications is that of user-sessions captured from previous releases of the software. A user-session-based test case is a sequence of HTTP requests containing base requests and name-value pairs that are recorded when a user accesses the application. In the example test case in Table 1, for the following request: *Login.jsp&name="john"&pswd="doe"*, the base request is *Login.jsp* and the parameter-value pairs are *name="john"* and *pswd="doe"*. Base requests can be HTTP request accesses to both static and dynamic web page content. In previous work, Sampath et al. [26] and Sprenkle et al. [28] generate user-session-based test cases from usage logs. When available, cookies were used to generate a user-session-based test case. Otherwise, a user-session-based test case begins when a request from a new IP address arrives at the server and ends when the user leaves the web site or the session times out. A 45 minute gap between two requests from a user is considered equivalent to a session timing out. Different strategies can construct test cases for the collected user sessions [9, 20, 22, 27].

Elbaum et al. [9] show that using usage data as test cases is efficient at detecting faults, but unscalable with larger numbers of user sessions. Sampath et al. [26] reduce the size of user-session-based test suites and empirically evaluate the effectiveness of the reduced suites.

The reduction techniques are based on criterion, such as covering all base requests in the application while maintaining the use case representation. The criteria create a test suite smaller than the original suite, but tests are in no particular order. Indeed, test suite reduction techniques strive to reduce the size of a test suite, while maintaining overall fault finding effectiveness. Whereas, test suite prioritization uses the entire test suite for execution, but the test cases are ordered based on pre-determined criteria that attempt to detect *faults as quickly as possible* in the test execution cycle.

2.3. Test Case Prioritization

In the life cycle of an application, a new version of the application is created as a result of (a) bug fixes and (b) requirements modifications [19]. A large number of test cases may be available from testing previous application versions which can be reused to test a newer version of the application. However, running such tests may take a significant amount of time. Rothermel et. al. report an example for which it can take weeks to execute all of the test cases from a previous version [24]. Due to time constraints, a tester must often select and order these test cases for execution.

One approach to selecting test cases is to schedule the test cases according to some *criterion* to satisfy a *performance goal*; scheduling test cases in this manner is known as test case prioritization. Examples of criteria include code coverage, fault likelihood, and fault exposure potential [6, 7, 24]. Binkley uses the semantic differences between two programs to reduce the number of tests that must be run during regression testing [3]. Jones et al. reduce and prioritize test suites that are MC/DC adequate [12]. Lee et al. reduce test suites by using tests that provide coverage of the requirements [15]. Offutt et al. use strategies that reorder tests to select a smaller number of test cases [18].

Additional criteria exist for GUI-based programs. For instance, Bryce and Memon prioritize pre-existing test suites for GUI-based programs by the lengths of tests (i.e., the number of steps in a test case, where a test case is a sequence of events that a user invokes through the GUI), early coverage of all unique events in a test suite, and early event-interaction coverage between windows (i.e., select tests that contain combinations of events invoked from different windows which have not been covered in previously selected tests) [4]. In half of their experiments, event-interaction-based prioritization results in the fastest fault detection rate.

Our work extends prior prioritization work for our specific application to web-based testing. We leverage actual

user-sessions that serve as our test cases and define and evaluate alternate criteria for test case prioritization.

3. Test Case Prioritization Strategies

The test suite prioritization problem is defined in [24]. Given (T, Π, f), where T is a test suite, Π is the set of all test suites that are prioritized orderings of T obtained by permuting the tests of T, and f is a function to evaluate the orderings from Π to the real numbers, the problem is to find a permutation, $\pi \in \Pi$ such that $\forall \pi' \in \Pi, f(\pi) \geq f(\pi')$.

Prioritization can be based on any criteria. Examples include code coverage, cost estimates, event coverage, and others [4, 7, 8, 24, 30]. In this work, we exploit the characteristics of user-session-based test cases and examine the following functions (or criteria) for web-based applications (each described in more detail in Sections 3.1-3.4):

- **Test length based on number of base requests** (**Req-LtoS**, **Req-StoL**): order by the number of HTTP requests in a test case
- **Frequency-based prioritization** (**MFAS**, **AAS**): order such that test cases that cover most frequently accessed pages/sequence of pages are selected for execution before test cases that exercise the less frequently accessed pages/sequences of pages.
- **Unique coverage of parameter-values** (**1-way**): order tests to cover all unique parameter-values as soon as possible
- **2-way parameter-value interaction coverage** (**2-way**): order tests to cover all pairwise combinations of parameter-values between pages as soon as possible
- **Test length based on number of parameter-values**(**PV-LtoS**, **PV-StoL**): order by number of parameter-values used in a test case
- **Random**: randomly permute the order of tests

3.1. Test Lengths

Prioritization of length by base requests selects a next test with the maximum count of base requests, counting duplicates. Since the number of requests in a test case partially determines how much of the application code is exercised by the test case, ordering test cases based on their length can affect the rate of fault detection of the ordered test suite. We order test cases in descending (**Req-LtoS**) and ascending (**Req-StoL**) order of length, where length of a test case is defined as the number of base requests the test case contains. For the example test case *tc1* in Table 1, the length of *tc1* is the number of base requests in *tc1*, i.e., four.

