2012

Combinatorial-Based Prioritization For User-Session-Based Test Suites

Schuyler Manchester
Utah State University

COMBINATORIAL-BASED PRIORITIZATION FOR USER-SESSION-BASED TEST SUITES

by

Schuyler Manchester

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Renée Bryce Major Professor	Dr. Dan Watson Committee Member
Dr. Curtis Dyreson Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2012

ABSTRACT

Combinatorial-Based Prioritization for User-Session-Based Test Suites

by

Schuyler Manchester, Master of Science

Utah State University, 2012

Major Professor: Dr. Renée Bryce
Department: Computer Science

Faults in software systems often occur due to interactions between parameters. Several studies show that faults are caused by two-way through six-way interactions of parameters. Studies have also shown that prioritization by two-way inter-window parameter interaction coverage is effective at finding faults quickly in the test execution cycle. However, since faults may be caused by interactions between more than two parameters, we provide a greedy algorithm for test suite prioritization by n-way combinatorial coverage of inter-window parameter interactions. While algorithms that generate combinatorial interaction test suites enumerate all possible t-tuples, we have observed that user-session-based test suites often do not contain every possible t-tuple. We take advantage of this in our algorithm by only storing t-tuples that appear in the test suite. Our empirical study shows a comparison for both time and memory usage associated with our algorithm for two-way and three-way inter-window parameter interaction coverage. Further, we compare the rate of fault detection for two-way and three-way prioritization on an open source web application called SchoolMate that we seeded with 66 faults. Our results show that the rates of fault detection for two-way and three-way prioritization are within 1% of each other, but two-way provides a slightly better result. A closer look at the web application, test cases, and faults reveals that most faults are triggered by two-way interactions. (66 pages)

PUBLIC ABSTRACT

Combinatorial-Based Prioritization for User-Session-Based Test Suites

by

Schuyler Manchester, Master of Science

Software defects caused by inadequate software testing can cost billions of dollars. Further, web application defects can be costly due to the fact that most web applications handle constant user interaction. However, software testing is often under time and budget constraints. By improving the time efficiency of software testing, many of the costs associated with defects can be saved.

Current methods for web application testing can take too long to generate test suites. In addition, studies have shown that user-session-based test suites often find faults missed by other testing techniques. This project addresses this problem by utilizing existing user sessions for web application testing. The software testing method provided within this project utilizes previous knowledge about combinatorial coverage testing and improves time and computer memory efficiency by only considering test cases that exist in a user-session-based test suite. The method takes the existing test suite and prioritizes the test cases based on a specific combinatorial criterion. In addition, this project presents an empirical study examining the application of the newly proposed combinatorial prioritization algorithm on an existing web application.

ACKNOWLEDGMENTS

Most importantly, I would like to express my immense gratitude to Dr. Renée Bryce. She has been an exceptional advisor and mentor. The experience and knowledge I have gained throughout my undergraduate and graduate work with her will stay with me for years to come. Without Renée, I would not have had all of the opportunities that I have had to excel in both the world of research and industry.

I would also like to make a special thanks to Dr. Sreedevi Sampath; her help through my undergraduate and graduate research experience has been truly rewarding. It was really a great privilege to get to collaborate with her and her students in the creation of CPUT.

This work is supported by the National Institute of Standards and Technology, Information and Technology Lab Award number 70NANB10H048. Any opinions, findings, and conclusions expressed herein are the authors' and do not reflect those of the sponsors. I would like to thank undergraduate and graduate researchers Devin Minson, Sachin Jain, and Nilesh Chaturvedi for their contributions to the open-source software testing tool CPUT.

Schuyler Manchester

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Software defects caused by inadequate software testing often incur billions of dollars in unnecessary costs [31]. At the same time, software testing is often performed under limited budget constraints. Algorithms, tools, and techniques are needed for efficient and effective testing, such that testing may be completed within the limited time and budget constraints most testers face.

Studies have found that some faults occur due to interactions between parameters, and in particular due to interactions among six or fewer parameters [20]. Wallace and Kuhn [21] reviewed 15 years of recall data for medical devices gathered by the U.S. Food and Drug Administration (FDA) and studied failure data on 109 medical devices. Of these 109 cases, 97% of the reported failures could have been detected by testing all pairs of parameter settings. Pairwise testing, or two-way testing, covers interactions between two parameters and is an effective approach, but may also miss some faults [12,19,20,23,39]. For example, in the web browser software studied by Kuhn et al. [20] 76% of the faults were triggered by two-way interactions between parameters. The remaining faults were triggered by higher strength interactions (e.g., three-way through six-way) between parameters. Thus, it is often important to test higher strength interactions between parameters during software testing.

Generating higher strength interaction test suites in the smallest number of test cases is NP-hard [11]. However, several approaches generate higher strength interaction test suites with trade-offs in effectiveness of time to generate test suites, size of test suites for fixed-level or mixed-level inputs, and ability to accommodate constraints [11]. Tools such as ACTS (formerly known as FireEye) [24] generate test suites that provide higher strength coverage and accommodate constraints, and hundreds of users routinely download such tools.

More recently, we prioritized existing test suites by two-way inter-window event coverage for event-driven systems, i.e., web and GUI systems [8, 9, 33]. Test suite prioritization is a test suite management technique wherein test cases are ordered for execution based on certain criteria such that faults may be found early in the test execution cycle. Our previous work applies test prioritization to the domain of web applications and prioritizes user-session-based test cases, i.e., test cases created from usage logs of the web system [8,33]. In our empirical studies, we compare two-way inter-window parameter-value interaction coverage with several other prioritization criteria, and two-way is among the best criterion in several of our subject applications. However, given the observation from existing research on traditional software that some faults are missed by two-way interaction test suites, we decided to investigate higher strength prioritization strategies, such as three-way.

In this work, we examine a greedy algorithm to prioritize by t-way combinatorial coverage. Our previous work generates covering arrays that represent t-way interaction test suites [4–7]. The inputs that we examined in previous work to generate test suites were significantly smaller, and we focused on obtaining 100% coverage of all t-way interactions. However, in our application of test suite prioritization, particularly for user-session-based test suites for web applications, we found that the test cases are much larger in terms of the length of test cases, and they often have an incomplete coverage of t-tuples. For example, a web application may allow users to select a set of dates, but users may not try selecting every possible combination of months, days, and years available. Consider that the parameters are continuous, but also can be considered discrete (e.g., although dates are continuous, the dates entered will generally be the current date +/- two years). Utilizing this estimate yields approximately 1460 different levels or values for any date entered given the options for month, day, and year.

To generate a combinatorial interaction test suite for t-way coverage, previously proposed greedy algorithms require that we enumerate every possible t-way interaction among the parameters, store these in memory, and track the coverage of these tuples as tests are ordered. However, this is prohibitive for large test suites. In our application of user-session-based testing, we have found sparse coverage of t-tuples, as our users do not systematically cover every possible t-way combination when using the web application. Thus, our application of test suite prioritization by t-way interaction coverage poses the challenge that test cases are often much longer than inputs that we have considered in combinatorial interaction test suite generation in the past. On the other hand, this application offers the opportunity that exhaustive enumeration of every possible t-tuple is not necessary and the algorithm can simply store the t-tuples available in the user-session-based test suite.

Our contribution in this paper is two-fold: (1) an algorithm that only uses memory for valid t-tuples in a test suite, and (2) an empirical study that examines three-way inter-window parameter-value interaction coverage for test suite prioritization. The algorithm that we introduce in this paper takes advantage of the incomplete coverage of t-way interactions in order to use less memory and accommodate larger inputs. In addition, we evaluate the efficiency and fault finding effectiveness of the algorithm, using three-way interaction-based prioritization. We measure efficiency by studying the time and memory requirements of the algorithm. To measure fault finding effectiveness, we present a case study with an open source web application, SchoolMate. We gather user sessions for SchoolMate and prioritize them using our three-way algorithm. We contrast the effectiveness of three-way with the other prominent prioritization criteria (including two-way) and measure the rate of fault detection, i.e., how quickly the test order locates faults. The rate of fault detection is the most commonly used measure of effectiveness of a prioritization criterion [30]. Though we evaluate the algorithm with a case study of web applications, the algorithm can be applied when working with test suites of any software systems that have sparse t-tuple coverage. A subsection of this empirical study has been submitted to the 2012 Journal of Combinatorial Mathematics and Combinatorial Computing.

CHAPTER 2

PREVIOUS WORK - LITERATURE REVIEW

While the greedy algorithm for test suite prioritization that we present may apply to any test suite, we tailor our focus to user-session-based test suites for web applications. In this section, we review previous work in two areas: (1) web applications and user-session-based testing, and (2) test suite prioritization and combinatorial-strategies for prioritization.

2.1 Web Application Testing

Several approaches exist for generating test cases for web applications. For instance, tools such as HTTPUnit [17] and RationalRobot [28] allow testers to record and play back test sequences and to measure performance. Some tools check for broken links, validate HTML code, and measure performance. More semi-automated approaches also exist. For instance, Veriweb offers a simple solution that starts at a given URL and non-deterministically traverses links in a web application [3]. Andrews et al. use finite state machines with constraints to provide system-level tests [2]. The constraints are used to select reduced inputs to help reduce the state space growth. Kung et al. consider object relations, state, and page navigation diagrams using a web test model (WTM) [22]. In addition, the model can be useful in identifying change ripple effects and finding test strategies that will be more cost-effective during regression testing. Ricca et al. use UML models from pages of a web application in order to automatically generate test cases for white box testing that are based on static HTML links [29]. Liu et al. use data flow interactions among clients [25]. These data flow interactions form test cases for web applications based on flow graphs. Halfond et al. discover web application interfaces from server code [16]. The discovery of the web application interfaces assist in generating test inputs for web applications. Wang et al. locate interaction faults by generating test cases that cover pairwise interactions between web pages [40]. Offut et al. use HTTPUnit and HtmlUnit to run bypass tests that bypass client-side checks [26]. Qian uses a genetic algorithm utilizing crossover and mutations to generate a large volume of test cases for a test suite [27].

One test generating framework that has been used for numerous studies is the automatic efficient tests generator (AETG). One example in which this framework is used is in a study by Cohen et al. [10]. In one experiment, the authors tested user interface modules from Telcordia, a telecommunications software company. In this experiment, they tested the modules from two releases, with nine modules from the first release and 13 modules from the second release. The pairwise test sets that were generated by AETG gave over 90% block coverage. They also did a comparison of pairwise testing and random input testing and found better coverage for pairwise testing.

All of these strategies generate test cases from models of the web system. Additional work to test rich internet applications exists, but such work is outside of the scope of this thesis. We focus on a particular type of web testing that occurs during the maintenance phase of the system, user-session-based testing.

Regression testing is one of the largest maintenance costs during the software development cycle. There are two primary options for test suites used during regression testing: (1) generate test suites that fulfill a certain criterion, or (2) user-session-based test suites. User-session-based testing has the benefit that tests can be automatically constructed from web logs for use in regression testing and they contain sequences of actions that real users have performed. User-session-based test cases also have the benefit that testers do not need to specify inputs for test cases. For instance, web applications are accessible through the Internet, and each HTTP POST and GET request that a user makes is written to a log file. The logs can then be parsed into test cases by using the IP addresses, cookies, and time stamp for each POST and GET request in order to identify the steps of each user and to create the respective test cases [13,36,38]. A user session is a sequence of base requests and name-value pairs associated with the requests. Table 2.1 shows an example user session for a bookstore application.

Table 2.1. Example User-Session-Based Test Case.

User 1
index.php
showbooks.php?book_name="java for beginners"&book_type="programming"
buybooks.php?book_id="7"
shippingMethod.php?carrier="ups"&type="ground"

Existing work on user-session-based testing falls under the categories of test case generation, test suite prioritization, test suite reduction, and test suite repair. Elbaum et al. conducted empirical studies and showed that user-session-based testing is a good option to augment white box testing techniques as they found different faults [13]. Sampath et al. [34] and Sprenkle et al. [38] also present a framework for user-session-based testing of web systems. The test case creation heuristics presented in their work is leveraged in this paper but is extended as we parse Apache web server logs and provide an XML format for test cases [32]. While there are advantages to user-session-based testing, two major problems arise over time: (1) user-sessions may become invalid during regression testing (i.e., the structure of the web application changes, including page names, links, options on a page, etc.), and (2) a large number of user-sessions build up, making it unrealistic to run all tests in practice. Alshahwan et al. present work on the first issue of repairing user-session-based test cases for use in regression testing [1]. Two approaches have been taken to address the second issue of managing large test suites, that of test suite prioritization [8, 33] and reduction [35, 36]. In the current work, we focus on test suite prioritization. In the next subsection, we provide an overview on test case prioritization and elaborate on our prior work in the area of test suite prioritization for user-session-based test cases.

2.2 Test Case Prioritization

Exhaustive testing of software is almost always an unachievable goal to obtain. This dilemma has brought software testing to a more attainable goal by using covering arrays with combinatorial testing. This process requires substantially less time and resources. This approach to software testing was first used by Cohen et al. [10] wherein they generated test suites that cover pairwise up to t-way combinations. One empirical study by Bryce and Memon [9] showed better fault finding effectiveness for test suites with the largest event interaction coverage when prioritizing by two-way and three-way interaction coverage.

Test suite prioritization is formally defined by [30]. Given T, a test suite, Π, the set of all test suites obtained by permuting the tests of T, and f, a function from Π to the set of real numbers, the problem is to find $\pi \in \Pi$ such that $\forall \pi' \in \Pi, f(\pi) \geq f(\pi')$. In this definition, Π refers to the possible prioritizations of T, and f is a function that is applied to evaluate the orderings.

Xie et al. examined the characteristics of a good GUI test suite [41]. The authors found there are two primary characteristics that increase the rate of fault detection: (1) diversity of states in which an event executes, and (2) the event coverage of a test suite. Several criteria have been applied for test suite prioritization to user-session-based test suites [14,30]. Previous work by Bryce et al. [9] examines two-way and three-way inter-window event coverage for test suite prioritization on GUI applications. For each application, they applied two-way and three-way inter-window event coverage, unique event coverage, length of test cases (longest to shortest and shortest to longest in terms of the number of parameter-values) and random ordering. The first application, a calculator, only had two windows, so with the exception of three-way, each technique was applied. The results show that two-way provides the best rate of fault detection. In the other three applications, there were three or more windows, so the authors were able to apply all of the prioritization criteria. For a paint program, choosing the longest tests first resulted in the best rate of fault detection, followed by three-way and finally two-way. For a spreadsheet program, unique event coverage provided the best rate of fault detection in the first half of the test suite, but then two-way and three-way alternate in providing the overall best rate of fault detection in the latter half of the test suite. Finally, in a word processing application example, two-way and three-way alternate in producing the best rate of fault detection. This work provides some motivation for us to explore the application of three-way inter-window parameter-value interaction coverage in the domain of web applications.

Bryce et al. [8] examine several prioritization criteria, including the combinatorial criterion, pairwise inter-window parameter-value interaction coverage (two-way), applied to user-session-based test suites and empirically evaluate them on three web applications, including an online bookstore, a course project manager, and a conference management system. All three applications were seeded with faults. The authors found that prioritization criteria based on the *longest tests with respect to the number of POST/GET requests*, *longest tests with respect to the number of parameters that users assigned values* and *two-way combinatorial coverage of inter-window interactions* were usually efficient techniques compared to the original order in which test cases were logged or ordered at random. Figure 2.1 shows the rate of fault detection for these four prioritization techniques applied to course project manager (CPM), a test suite for web applications. In this example, prioritization by length (gets/posts), number of parameters, and two-way combinatorial coverage provide a faster rate of fault detection than random ordering. We provide this example here because CPM was the largest web application with the largest test suite that was studied in our previous work (9,401 lines of code, 890 tests) [8]. (We refer the reader to [8] for further details and case studies.)

However, since existing literature recognizes that certain faults are detected by interactions between parameters that are stronger than pairwise interactions, we are interested in investigating this hypothesis for web applications and user-session-based testing. In the remainder of this paper, we propose an algorithm for n-way combinatorial interaction coverage and present an empirical evaluation of the efficiency and effectiveness of the algorithm up to t=3.

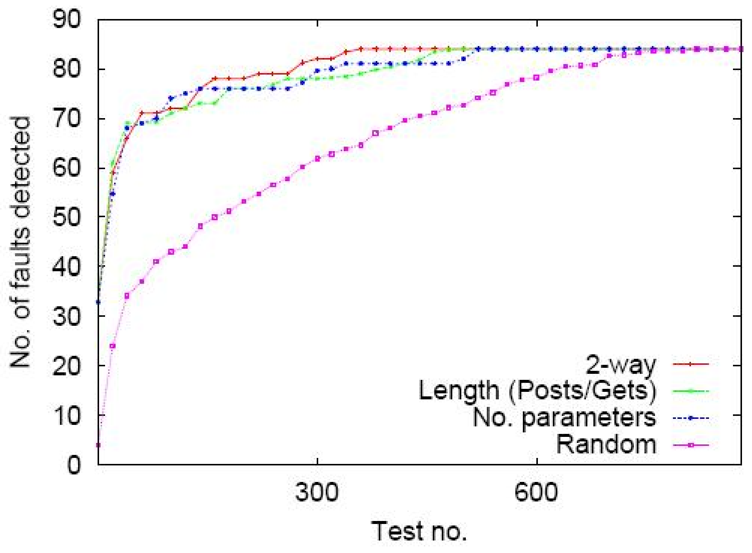


Figure 2.1. Rate of fault detection for the CPM web application.