Test case tc1
Register.jsp&name=john&pswd=doe&fname=John&lname=Doe
Login.jsp&name=john&pswd=doe
Search.jsp&bookid=10
Logout.jsp

Base request	**Parameter-value pairs**
Register.jsp	name=john, pswd1=doe, fname=John, lname=Doe
Login.jsp	name=john, pswd=doe
Search.jsp	bookid=10
Logout.jsp	null

Table 1. Example Test Case

3.2. Frequency-based Prioritization

In this prioritization methodology, we prioritize test cases based on the count of the most-frequently accessed sequences of pages that appear in the test case. Since failures in frequently accessed application components have more impact on user-perceived reliability of the application, we hypothesize that by favoring test cases that exercise frequently accessed application components, the rate of fault detection of the ordered test suite can be improved.

We identify the total number of times that each unique sequence of pages is accessed in the entire test suite and construct a frequency table. We consider sequences in terms of base requests (i.e., ignoring the parameter-value pairs) and only sequences that involve interactions between JSP and Java servlet pages, i.e., we do not include sequences that contain static HTML pages. We assume that faults will exist in the application code and thus consider only sequences between pages that access application code. In this paper we consider page sequences of size 2. Using the frequency table to identify the most frequently accessed sequences, we then prioritize the test cases in two ways.

Most Frequently Accessed Sequence (MFAS). This approach identifies the most frequently accessed request sequence, s_i, in the test suite and orders test cases in decreasing order of the number of times that s_i appears in the test case.

All Accessed Sequences (AAS). In **AAS**, the frequency of access of all sequences is used to order the test suite. For each sequence, s_i, in the application, beginning with the most frequently accessed sequence, test cases that have maximum occurrences of these sequences are selected for execution before other test cases in the test suite. Figure 1 describes the **AAS** test case prioritization algorithm.

Consider the example shown in Table 2. Sequences *s1, s2, s3,* and *s4* represent all the sequences of pages in the test suite, ordered in decreasing order of frequency of access. Starting with sequence *s1*, one test case, say *tc1* is selected at random. Then, all of the other sequences that *tc1* covers are marked satisfied and removed from further consideration, in this example the only other sequence that *tc1* cov-

Algorithm: *All Accessed Sequences*
Input:set of sequences in web application, S,
set of test cases T
Output: Prioritized order of test cases, P
OS = Sequences $s_i \in S$ ordered in
decreasing order of frequency of access
foreach sequence $os_i \in OS$
$T(os_i)$=test cases $t \in T$ such that t has
maximum occurrences of os_i in T
end foreach
do
Select sequence os_i not yet satisfied
Select test case t from $T(os_i)$ at random and add to P
Mark all other sequences that t covers as satisfied
until all sequences $os_i \in OS$ have been satisfied
Randomly order test cases $t \in T$ not yet selected and add to P

Figure 1. All Accessed Sequences Algorithm

ers is *s4*. Then, the test cases that cover *s2* are considered and one is selected at random, assume *tc3*. Since *tc3* does not cover any sequence other than *s2* maximum number of times, only *s2* is marked as satisfied in this step. The algorithm continues selecting test cases until all of the test cases from the table are ordered. Remaining test cases that do not appear in the table are randomly ordered and appended.

Sequence Name	Tot. No. of Occurrences	Tests with max. occurrences of sequence
s1:$< Register, Login >$	10	tc1, tc2
s2:$< Login, Default >$	7	tc3, tc4
s3:$< Default, Regsiter >$	5	tc2, tc5
s4:$< Register, Shop >$	4	tc1, tc4
Prioritized Order: tc1-tc3-tc2-tc5-tc4		

Table 2. AAS Example

3.3. Systematic Prioritization by Parameter-Values

Web applications contain *pages* that contain *parameters* for which users may specify *values*. For instance, consider the example test case in Table 1. The *Login.jsp* page accessed in the test case has two *parameters*, "name" and "pswd" that can take on values. The user sessions that we prioritize include a discrete number of values that have been specified for these parameters. For instance, test case, *tc1* in Table 1 has the *parameter* "name" set to the *value* "john". We refer to this as a *parameter-value*.

Log-in	Member Type	Discount Status	Shipping Method
New Member	Basic	None	Standard
Member (logged in)	Silver	$10 off	Express
Member (not logged in)	Gold	Free Ship.	Overnight

Table 3. Four parameters can take on one of three values each

Unique parameter-value coverage. The **1-way** criterion selects a next test that maximizes the number of parameter-values that have not appeared in previously selected tests.

Test No.	Log-in	Member Type	Discount Status	Shipping Method
1	New Member	Basic	None	Standard
2	New Member	Basic	$10 Off	Express
3	New Member	Basic	Free Ship.	Overnight
4	Member (logged in)	Silver	None	Overnight
5	Member (logged in)	Gold	$10 Off	Standard
6	Member (not logged in)	Basic	$10 Off	Overnight

Table 4. A set of test cases

Parameter-value Interaction Coverage. The **2-way** criterion selects a next test that maximizes the number of t-way parameter-value interactions between pages that occur in a test. In this paper, we set t=2 for pairwise coverage of parameter-values. Consider the example of 4 parameters that can each take on one of three values from Table 3. Table 4 shows an example of parameter-values that occur in a set of test cases. Table 5 lists the six pairwise parameter-value interactions that occur in Test 1. In our experiments, we count the number of previously uncovered parameter-values in each test and prioritize by selecting the test with the maximum count.

Test No. 1	
Pair	**Parameter-values**
1	(New Member, Basic)
2	(New Member, None)
3	(New Member, Standard (5-7))
4	(Basic, None)
5	(Basic, Standard (5-7))
6	(None, Standard (5-7))

Table 5. 2-way Parameter-Value Interactions

Length by parameter-value counts. During a user-session, a user may specify any number of parameter-values. We prioritize tests by the number of parameter-values in a test case (duplicates included). This includes selecting those tests with the largest number of parameter-values in a test first, called **PV-LtoS**. We also prioritize in the opposite manner by selecting those tests with the smallest number of parameter-values first, called **PV-StoL**.

3.4. Random

Prioritization by **Random** selects a next test at random. In this work, we randomly select test cases for the prioritized test suite until there are no more test cases to select.

4. Experimental Evaluation

In our experiments, we study the effectiveness of the prioritization strategies by evaluating their fault detection rate.

Independent and Dependent Variables. Independent variables in our study are the user-session-based test suites, the seeded faults and the test case prioritization techniques. Dependent variables are rate of fault detection, average percent of faults detected (APFD) [24], and test execution time.

Subject Applications and Test Suites. We used three web-based applications and their pre-existing test suites, where test suites are the previously recorded user-sessions in experiments by Sampath et al. [26] and Sprenkle et al. [28]. The subject programs have different characteristics: an open-source, e-commerce bookstore (Book) [10], a Course Project Manager (CPM), and the web application used for the Mid-Atlantic Symposium on Programming Languages and Systems (Masplas). Table 6 shows the subject programs and test suite characteristics.

Book. Book allows users to register, login, browse for books, search for books by keyword, rate books, add books to a shopping cart, modify personal information, and logout. Since our interest was in testing consumer functionality, we did not include the administration code in our experiments [26]. Book uses JSP for its front-end and a MySQL database back-end. Sampath et al. [26] collected 125 test cases by sending emails to local newsgroups and by posting advertisements in the University of Delaware's classifieds web page asking for volunteer users.

CPM. In CPM, course instructors login and create *grader* accounts for teaching assistants. Instructors and teaching assistants create *group* accounts for students, assign grades, and create schedules for demonstration time slots. Users interact with an HTML application interface generated by Java servlets and JSPs. CPM manages state in a file-based datastore. Sampath et al. [26] and Sprenkle et al. [28] collected 890 test cases from instructors, teaching assistants, and students using CPM during the 2004-05 and 2005-06 academic years at the University of Delaware.

Masplas. Masplas is a web application developed at the University of Delaware for a regional workshop. Users can register for the workshop, upload abstracts and papers, and view the schedule, proceedings, and other related information. Masplas is written using Java, JSP, and MySQL. Sampath et al. [26, 25] and Sprenkle et al. [29] collected 169 test cases that we use in our experiments.

Evaluation Metrics. For evaluating the prioritization techniques, we assume that prior knowledge of the faults detected by the regression test suites is available to the testers. We evaluate the test suites with respect to their *rate of fault detection*, the *average percent of faults detected* (APFD) [24], and the *test suite execution time*. The *rate of fault detection* is defined as the total number of faults detected for a given subset of the prioritized test case order. We present the *average percent of faults detected (APFD)* using the notation in [24]. For a test suite, *T* with *n* test cases, if *F* is a set of *m* faults detected by *T*, then let TF_i be the position of the first test case *t* in T', where T' is an ordering of *T*, that detects fault *i*. Then, the APFD metric for T' is given as

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + ... + TF_n}{mn} + \frac{1}{2n} \quad (1)$$

Informally, APFD measures the area under the curve that plots test suite fraction and the number of faults detected by the prioritized test case order.

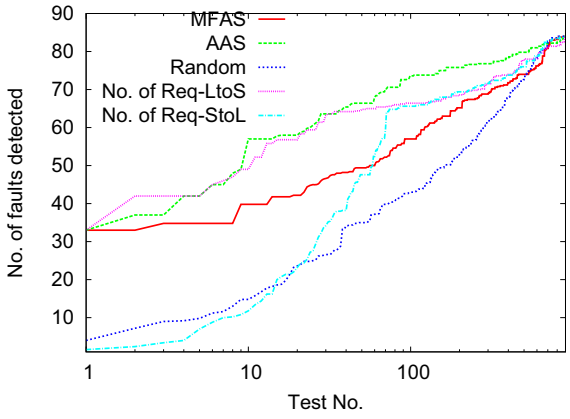
In our empirical study we are interested in *finding the most faults in the earliest tests (i.e., in the first 10% of the tests executed)* and *locating 100% of the faults earliest*.