CHAPTER 3
ALGORITHM

Our previous work focuses on generating covering arrays for combinatorial interaction testing (CIT) [4–7]. In such test suite generation, we generate a covering array in which all t-tuples, with the exception of constraints, are covered at least once. However, in our work on test suite prioritization, we noticed that the user-session-based test suites did not contain all possible t-tuples [8]. For instance, we examined three web applications and found that the associated user-session-based test suites did not contain an exhaustive collection of all t-tuples of parameter-values between windows. This makes sense because our web applications had many fields in which different users could manually enter personal data such as user ids, passwords, mailing addresses, and other textual data. This observation led us to design an algorithm that does not identify all parameter-values in a system nor enumerate the possible t-tuples, but rather only stores the t-tuples that appear in the test suite.

In this section, we illustrate t-way test suite prioritization for $t = 2$ and $t = 3$ and then present the prioritization algorithm.

3.1 Example of t-way Prioritization

Consider an e-commerce application wherein users purchase and ship items. Table 3.1 defines four pages of the web application. In the first page, the user may select one of three options for the *shipment time*. The user may then select one of three options for the *postal carrier* on the second page. On the third page, the user may specify one of three options for *tracking*. Finally, the user may choose one of three options for *insurance* on the fourth page. Selecting different options will execute different lines of code in the system. For instance, if the user selects any *tracking* option other than “None”, the system generates a unique

Table 3.1. Web Testing Example with Four Factors and Three Levels for Each Factors.

Page 1, Shipment Time	**Page 2, Carrier**	**Page 3, Tracking**	**Page 4, Insurance**
5 - 10 days	USPS	Status tracking (Stat)	up to \$100
1 - 3 days	UPS	Signature confirmation (Sig)	up to \$1000
overnight	Fedex	None	None

tracking identifier and directs the user to a separate page that describes the conditions of the desired tracking. Thus, having test coverage for the different values for tracking could potentially uncover a fault that might have been overlooked by a different test.

Table 3.2 contains an example test suite that accesses the pages. This test suite contains a total of four different test cases and covers a total of twenty unique 2-tuples and fifteen unique 3-tuples as shown in Table 3.3. The tuples represent the

inter-window parameter-value interactions in the test case.

When prioritizing by two-way, we select the first test case such that it covers the largest number of 2-tuples. The second column of Table 3.3 shows that all four test cases cover six 2-tuples. We then break the tie at random, select t_2, and mark the 2-tuples in this test as covered. We next examine which of the remaining tests cover the most remaining uncovered 2-tuples. Test case t_1 covers three uncovered 2-tuples, t_3 covers six new uncovered 2-tuples, and t_4 covers five uncovered 2-tuples, so we choose t_3. We mark the 2-tuples in t_3 as covered and examine the last two remaining tests to select the test that covers the most uncovered 2-tuples. We select test t_4 as it covers the most uncovered 2-tuples. The ordering for two-way prioritization is then $\{t_2,t_3,t_4,t_1\}$.

To prioritize by three-way, we select the first test case that covers the most 3-tuples. All four test cases cover four 3-tuples, so we break the tie at random and select t_3. We then mark the 3-tuples covered in test t_3 as covered. In the next iteration, there is a tie among all of the tests as they cover the same number of 3-tuples, so we break the tie at random and select t_2. In the next iteration, t_4 covers more uncovered 3-tuples than t_1, so we select t_4. Finally, we add the last test case, t_1 to the test suite. The final ordering by three-way is $\{t_3,t_2,t_4,t_1\}$.

Table 3.2. Test Suite Example of Table 3.1 Web Testing Example.

Test case	**Ship Time**	**Carrier**	**Track**	**Insur**	**Parameter-Values (P-V) Covered**
t_1	5-10 days	USPS	None	\$100	Ship Time:5-10 days, Carrier:USPS, Track.:None Insur:\$100
t_2	1-3 days	USPS	None	\$100	Ship Time:1-3 days, Carrier:USPS, Track.:None Insur:\$100
t_3	overnight	UPS	Sig.	\$1000	Ship Time:overnight, Carrier:UPS, Track.:Sig. Insur:\$1000
t_4	1-3 days	Fedex	Stat.	\$100	Ship Time:1-3 days, Carrier:Fedex, Track.:Stat. Insur:\$100

Table 3.3. Two-way and Three-way P-V Covered from Test Suite Described in Table 3.2.

Test case	**Two-way Tuples Covered**	**Three-way Tuples Covered**
t_1	(Ship Time:5-10 days, Carrier:USPS), (Ship Time:5-10 days, Track.:None), (Ship Time:5-10 days, Insur:\$100), (Carrier:USPS, Track.:None), (Carrier:USPS, Insur:\$100), (Track.:None, Insur:\$100)	(Ship Time:5-10 days, Carrier:USPS, Track.:None), (Ship Time:5-10 days, Carrier:USPS, Insur:\$100), (Ship Time:5-10 days, Track.:None, Insur:\$100), (Carrier:USPS, Track.:None, Insur:\$100)
t_2	(Ship Time:1-3 days, Carrier:USPS), (Ship Time:1-3 days,Track.:None), (Ship Time:1-3 days, Insur:\$100) (Carrier:USPS, Track.:None) (Carrier:USPS, Insur:\$100) (Track.:None, Insur:\$100)	(Ship Time:1-3 days, Carrier:USPS, Track.:None), (Ship Time:1-3 days, Carrier:USPS, Insur:\$100), (Ship Time:1-3 days, Track.:None, Insur.:\$100), (Carrier:USPS, Track.:None, Insur:\$100)
t_3	(Ship Time:overnight,Carrier:UPS), (Ship Time:overnight, Track.:Sig.), (Ship Time:overnight, Insur:\$1000) (Carrier:UPS, Track.:Sig.) (Carrier:UPS, Insur:\$1000) (Track.:Sig., Insur:\$1000)	(Ship Time:overnight, Carrier:UPS, Track.:Sig.), (Ship Time:overnight, Carrier:UPS, Insur:\$1000), (Ship Time:overnight, Track.:Sig., Insur:\$1000), (Carrier:UPS, Track.:Sig., Insur:\$1000)
t_4	(Ship Time:1-3 days, Carrier:Fedex), (Ship Time:1-3 days,Track.:Stat.), (Ship Time:1-3 days, Insur:\$100) (Carrier:Fedex, Track.:Stat.) (Carrier:Fedex, Insur:\$100) (Track.:Stat., Insur:\$100)	(Ship Time:1-3 days, Carrier:Fedex, Track.:Stat.), (Ship Time:1-3 days, Carrier:Fedex, Insur:\$100), (Ship Time:1-3 days, Track.:Stat., Insur:\$100), (Carrier:Fedex, Track.:Stat., Insur:\$100)

3.2 Algorithm for t-tuple Prioritization

Phase 1: Preprocessing. We iterate through each test cases t_i in the test suite TS. For each URL, we identify each parameter p that has been assigned a value v on each page. We refer to the assignment of a value to a parameter as a parameter-value. We then create a list of all t-way inter-window parameter-value interactions (t-tuples) in the test suite and store them in our $tuplesList$. This preprocessing stage allows us to only store tuples in memory that are contained in the test suite.

Phase 2: t-way Prioritization. The greedy algorithm selects the test case that has the largest count of uncovered t-tuples from the $tuplesList$, marks those t-tuples as covered (i.e., removes them from the $tuplesList$), and repeats this process until the entire test suite has been prioritized. In each iteration, for each test case, the $tCountMax$ is computed as the number of uncovered t-tuples that are in the test case. The test case with the highest $tCountMax$ is added to the test suite, and the t-tuples that are covered in this test case are removed from the $tuplesList$ that stores the uncovered t-tuples. Figure 3.1 provides pseudocode for this algorithm.

In the next section, we present our empirical study that shows the scalability of the algorithm for $t = 3$ and the effectiveness of the prioritized test orders.

```
// Preprocessing of test suite
$sizeOfTestSuite = 0$
foreach test case $t_i$ in the test suite $TS$
  foreach URL $url$ in $t_i$
    foreach param $p$ assigned a value $v$ in $t_i$
      tuple $tu = url + p + v$
      if $tuplesList$ does not contain $tu$
        $tuplesList$ add $tu$
    end foreach
  end foreach
  $sizeOfTestSuite$++
end foreach

// Generate all t-tuple combinations and insert into t-tuplesList
$t$-wayTuplesList = generate-t-wayTupleList($tuplesList$)

// Test Suite Prioritization
$test_{bestTest}$ = select a test that covers the most unique $t$-way tuples from $t$-wayTuplesList
add $test_{bestTest}$ to prioritized test suite $TS_p$
mark $test_{bestTest}$ as used
remove the $t$-way tuples that appear in $test_{bestTest}$ from $t$-wayTuplesList
selectedTestCount = 1
while(selectedTestCount < $sizeOfTestSuite$)
  $tCountMax$ = -1
  for j=1 to ($sizeOfTestSuite$-$selectedTestCount$)
    if $test_j$ is not used
      compute $tCount$ as the number of newly covered $t$-way tuples from $t$-wayTuplesList in $test_j$
      if($tCount > tCountMax$)
        $tCountMax = tCount$
        $test_{bestTest}$ = j
      else if($tCount == tCountMax$)
        break the tie at random
  end for
  add $test_{bestTest}$ to $TS_p$
  mark $test_{bestTest}$ as used
  remove the $t$-way tuples that appear in $test_{bestTest}$ from $t$-wayTuplesList
  $selectedTestCount$++
end while
```

Figure 3.1. Algorithm for test suite prioritization by t-way combinatorial coverage.

CHAPTER 4
EXPERIMENT

The two main lines of inquiry in our study are:

1. How well does the *t*-way prioritization algorithm scale, for t=3, for user-session-based test suites?

2. How effective are the prioritized test orders generated by two-way and three-way at detecting faults?

4.1 Subject Application

Our subject application is an open-source web-based application called SchoolMate. It is written in PHP with a MySQL backend. SchoolMate is designed for elementary, middle, and high schools to manage classes, registration, assignments, and grades. There are four different types of users that can log into this application: (1) admin (2) teacher (3) parent and (4) student. The parent and student can access identical web pages, except that a parent also has a list of assigned children. Therefore, we refer to this functionality as parent/student in the remainder of the paper. Table 4.1 describes the technical characteristics of the application, information about the test suite, and seeded faults.

4.2 Test Suites

The test suites for this study were gathered by an undergraduate software testing class. The class was instructed to login to the web application and test out as many web pages as possible. The test cases are constructed using the IP addresses that are associated with each GET/POST request. If there is more than a 45-minute break in between a GET/POST request from the same user, we begin a new test case. We initially collected a large test suite with 125 test cases, but later broke it down into three smaller test suites where each

Table 4.1. Composition of the Web-based Application and Test Suite in Our Study.

Description	**SchoolMate**
Files	63
Unique parameter-values in test suite	2,611
Lines of Code (LOC)	6652
PHP methods	15
JavaScript methods	70
Branches	618
Total no. of tests	125
Total no. of tests with Administrator users	59
Total no. of tests with Student users	44
Total no. of tests with Teacher users	55
Largest count of GETS/POSTS in a test case	193
Average count of GETS/POSTS in a test case	36.5
Largest count of parameters in a test case	874
Average count of parameters in a test case	163.04
Two-way parameter-value interactions covered in test suite	278,109
Three-way parameter-value interactions covered in test suite	477,450
No. of seeded faults	66

test suite contains tests for either an administrator, teacher, or parent/student user. While there is overlap in the code that some of the user types access, we split the test suites based on user types because different user types are required in order to access certain parts of the code and seeded faults. Table 4.1 shows that the three test suites have between 44 to 59 test cases. See Appendix A for a more detailed sample test case. In addition, see Appendix B for details on the percent code covered in SchoolMate for each test case within the test suite.

4.3 Faults

A total of 66 faults were seeded into SchoolMate by a graduate student. Each seeded fault version belongs to two categories, user type and fault classification. We seeded faults for each type of user of the system, i.e., admin, teacher, parent/student, and any. Sampath et al. [36] and Guo et al. [15] presented a fault classification for web applications. We use this classification (described below) when seeding faults in SchoolMate.

- Appearance faults: Faults in the application code that change the display of a web page. An example is that a missing print statement in the PHP code can sometimes cause the code to display in an incorrect manner.

- Link faults: Faults in the application code that manipulate the page pointed to by a URL. An example is a link that points to a non-existent page causing an error to display.

- Data Store faults: Faults in the code that modify data storage within the application. An example includes swapping variables for an SQL Insert query which causes the data to be stored improperly.

- Form faults: Faults in the application code that manipulate a form's name-value pairs. An example includes swapped variables for the text and values for an HTML option list which causes incorrect text to be displayed, and incorrect values to be sent to the server.

- Logic faults: Faults in the application code that manipulate control flow and/or business logic. An example is displaying an improper date format that causes the date to be saved incorrectly.

Logic faults have seven subcategories, of which we use five, because the remaining categories were not applicable for our subject application, SchoolMate. The five subcategories we used are:

- Session faults: Faults in the application code that manipulate the current session state of the application or faults that manipulate other session-based operations such as using sessions to save information entered on a form and display the information after the sessions have been validated. An example is accidentally setting a variable that determines what menu navigation screen is displayed; this causes undesired behavior.

- Paging faults: Faults in the application code that manipulate the display of large amounts of data. An example is using a '<' instead of a '<=' when iterating through the pages of users, which causes the last page of users to never be displayed.

- Server-side parsing faults: Faults in the application code that change server-side parsing of data. An example is an escaped variable that causes the variable name to be saved instead of the value assigned to that variable.

- Encoding/decoding faults: Faults in the application code that encode or decode information during transmission, storage, and/or display. An example is a missing convert from a database function that causes the data to not be decoded into a more readable format.

- Locale faults: Faults that exist in code that manipulate locale-specific information within the application, such as date format or language. An example is that saving the date format in European date encoding causes the date to be saved incorrectly for a U.S. application.

See Appendix C for details on the fault type distribution.

4.4 Prioritization Criteria

We use five prioritization criteria. First, we use two-way and three-way inter-window parameter-value interaction coverage in order to evaluate whether three-way offers improvement over two-way in fault finding effectiveness. Second, we use two length-based criteria, namely, number of GET/POST requests in a test case and the number of parameter-values in a test case. We choose these length-based criteria as they performed well in our previous work [8]. Random ordering is used as a control.

- **Two-way.** We select the test cases in non-ascending order of the number of inter-window parameter-value combinations between two separate pages. Ties are broken at random.

- **Three-way.** We select the test cases in non-ascending order of the number of inter-window parameter-value combinations between three separate pages. Ties are broken at random.

- **Length (GETS/POSTS).** We select the test cases in non-ascending order of the number of GET/POST requests. Ties are broken at random.

- **Number of parameter-values.** We select the test cases in non-ascending order of the number of parameter-values. Ties are broken at random.

- **Random.** We use the random function available in Java to randomly swap the ordering of the test cases. The tool produces a different random ordering each time the user chooses to prioritize at random.

4.5 Experimental Framework

The usage logs for SchoolMate are converted into test cases and prioritized within our tool, CPUT [32]. The t-way prioritization algorithm is implemented in CPUT for $t = 2$ and $t = 3$, in addition to other criteria, such as length, randomness and frequency-based prioritization criteria. Apache logs are parsed and XML format test cases are created. The test cases can then be prioritized using the different prioritization criteria. Figure 6.3 shows a screenshot of CPUT with the prioritized test order displayed in the bottom-left window pane, and the test case displayed in the bottom-right pane.

We next execute the test cases using a replay tool. We created a new replay tool that can execute the XML format test cases. We conducted fault detection experiments using the framework presented by Sprenkle et al. [38]. Initially, we execute the test cases on a clean version of the application and save the returned files. This is the expected output, since we consider the non-fault-seeded version of the application as our gold standard. Then, one fault is seeded in the application at a time, and all the test cases are executed. The returned HTML files are saved (this is the actual output). The test oracle is then executed on the returned files to determine if the test case detects the fault. We present the results from the *struct* oracle here. The *struct* oracle [38] compares the expected and actual output in terms of the HTML tags in the files to identify differences. A fault matrix is generated that shows how many faults and which faults are detected by each test case.

CHAPTER 5
RESULTS

We first review the scalability of our algorithm on several web logs for the SchoolMate application and next present our findings on the fault-finding effectiveness of prioritization by three-way inter-window parameter-value interaction coverage for our application and test suites.

5.1 Scalability

To study the scalability potential of our algorithm, we record each component's execution time and space requirements of the output. The experiment was run on a machine with a Windows 7 operating system with 6GB of RAM, and with an i7 processor with 1.6 GHz per core. In this study, we split the log into 1 day usage, 5 days usage, 10 days usage, 15 days usage (which is the entire log file), and doubled the size of the log file. To double the size of the log file, we modified the log file by changing the year from 2010 to 2011 for the date. We then replaced digits in the IP addresses and cookies to make them different. For instance, we swapped occurrences of the number 4 with the number 3.