We also measure the time taken by each prioritized suite to achieve 100% fault detection. From Sprenkle et al. [28] and through recent experiments, we measured the execution time for replaying the entire suite in the logged or originally recorded order. We then compute the average time per request by dividing the total suite execution time by the total the number of requests in the test suite. We use the average time per request to compute the execution time per test case for each of the prioritized order of test cases. Note that these execution times are the time taken for a replay of test cases in the logged order and not the time taken for fault detection replay (difference between the two types of replay is described in Sprenkle et al. [28]).

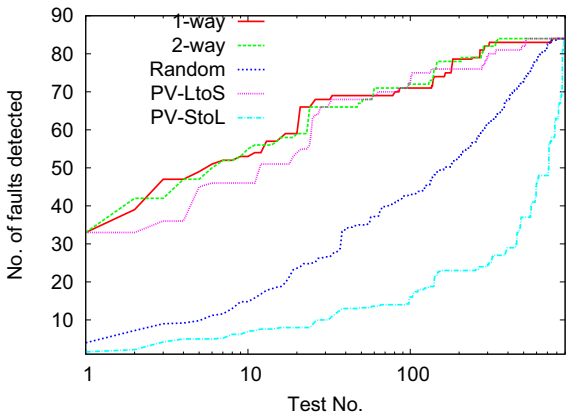
Experimental Methodology. From previous work by Sampath et al. [26] and Sprenkle et al. [28, 29] we had information on how many faults are detected by each test case, i.e., a fault matrix, mapping each test case to the faults detected by test case. The fault matrices used in this paper are generated by using the *struct* oracle for CPM and Masplas and the *diff* oracle for Book [28, 29]. As described in [26, 28], faults were seeded manually by graduate and undergraduate students. In addition, some naturally occurring faults discovered during deployment were also seeded in the applications [26]. In general, five types of faults were seeded into the applications—data store (faults that exercise application code interacting with the data store), logic (application code logic errors in the data and control flow), form

Metrics	Book	CPM	Masplas
Classes	11	75	9
Methods	319	173	22
Conditions	1720	1260	108
Non-commented Lines of Code	7615	9401	999
Seeded faults	40	135	29
Total number of user sessions	125	890	169
Total number of requests accessed	3640	12352	1107
Number of unique requests	10	69	24
Largest user session in number of requests	160	585	69
Average user session in number of requests	29	14	7
Number of unique parameter-values	1,415	4,146	645
% of 2-way parameter-value interactions covered in pre-existing test suite	92.5%	97.8%	96.2%

Table 6. Subject Applications and Test Suite Characteristics



(a) Results using Frequency and Request Length-based Metrics



(b) Results using Parameter-Value Interaction-based Metrics

Figure 2. CPM: Rate of fault detection (log scale)

(modifications to parameter-value pairs and form actions), appearance (faults which change the way in which the user views the page), and link (faults that change the hyperlinks location) [26]. Fault categories are not mutually exclusive.

In [28], Sprenkle et al. propose the *with_state* replay mechanism used for collecting the fault matrices that we use in this paper. In *with_state* test suite execution on a faulty application version, the application state is restored before each test case is replayed on the fault-seeded version so that replay closely matches the clean (non-fault seeded) execution. Thus, faults that affect the underlying application state do not propagate in the application state for latter test cases to expose.

We implemented each of the prioritization techniques as described in Section 3 in C++ and Perl. In all of the implementations, in case of tie between two or more tests that meet the prioritization criterion, a random tie-breaking strategy is implemented. To account for the non-determinism introduced by random tie breaking, we execute each prioritization technique five times and report the average rate of fault detection, APFD, and execution time.

Threats to Validity. Threats to validity in this experiment include our ability to generalize the results to all web-based applications. While we examine three actual web-based applications, the trends observed may not be representative of all web-based applications. Further, many of the faults were manually seeded into the application. Due to challenges faced in seeding faults in object-oriented code, in addition to the challenges of detecting hand-seeded faults described in [2], faults may not be evenly distributed in all the classes. Though we tried to model the seeded faults similar to naturally occurring faults—we even included a few naturally occurring faults in CPM—some of the seeded faults may not be representative of naturally occurring faults. The oracle comparators used in the study can have false positives or false negatives [29]. The execution times reported in this paper are an approximation based on the total time taken to execute the entire test suite. Actual individual test case execution times could vary depending on the test case's characteristics. Also, the test suites for the three applications were not executed on the same machine—execution time comparisons across applications would not be fair.

5. Results and Analysis

5.1. CPM

Figure 2, Table 7 and Table 8 show the results for CPM. We show the rate of fault detection for different prioritization techniques in logarithmic scale. To better show the trends between the prioritization techniques, for each application we divide the prioritization techniques into two sets and present them in two graphs. For CPM, Figure 2(a) shows the rate of fault detection for the frequency-based techniques (**MFAS**, **AAS**), length based on number of base requests (**Req-LtoS**, **Req-StoL**), and **Random**. Figure 2(b) shows the rate of fault detection for parameter-value interaction (**1-way**, **2-way**, **PV-LtoS**, **PV-StoL**), and **Random**. Table 7 shows the APFD in increments 10% of the number of executed tests. The bold-faced numbers in Table 7 show the prioritization technique with highest APFD for the corresponding percentage of the test suite run. Table 8 shows the actual execution time of the subset of tests that identify 100% of the faults. We use the same notation for showing the results for Masplas and Book.

Finding the most faults in the earliest tests. From Table 7 and Figure 2(a), if the APFD in the first 10% of the tests

Percentage of test suite run	MFAS	AAS	Req-LtoS	Req-StoL	Random	1-way	2-way	PV-LtoS	PV-StoL
10	65.43	**85.28**	78.17	75.14	48.63	83.79	83.72	83.53	16.38
20	74.16	88.52	80.34	77.76	57.55	87.78	**90.8**	88.77	25.6
30	77.75	89.4	81.77	80.27	64.51	91.54	**91.72**	88.77	26.44
40	79.61	89.86	84.58	81.39	69.19	94.79	**95.64**	92.71	28.76
50	80.92	91.04	85.58	82.95	73.03	94.79	**95.64**	92.71	30.33
60	81.71	91.58	87.14	84.44	75.37	94.79	**95.64**	94.26	34.64
70	82.73	92.1	87.74	85.15	77.37	94.79	**95.64**	94.26	39.15
80	84.28	92.35	88.27	86.21	78.24	94.79	**95.64**	94.26	39.58
90	84.57	92.37	88.3	86.31	78.45	94.99	**95.64**	94.26	42.18
100	84.64	92.45	88.36	86.35	78.49	94.99	**95.64**	94.26	43.09

Table 7. CPM: APFD Metric (in percentages). 10% of test suite = 89 test cases

	AAS	MFAS	Req-LtoS	Req-StoL	Random	1-way	2-way	PV-LtoS	PV-StoL
CPM	98.76 (882)	99.44 (884)	99.78 (887)	97.64(873)	91.46(817)	83.26(813)	**38.88 (618)**	58.20 (746)	100 (889)
Masplas	79.29 (60)	80.47 (61)	89.94 (67)	82.25 (45)	73.96 (55)	40.24 (35)	**33.73 (36)**	42.01 (46)	97.04 (71)
Book	80 (994)	99.20 (1169)	92.80 (1121)	100 (1177)	99.20 (1172)	57.60 (907)	66.40 (1024)	60.80 (1002)	**54.40 (300)**

Table 8. Percentage of the test suite run (Execution Time in seconds) for 100% Fault Detection

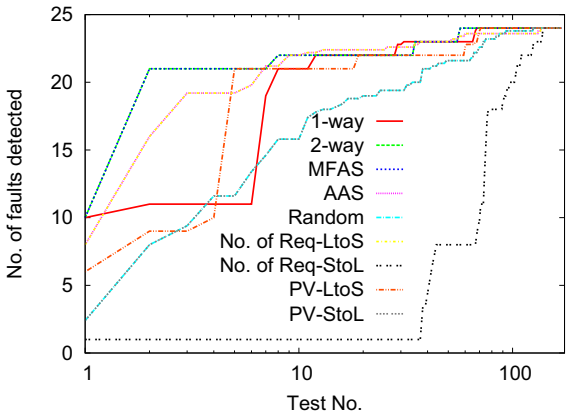
that are run is of primary concern, the **AAS** prioritization technique is the most effective technique.

Locating 100% of the faults earliest. From Table 7, after the first 10% of the test suite is run, prioritization by **2-way** has the best APFD for the remaining 90% of the test suite. Also, from Table 8 we see that **2-way** finds all of the faults before any of the other techniques (with 38.8% of the test suite in 618 seconds). Prioritization by **PV-StoL** has the lowest APFD among the different prioritization techniques. The remaining prioritization techniques fall in between these best and worst cases of APFD. For instance, prioritization by **1-way** and by **PV-LtoS** are generally the second most effective techniques in the latter 90% of the tests run. Prioritization by **MFAS**, **Random**, **Req-StoL**, **Req-LtoS** are less effective than the other techniques.

5.2. Masplas

Finding the most faults in the earliest tests. Figure 3 and Table 9 show that prioritization by **Req-LtoS** is the most effective if APFD during the first 30% of the test suite is of primary concern. The results for **AAS**, **1-way** and **2-way** are slightly less competitive than **Req-LtoS**.