Table 5.1 summarizes the results. We present the results for each log file separately. For each log file, we present the time taken by the test case creation engine and the different prioritization and reduction criteria (column 4). We also present the space occupied by the output of the different components of the framework (column 5). We note that the test creation engine takes from a few seconds to slightly over one minute for the double log file. The time taken results from the tool parsing the web log into test cases and storing the data from the test suite using the pre-processing part of our algorithm that is run before the test suite prioritization options are available to the user. Once the web log parsing and pre-processing are complete, the user may choose the prioritization technique

Table 5.1. Execution Time and Size of Test Suites for SchoolMate Logs.

Log file	**Component name**	**Component output**	**Execution time**	**Output Space requirements**
1 day	Test case creation engine	XML format tests	3.831	99 kb (10 test cases)
	Test prioritization (Length)	Order file	0	138 bytes
	Test prioritization (No. of params)	Order file	0	138 bytes
	Test prioritization (two-way)	Order file	0.359	138 bytes
	Test prioritization (three-way)	Order file	0.701	138 bytes
5 days	Test case creation engine	XML format tests	7.196	488 kb (50 test cases)
	Test prioritization (Length)	Order file	0	658 bytes
	Test prioritization (No. of params)	Order file	0.001	658 bytes
	Test prioritization (two-way)	Order file	1.403	658 bytes
	Test prioritization (three-way)	Order file	2.118	658 bytes
10 days	Test case creation engine	XML format tests	20.749	1326 kb (110 test cases)
	Test prioritization (Length)	Order file	0	1438 bytes
	Test prioritization (No. of params)	Order file	0.001	1438 bytes
	Test prioritization (two-way)	Order file	3.242	1438 bytes
	Test prioritization (three-way)	Order file	6.778	1438 bytes
15 days	Test case creation engine	XML format tests	26.899	1888 kb (173 test cases)
	Test prioritization (Length)	Order file	0.001	2257 bytes
	Test prioritization (No. of params)	Order file	0.002	2257 bytes
	Test prioritization (two-way)	Order file	5.248	2257 bytes
	Test prioritization (three-way)	Order file	10.292	2257 bytes
double	Test case creation engine	XML format tests	70.552	3776 kb (348 test cases)
	Test prioritization (Length)	Order file	0.004	4532 bytes
	Test prioritization (No. of params)	Order file	0.002	4532 bytes
	Test prioritization (two-way)	Order file	11.662	4532 bytes
	Test prioritization (three-way)	Order file	25.406	4532 bytes

to run. Prioritization by the length and number of POST/GET requests generally takes negligible time, reported as 0 seconds, up to 0.004 seconds. Prioritization by the number of parameters-values in a test is also negligible, taking 0 to .002 seconds. The two-way prioritization criteria are also in the order of seconds—starting at 0.359 seconds and going up to 11.662 seconds. The three-way prioritization criteria take .701 seconds, up to 25.406 seconds.

The output of the test case creation engine is the test suite. We see that tests created vary from 10 to 348 tests, occupying from a few bytes to 3,776 kilobytes. The output of the different prioritization criteria is stored in the order file which contains the test case names printed to file. The size of this output file also ranges from a few bytes to 4 kilobytes.

From these results, we note that the time taken by the *t*-tuple prioritization algorithm is in the order of a few seconds. Therefore, our algorithm has the potential to scale to larger usage logs and test cases on which the test prioritization criteria need to be applied.

5.2 Rate of Fault Detection

We prioritized our three test suites by the prioritization criteria described above and measured the rate of fault detection with the metric, average percentage of fault detection (APFD) [30]. APFD measures the area under the curve that plots test suite fraction and the number of faults detected by the test ordering. Rothermel et al. [30] define APFD as follows: 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

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + ... + TF_m}{mn} + \frac{1}{2n} \tag{5.1}$$

Intuitively, APFD measures the area under the curve that plots test case size on the x-axis and percent of unique faults detected on the y-axis. APFD is a measure of how quickly faults are detected by the test suite, which is essential in regression testing scenarios.

Out of the total 66 faults seeded, the parent/student test suite with 44 test cases detected 33 faults, the admin test suite with 59 test cases detected 50 faults, and the teacher test suite with 55 test cases detected 39 faults. The full test suite detects 51 of the 66 faults. However in this section, we present our results with the test suites divided by the user-type. We divide the test suite into three smaller test suites based on the user type as the three different user types could access different parts of code and we did not want one user type to dominate over others.

We ran our algorithm five times for each combinatorial prioritization criterion and report the average of these runs in Table 5.2. For all three test suites, two-way and three-way provide the best APFD and are within 1% of each other, with two-way performing slightly better. Prioritization by length of GET/POST requests and the number of parameter-values in a test case alternate in providing the third and fourth best APFD for the admin and teacher test suites; however, random ordering provides the third best APFD for the parent/student test suite. The random ordering is the least effective for the admin and teacher test suites. The results among two-way, length by GET/POST requests, number

Table 5.2. Average APFD of the different test orders.

% of test suite	**Admin two-way**	**three-way**	**GET/POST**	**P-Vs**	**Random**
10%	70.44	70.42	70.13	70.13	63.13
20%	86.89	73.09	71.48	72.75	73.32
30%	86.89	85.43	74.86	77.12	75.83
40%	88.76	85.43	83.68	84.04	78.98
50%	89.63	87.98	85.25	86.52	79.63
60%	89.63	87.98	85.98	86.52	80.56
70%	89.63	87.98	85.98	86.52	81.91
80%	89.63	87.98	85.98	86.52	82.07
90%	89.63	87.98	85.98	86.52	82.77
100%	89.75	88.01	86	86.65	82.9
% of test suite	**Parent/Student two-way**	**three-way**	**GET/POST**	**P-Vs**	**Random**
10%	64.6	64.6	63.57	63.77	58.77
20%	64.6	64.6	63.57	63.77	63.62
30%	64.6	64.6	63.57	63.77	64.31
40%	64.6	64.6	65.39	65.67	64.94
50%	67.13	66.22	65.39	65.67	65.6
60%	67.13	66.96	65.39	65.67	65.88
70%	67.13	66.96	65.86	66.27	66.29
80%	67.13	66.96	65.86	66.27	66.45
90%	67.13	66.96	65.86	66.27	66.66
100%	67.3	67.2	66.1	66.51	66.66
% of test suite	**Teacher two-way**	**three-way**	**GET/POST**	**P-Vs**	**Random**
10%	68.94	68.91	64.24	67.02	47.32
20%	68.94	68.91	66.91	67.02	56.53
30%	68.94	68.91	68.06	68.17	61.48
40%	68.94	68.91	68.06	68.17	62.47
50%	68.94	68.91	68.06	68.17	64.1
60%	68.94	68.91	68.06	68.17	65.01
70%	70.39	68.91	69.08	69.19	65.43
80%	71.02	69.71	70.11	69.19	66.13
90%	71.64	71.03	70.73	70.51	66.64
100%	71.64	71.03	70.73	70.51	66.72

of parameter-values, and random ordering are consistent with previous literature [8], but the results for three-way rank first within the domain of user-session-based testing for web applications.

We examined the faults detected by the two-way and three-way test suites to understand our results. We noticed that we seeded faults in our system based on the existing fault classification (presented in Section 4.3) without regard to whether they were caused by three-way interactions between parameters. An example of a web application fault that could be triggered by three-way inter-window interaction between parameters is as follows: First add a new teacher, next navigate to a page where classes are managed, and finally edit an existing class to assign the newly added teacher to the class. The fault could be introduced when assigning the teacher to an existing class. The new teacher could be assigned a substitute teacher designation instead of being noted as main teacher. This fault thus occurs due to interactions on parameters from three different windows.

An example of a web application fault triggered by two-way inter-window parameter-value interactions is as follows: A user creates a new semester, the application sets a hidden parameter "addsemester" to the value "4" (new semester id). After this new semester is successfully saved, a fault is introduced in the web page, by replacing all of the other semester's hidden ids to "4" (new semester id). The user then attempts to delete a previously existing semester, which is when the fault actually appears. The attempted delete should have the parameter "deletesemester" set to the value "1" (the previously existing semester's id); however, the fault has every semester id value set to "4". Thus, the interaction of these two parameter-values on different pages causes a fault when attempting a deletion of a semester immediately after adding a new semester.

We also note that some failures were observable only when the user session contained a certain sequence of URLs, where sequences of size two or size three caused the failure to be observed in the test case. For example the fault is introduced by a typo in the database insertion statement when a new username is created. This causes the username to save with an extra character appended to it. The fault is observed only when the user navigates to the page where she/he can view all users of the system. However, the fault itself is not caused by interactions of parameters across multiple pages. The fact that there were no faults seeded in the system that would have been expressly identified by the three-way test suites could explain why two-way and three-way test orders performed equivalently. However, in SchoolMate, there are few opportunities for such three-way interaction faults. Nonetheless, in the future, we will examine SchoolMate and other applications and seed

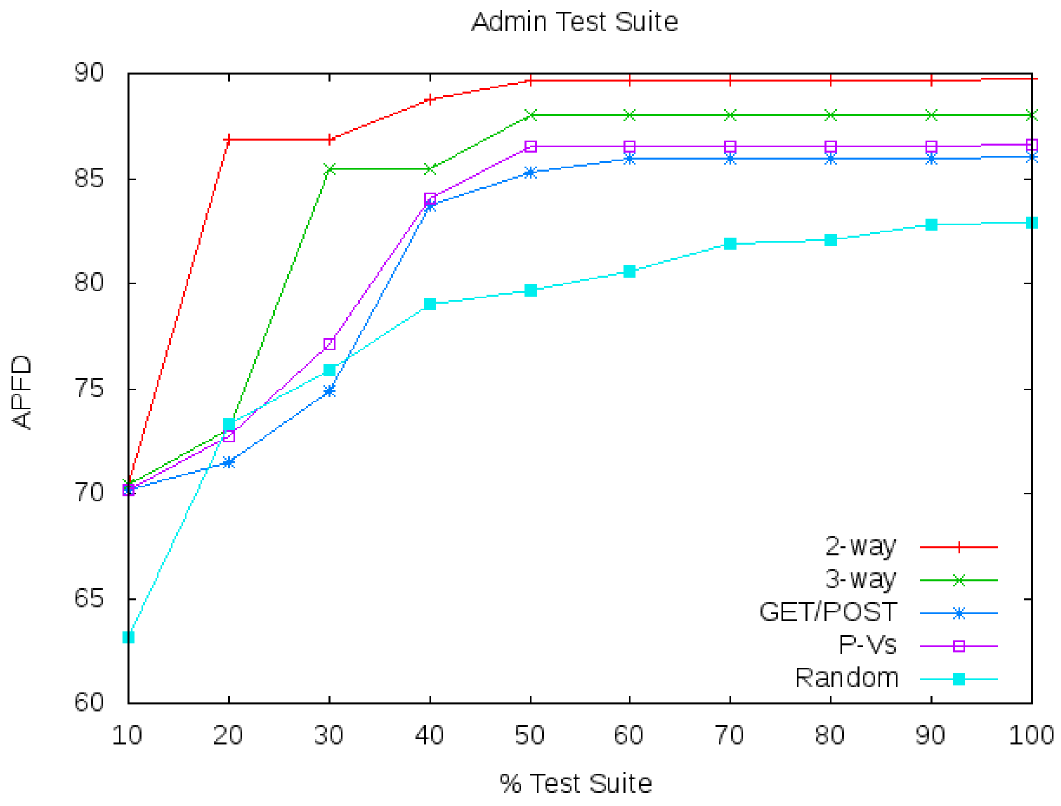


Figure 5.1. APFD for admin test suite of SchoolMate web application.

faults that capture three-way interactions between parameters and evaluate the test orders to see if the three-way orderings are better at detecting these faults than the two-way orderings. Figures 5.1, 5.2, and 5.3 show that both two-way and three-way combinatorial prioritization were within a fraction of a percent for APFD, with the exception for the admin test suite.

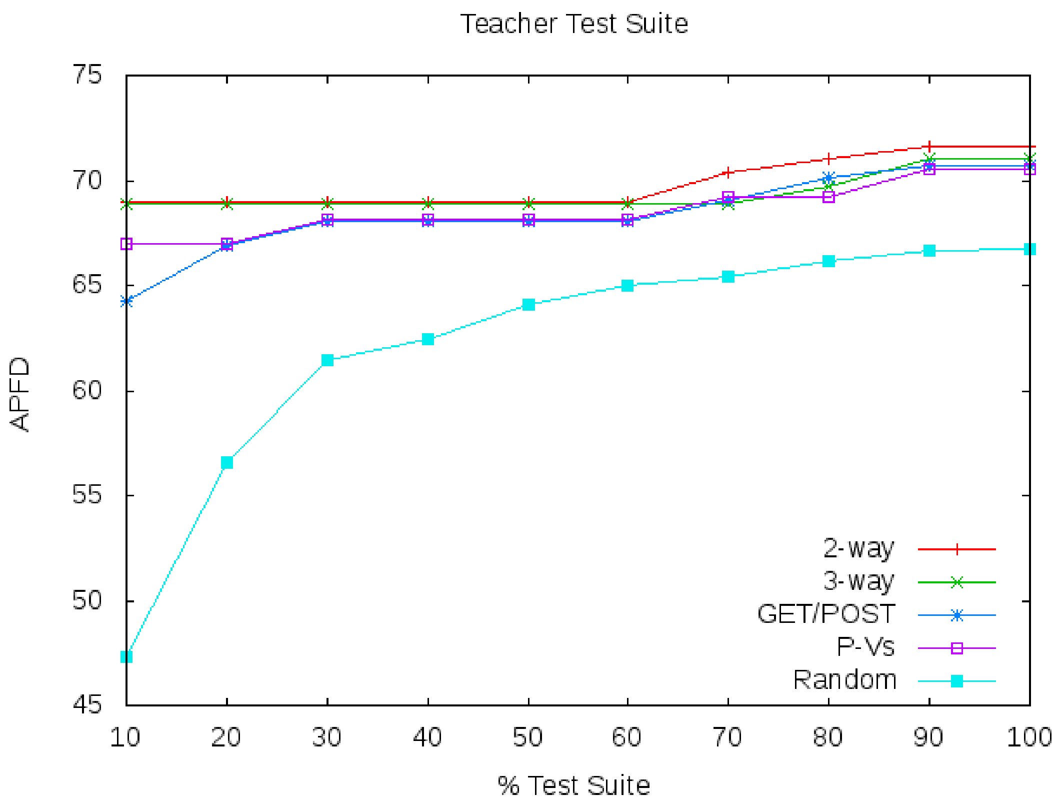


Figure 5.2. APFD for teacher test suite of SchoolMate web application.

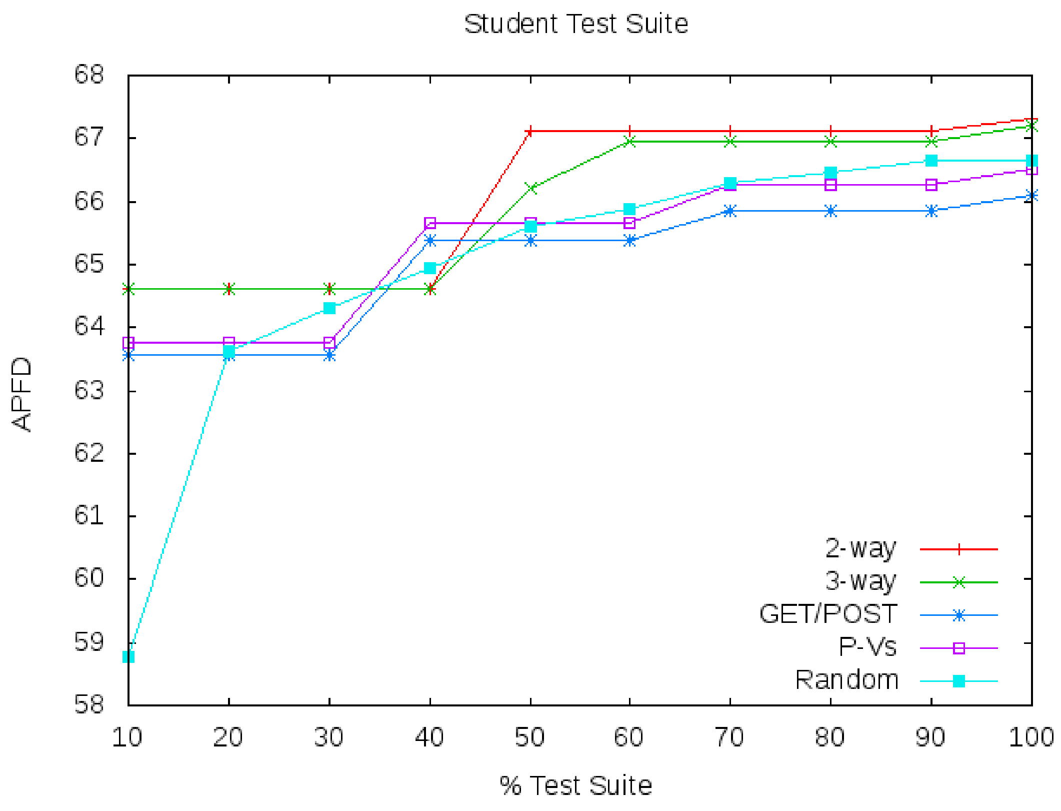


Figure 5.3. APFD for student test suite of SchoolMate web application.