Locating 100% of the faults earliest. From Table 9, we note that after the first 30% of the test suite is run, prioritization by **2-way** has the best APFD for the remaining 70% of the test suite. From Table 8, **2-way** finds all of the faults with 33.3% of the test suite in 36 seconds, followed closely by **1-way**. From Table 9, in the last 70% of the test suite, **AAS**, **Req-LtoS**, **PV-LtoS** are comparable in their APFD. **PV-StoL**'s APFD suggests that it is the least effective prioritization technique. The remaining prioritization techniques fall in between these best and worst cases.



Results using All Prioritization Metrics (log scale)

Figure 3. Masplas: Rate of fault detection.

Finding the most faults in the earliest tests. Table 10 shows that **1-way** has the highest APFD in the first 20% of the test suite's execution. Figure 4 shows that **PV-StoL** and **Req-StoL** are slow starters during the first 10% of tests run, i.e., the first test case in each technique detects only 6 faults, whereas the first test case in the other techniques detects between 15 and 24 faults.

5.3. Book

Locating 100% of the faults earliest. Table 10 shows that prioritization by **1-way** and **MFAS** have a high APFD.

Percentage of test suite run	MFAS	AAS	Req-LtoS	Req-StoL	Random	1-way	2-way	PV-LtoS	PV-StoL
10	78.54	92.17	**95.12**	81.5	76.33	89.6	90.98	86.05	4.44
20	84.28	92.88	**95.12**	91.06	80.51	93.04	90.98	89.74	4.44
30	88.86	94.19	**95.12**	91.06	85.57	93.04	94.28	89.74	26.61
40	90.42	95.86	95.68	91.59	87.59	95.56	**97.06**	93.38	30.08
50	91.34	95.86	95.68	91.59	89.91	95.56	**97.06**	94.84	50.16
60	91.7	95.86	95.68	91.59	90.69	95.56	**97.06**	94.84	53.91
70	91.99	95.86	95.97	91.89	90.69	95.56	**97.06**	94.84	57
80	92.16	96.21	96.14	92.08	90.91	95.56	**97.06**	94.84	58.1
90	92.16	96.21	96.22	92.17	90.91	95.56	**97.06**	94.84	58.85
100	92.16	96.21	96.22	92.2	90.91	95.56	**97.06**	94.84	58.85

Table 9. Masplas: APFD (in percentages). 10% of test suite $\approx$ 18 test cases

From Table 8, **PV-StoL** detects 100% of the faults earliest, with 54% of the test suite in 300 seconds but has the lowest overall APFD. **2-way**, **PV-LtoS**, **AAS** are the next best prioritization metrics with respect to their APFD. **Req-StoL** performs poorly, in general, indicating that the number of base requests in the test case affect the fault detection rate.

Fault Detection Density. From Table 10, we note that **Random** creates a reasonably effective test order with APFD comparable to the other techniques. To study **Random**'s behavior in Book, we define a metric called the *fault detection density*, which is a measure of the average number of faults detected by each test case. Given a set of test cases, $t_i \in T$ and a set of faults F detected by test cases in T, let tf_i be the number of faults detected by t_i, then the fault detection density,

$$FDD = \frac{tf_1 + tf_2 + ... + tf_n}{|T| * |F|} \quad (2)$$

FDD is the ratio of the sum of the *total number of faults detected by each test case* and *the total number of test cases*, normalized to the *total number of faults detected*. A fault detection density of 1 for a test suite indicates that each test case in the suite detects every fault. We found that Book's test cases have a FDD of 0.59 (compared to 0.056 for CPM and 0.19 for Masplas). With a small test suite size (125 test cases) and a high FDD, **Random** has a greater chance of selecting a test case that detects several of the web application's faults and thus creates an effective test suite order.

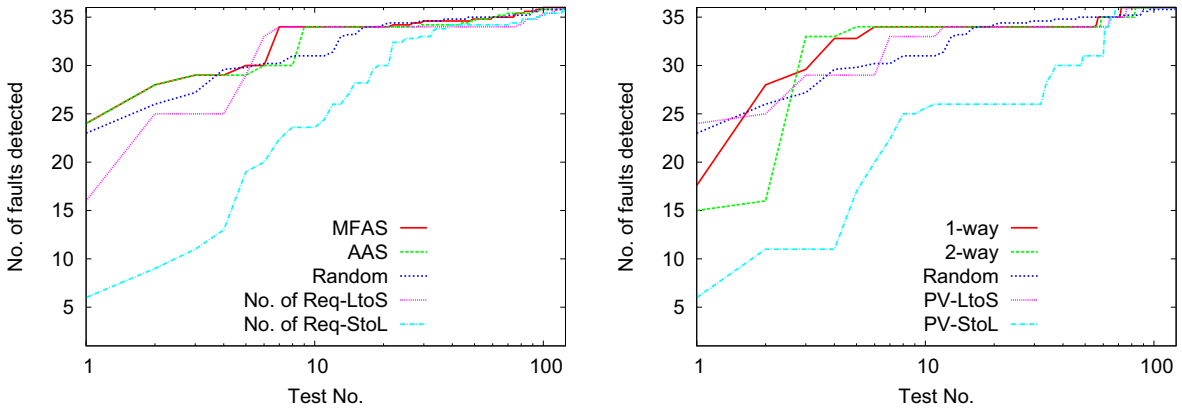
5.4. Summary of Results

Our experimental results show that none of our prioritization criteria is clearly the "best criteria" for all three of our web-based applications. However, for two of our three applications, prioritization techniques that consider parameter-value counts or interactions find 100% of faults before the other techniques. In particular **2-way** prioritization finds all of the faults with 38% of the test suite for CPM in 618 seconds, and 33% of the test suite for Masplas in 36 seconds. In both these applications, **2-way** has the highest APFD values overall (after 100% of the test suite is executed). In Book, however, **MFAS** has the highest overall APFD, followed by **1-way**.

In CPM, **AAS** is the best choice if effectiveness of the first 10% of tests is considered. If the APFD during any of the latter part of the test suite is considered, prioritization by **2-way** is most effective. In Masplas, **Req-LtoS** is the best technique when the APFD during the first 30% of the test suite is of primary concern. In the latter 70% of the test suite, giving preference to covering every 2-way parameter-value interaction creates the most effective test suite order. In Book all the metrics other than **Req-StoL**, **PV-StoL**, **Random** are good options if the goal is to achieve good fault detection early (first 10%) in the test cycle. However, **PV-StoL** is the best prioritization technique, if the goal is to achieve 100% fault detection with the smallest test number of test cases but has the overall lowest APFD. The frequency-based and the parameter-value interaction coverage techniques are better at detecting more faults early in the test execution cycle. Though it appears that **Random** creates an effective test suite ordering for Book, it is important to note that with larger number of test cases and low fault detection densities, **Random**'s effectiveness will decrease.

In terms of execution time, in CPM we observe that **2-way** detects 100% of the faults 30% faster than the worst technique, **PV-StoL**, and in Masplas **2-way** detects 100% of the faults 40% faster than the worst technique **PV-StoL**. **PV-StoL** in Book has the fastest rate of fault detection and detects 100% of the faults 74.5% faster than the worst technique, **Req-LtoS**, but has the lowest overall APFD.

Guidance to Testers. From our results, we find that depending on the tester's goal and the characteristics of the web applications, different prioritization strategies may be useful to a tester. If a tester's primary goal is to find 100% of the faults as soon as possible, prioritization by **2-way** and **PV-LtoS** are good choices. This indicates that systematically covering parameter-values is important when pri-



(a) Results using Frequency and Request Length-based Metrics

(b) Results using Parameter-Value Interaction-based Metrics

Figure 4. Book: Rate of fault detection (log scale).

Percentage of test suite run	MFAS	AAS	Req-LtoS	Req-StoL	Random	1-way	2-way	PV-LtoS	PV-StoL
10.4	93.33	93.13	92.96	70.04	90.34	**93.44**	93.22	93.11	70.13
20	93.79	93.13	92.96	86.09	93.7	**93.44**	93.22	93.11	70.13
30	**94.65**	93.56	92.96	88.15	94.52	93.44	93.22	93.11	78.17
40	**94.99**	94.26	92.96	88.91	94.86	93.44	93.22	93.11	79.86
50	**95.28**	95.16	92.96	88.91	94.86	94.96	94.69	93.11	84.12
60	**95.5**	95.41	92.96	89.15	95.11	96.13	94.69	94.47	86.73
70	95.9	95.41	93.74	89.54	95.27	**96.13**	95.62	95.56	86.73
80	**96.16**	95.69	94.11	89.81	95.56	96.13	95.62	95.56	86.73
90	**96.16**	95.69	94.18	89.92	95.56	96.13	95.62	95.56	86.73
100	**96.16**	95.69	94.27	89.94	95.57	96.13	95.62	95.56	86.73

Table 10. Book: APFD Metric (in percentages). 10% of test suite $\approx$ 13 test cases

oritizing these test suites. Similarly, if a web application has a large number of parameter-values (similar to CPM, in our study) the tester may benefit from using the systematic coverage of parameter-value interaction metrics. However, if an organization determines that certain pages or certain functionality is critical to the functioning of their web application, and hence their business—then these pages or sequences of pages can be identified (by either measuring their frequency in the web logs, or by some other means) and frequency-based prioritization metrics could be used.

Based on our results from the FDD metric for Book, we believe that if a tester knows that their test suite has a high fault detection density, they may benefit by ordering test cases randomly. Given no particular tester preference/application characteristic, if the FDD values of pre-existing test suites are low, our results suggest that longer test cases achieve better fault detection than shorter test cases. In this case, instead of random ordering, the tester would benefit by prioritizing by number of base requests.

From the execution time results, we note that choosing the right prioritization could help the tester find and fix faults in the application quickly, which could translate into thousands of dollars in cost savings. Though the actual execution times for our applications are small, since they represent only the time to replay the test suite and do not include the time taken to manually or automatically identify and locate faults, our results indicate that choosing a prioritization technique can have a big time, and thus cost, impact.