CHAPTER 6
CPUT

In addition to the research contributions found in the algorithms presented in Chapter 3, the experiments discussed in Chapter 4, and the results presented in Chapter 5; this project also contributes to the open-source software testing tool **C**ombinatorial-based **P**rioritization for **U**ser-Session-Based **T**esting (CPUT), thus making three significant contributions to the field of software testing. These are listed below. In particular, my contribution of the prioritization functionality makes my research accessible to others.

> **New logger for Apache:** We developed our logger for the most popular open-source web server. Previous tools provided loggers for Resin, which has a substantially smaller user population.
>
> **Conversion of web usage logs into an XML test case format:** We generate tests in an XML format that can easily be rerun by replay tools that can parse XML data. Previous tools generate test files that have sequences of URLs and parameter-value pairs.
>
> **Prioritization criteria:** Prior empirical studies have shown that black-box test prioritization of length-based and combinatorial-based criteria create effective test order [8]. Previous tools have not implemented these criteria for web-based applications.

Figure 6.1 gives an overview of CPUT functions. The logger for Apache is implemented as a module in C. We implemented the logger such that it easily integrates with other modules of Apache. The remaining components of CPUT – the test case creation engine, the prioritization engine, and the user interface – are implemented in Java. A user of the tool first deploys the module in Apache to enable the logging of user sessions. The user

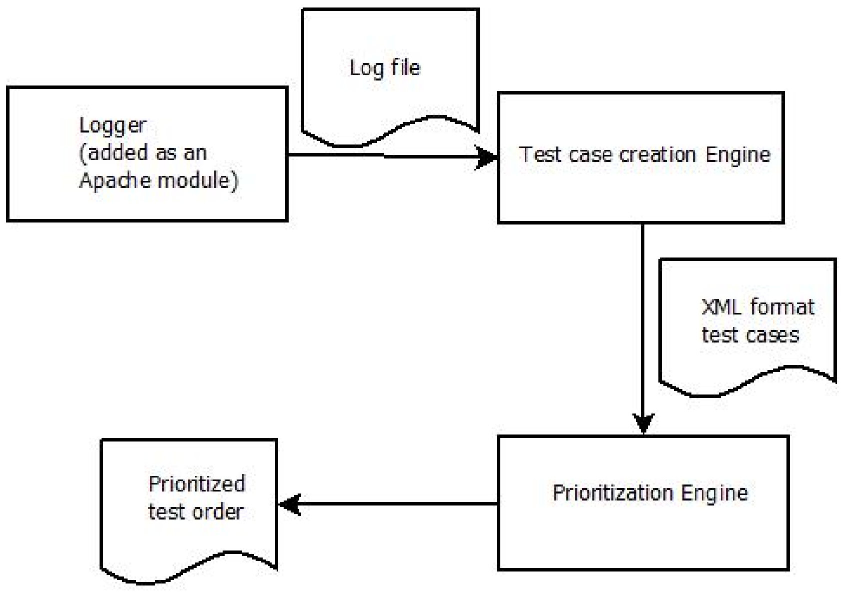


Figure 6.1. CPUT: Overview.

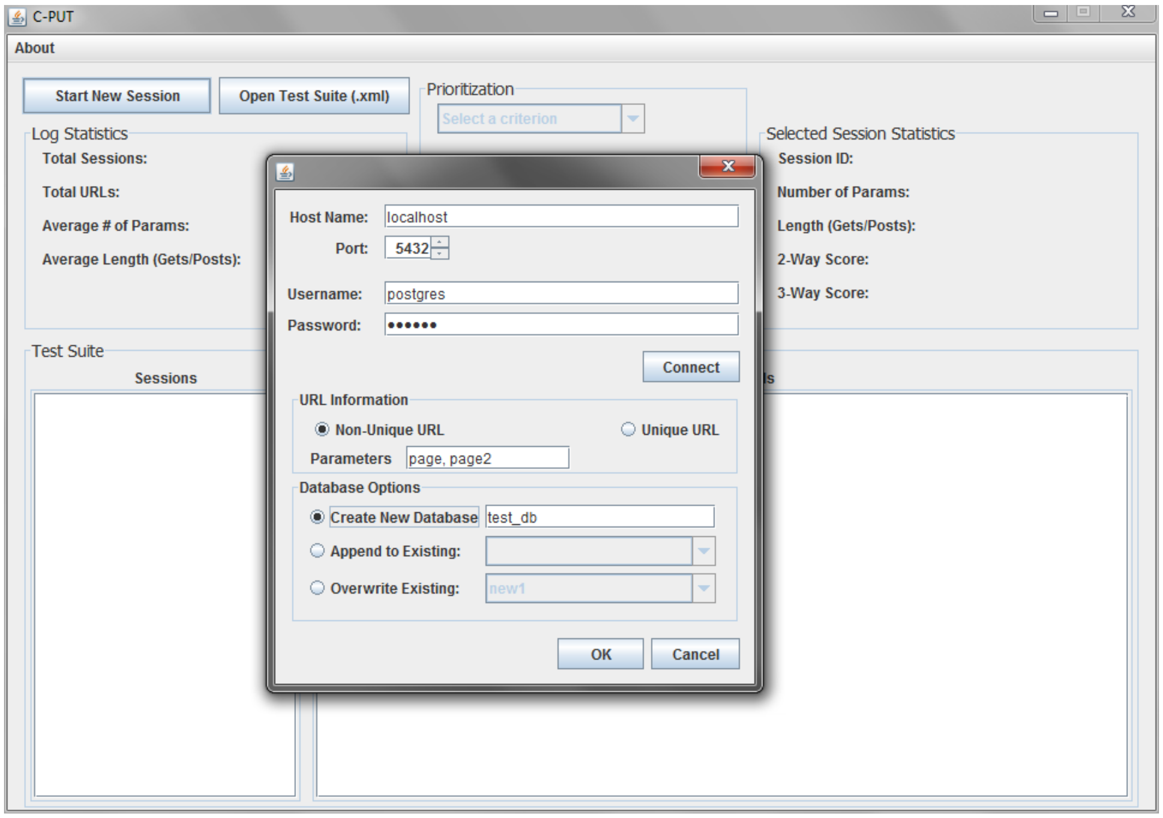
then loads the usage logs into CPUT which parses the log to create XML-format test cases. The XML test cases can then be prioritized using a particular test prioritization method.

6.1 Logger for Apache

The logger for Apache was implemented in C as a module. The module was generically designed to deploy on Apache that is running on both Windows or Linux platforms. The module logs the HTTP GET and POST requests. The HTTP GET requests are typically logged by default in most web servers. HTTP POST requests generally transmit form data as part of the HTTP request body, instead of being appended to the URL. Therefore, additional methods were necessary to gather the date associated with an HTTP POST request. This module should be included with other Apache modules and can be enabled by setting the Apache server's configuration file.

6.2 XML Test Case Generation

The test case generation utilizes previously used heuristics to convert a usage log into test cases. Specifically, the cookie information, the IP address, and the time stamp of each