6. Conclusions and Future Work

The web-application domain has an advantage that actual user-sessions can be recorded and used for regression testing. While these tests are indicative of user's interactions with the system, selecting and prioritizing user-sessions has not been thoroughly studied. In this paper, we examine prioritization of such user-sessions for three web applications. We apply several new prioritization criteria to these test suites to identify whether they can be used to increase the rate of fault detection. Results from the experiments show that prioritization by frequency metrics and systematic coverage of parameter-value interactions may in-

crease the rate of fault detection for web applications.

In the future, we will consider the costs associated with the prioritization strategies and extend these studies to consider hybrid approaches to prioritization that combine more than one strategy for prioritizing test cases. Finally, we will extend our frequency-based metrics to consider other metrics, such as critical components of the application as determined by an organization using the web application, or module size. Further, for some web applications, a sequence of size 2 may not be large enough to capture faults in the application [25]. We plan to extend the frequency-based metrics to include sequences of size greater than two. We also plan to study whether the prioritization strategies can be applied to other types of web application test cases that are in the form of HTTP requests.

Acknowledgements. We thank Sara Sprenkle at Washington and Lee University for running experiments for CPM and Masplas execution time results. We thank Richard Kuhn, Raghu Kacker, Jeff Lei, Atif Memon and Sara Sprenkle for providing feedback on the paper.

References

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

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *the Intl. Conf. on Software Engineering*, pages 402–411, May 2005.

[3] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *the Intl. Conf. on Software Maintenance*, pages 41–50, Nov. 1992.

[4] R. C. Bryce and A. M. Memon. Test suite prioritization by interaction coverage. In *the Workshop on Domain-Specific Approaches to Software Test Automation*, pages 1–7, Sep. 2007.

[5] H. Do, G. Rothermel, and A. Kinneer. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. In *the Intl. Symp. on Software Reliability Engineering*, pages 113–124, Nov. 2004.

[6] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *the Intl. Symp. on Software Testing and Analysis*, pages 102–112, Aug. 2000.

[7] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. On Software Engineering*, 28(2):159–182, Feb. 2002.

[8] S. Elbaum, G. Rothermel, S. Kanduri, and A. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, Sep. 2004.

[9] S. Elbaum, G. Rothermel, S. Karre, and M. F. II. Leveraging user session data to support web application testing. *IEEE Trans. on Software Engineering*, 31(3):187–202, May 2005.

[10] Open source web applications with source code. `http://www.gotocode.com`, 2006.

[11] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *the Intl. Computer Software and Applications Conf.*, pages 411–418, Sep. 2006.

[12] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition / decision coverage. *Trans. on Software Engineering*, 29(3):195–209, Mar. 2003.

[13] E. Kirda, M. Jazayeri, C. Kerer, and M. Schranz. Experiences in engineering flexible web service. *IEEE MultiMedia*, 8(1):58–65, Jan. 2001.

[14] D. C. Kung, C.-H. Liu, and P. Hsia. An object-oriented web test model for testing web applications. In *The Asia-Pacific Conf. on Quality Software*, pages 111–120, Oct. 2000.

[15] J. Lee and X. He. A methodology for test selection. *Journal of Systems and Software*, 13(3):177–185, Nov. 1990.

[16] G. D. Lucca, A. Fasolino, F. Faralli, and U. D. Carlini. Testing web applications. In *the IEEE Intl. Conf. on Software Maintenance*, pages 310–319, Oct. 2002.

[17] Michal Blumenstyk. Web Application Development–Bridging the Gap between QA and Development. `http://www.stickyminds.com`.

[18] J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *Intl. Conf. on Testing Computer Software*, pages 111–123, Jun. 1995.

[19] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, May 1988.

[20] Parasoft WebKing. `http://www.parasoft.com`, 2004.

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

[22] Rational Robot. `http://www.ibm.com/software/awdtools/tester/robot/`, 2006.

[23] F. Ricca and P. Tonella. Analysis and testing of web applications. In *the Intl. Conf. on Software Engineering*, pages 25–34, May 2001.

[24] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. on Software Engineering*, 27(10):929–948, Oct. 2001.

[25] S. Sampath, S. Sprenkle, E. Gibson, and L. Pollock. Web Application Testing with Customized Test Requirements—An Experimental Comparison Study. In *the Intl. Symp. on Software Reliability Engineering*, pages 266–278, Nov. 2006.

[26] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. S. Greenwald. Applying concept analysis to user-session-based testing of web applications. *IEEE Trans. on Software Engineering*, 33(10):643–658, Oct. 2007.

[27] J. Sant, A. Souter, and L. Greenwald. An exploration of statistical models of automated test case generation. In *the Intl. Workshop on Dynamic Analysis*, pages 1–7, May 2005.

[28] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *The Intl. Conf. of Automated Software Engineering*, pages 253–262, Nov. 2005.

[29] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott. Automated oracle comparators for testing web applications. In *the Intl. Symp. on Software Reliability Engineering*, pages 253–262, November 2007.

[30] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *the Int. Symp. on Software Testing and Analysis*, pages 97–106, Jul. 2002.