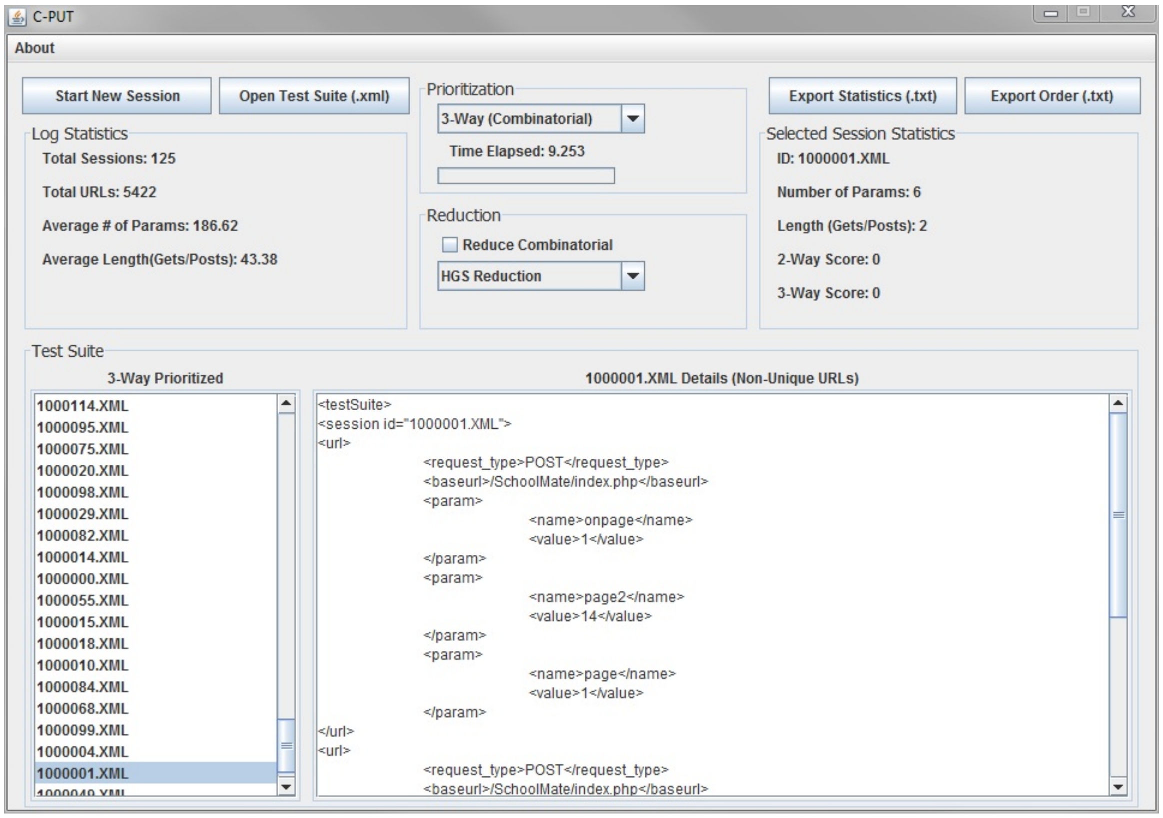
Host Name: localhost
Port: 5432
Username: postgres
Password: ••••••

[✓ Non-Unique URL] Unique URL
Parameters: page, page2

[✓ Create New Database] test_db
Append to Existing:
Overwrite Existing: new1

Figure 6.2. CPUT screen with options to load log file into database.

request are used to assign a request with a test case [37, 38]. The usage log and the test cases are stored in a PostGreSQL database. Figure 6.2 shows the CPUT screen when a user specifies options to load the log file into the database. Storing the logs and test cases in a database allows for efficient storage and retrieval. The test cases from the database table are then converted into test cases in XML format. Figure 6.3 shows the CPUT screen with all the XML test cases parsed from the log file. An XML format was chosen because of the extensible and easily parsable nature of XML. Figure 6.4 shows a sample test case. The important tags include: *testSuite* denotes the test suite, *session_id* represents the unique ID of a test case within the test suite, and *url* represents a page that the test case accesses and has an associated request with a *request_type* of GET or POST. Within a request, a *baseurl* incudes the specific page that is accessed and parameters (denoted as *param*) that have *names* for parameters and *values* that are assigned to the parameter.



Prioritization: 3-Way (Combinatorial)

Time Elapsed: 9.253

Reduction: HGS Reduction

Log Statistics

Total Sessions: 125

Total URLs: 5422

Average # of Params: 186.62

Average Length(Gets/Posts): 43.38

Selected Session Statistics

ID: 1000001.XML

Number of Params: 6

Length (Gets/Posts): 2

2-Way Score: 0

3-Way Score: 0

```
1000114.XML
1000095.XML
1000075.XML
1000020.XML
1000098.XML
1000029.XML
1000082.XML
1000014.XML
1000000.XML
1000055.XML
1000015.XML
1000018.XML
1000010.XML
1000084.XML
1000068.XML
1000099.XML
1000004.XML
1000001.XML
```

```
<testSuite>
<session id="1000001.XML">
<url>
            <request_type>POST</request_type>
            <baseurl>/SchoolMate/index.php</baseurl>
            <param>
                        <name>onpage</name>
                        <value>1</value>
            </param>
            <param>
                        <name>page2</name>
                        <value>14</value>
            </param>
            <param>
                        <name>page</name>
                        <value>1</value>
            </param>
</url>
<url>
            <request_type>POST</request_type>
            <baseurl>/SchoolMate/index.php</baseurl>
```

Figure 6.3. CPUT: Screenshot.

6.3 Prioritization Criteria

The test prioritization engine allows a user to select one of several prioritization methods shown to be effective in [8]. The options include:

1. **Length (GETS/POSTS):** Order test cases in descending order of the number of GET/POST requests in each test case.

2. **Number of parameters:** Order test cases in descending order of the number of parameters that are assigned values in each test case, i.e., set a parameter called "shipment type" to the value "overnight" on a page of an on-line store.

3. **Two-way combinatorial:** Order test cases in descending order of the number of unique two-way parameter-value interactions between windows in each test case. Once

```
<testSuite>
<session id=“1.XML”>
<url>
  <request_type>GET</request_type>
</url>
<url>
  <request_type>POST</request_type>
  <baseurl>/SchoolMate/index.php<baseurl>
  <param>
    <name>username</name>
    <value>johndoe</value>
  </param>
  <param>
    <name>password</name>
    <value>myUnsecurePassword!</value>
  </param>
  <param>
    <name>login</name>
    <value>1</value>
  </param>
</url>
<url>
  <request_type>POST</request_type>
  <baseurl>/SchoolMate/index.php<baseurl>
  <param>
    <name>logout</name>
    <value>1</value>
  </param>
</url>
</session>
</testSuite>
```

Figure 6.4. Sample test case in XML format of a user that logs in and out of a web site.

a t-tuple is covered in a test, we mark it as “covered” and only count unique t-tuples that have not been covered in previously selected test cases.

4. **Three-way combinatorial:** Order test cases in descending order of the number of unique three-way parameter-value interactions between windows in each test case. Once a t-tuple is covered in a test, we mark it as “covered” and only count unique

t-tuples that have not been covered in previously selected test cases.

5. **Random:** Prioritize using the random function in Java. This option is only recommended for use as a baseline in empirical studies.

Depending on the user's selection, CPUT applies the prioritization criteria and generates a test ordering. The ordered test cases are displayed to the user and written to a file. A tester can apply some prioritization criteria, export the test order, and then replay the test cases using a test execution engine. CPUT also generates test statistics, such as the number of parameters in the test case, the length of the test case in terms of GET/POST requests, and the two-way score which is the number of previously uncovered two-way interactions in the test case. Figure 6.3 shows the CPUT screen when the three-way prioritization criterion is selected. The tests in the bottom-left pane are ordered by the three-way prioritization criterion. The test order can be exported to a file that the tester can use when replaying the tests. We implemented CPUT and tested the tool with the deployment and collection of logs for an open-source PHP application, SchoolMate. We include our SchoolMate log as an example for users that download CPUT.

CHAPTER 7
CONCLUSIONS

Algorithms for combinatorial interaction testing provide systematic coverage of t-way interactions in a system. Our application of t-way combinatorial coverage for test suite prioritization of user-session-based testing differs in that the test suite already exists and may not contain all possible t-way interactions in a system, since test cases are generated by users that visit a website. It is unlikely that users of many systems will exhaustively cover all t-way interactions during their visits, particularly when users have unique user ids, passwords, and personal information that they enter into a system. This raises the need for an algorithm that does not enumerate all possible t-tuples to track and instead only stores the valid t-tuples in the test suite in order to save memory. Our experiments show that our approach scales well for a medium-sized web application and user base in which we capture test cases for 15 days and then double the log. Further, our empirical study examines the application of three-way inter-window parameter-value interaction coverage applied to the SchoolMate web application that was seeded with 66 faults. We collected test suites for each of the three user types for SchoolMate, prioritized the test suites, and compared the rate of fault detection with five prioritization criteria. Prioritization by two-way and three-way criteria were most effective, both performing within 1% of each other. However, two-way prioritization provided a slightly better rate of fault detection. A closer look at the data revealed that the system contained more faults triggered by two-way than by three-way inter-window parameter-value interactions. These results are similar to previous work by Kuhn et al. The authors report that systems typically have more faults triggered by lower strength interaction coverage [18].

Future work may examine a larger set of empirical studies with applications in which faults may potentially be triggered by higher strength interactions. Future work may also look at intra-window event interactions. Finally, we envision that future work may examine rich internet applications (RIAs), specifically RIAs with many AJAX type requests to the server. Another area would be to have a slight variation on the way the t-way scores are calculated. For instance, weights may be applied for preference to specific pages, parameters, or values. Finally, other algorithmic techniques such as heuristic searches may be used to prioritize test suites.

REFERENCES

[1] Alshahwan, N., and Harman, M. Automated session data repair for web application regression testing. In *International Conference on Software Testing, Verification and Validation* (Apr. 2008), pp. 298–307.

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

[3] Benedikt, M., Freire, J., and Godefroid, P. VeriWeb: Automatically testing dynamic web sites. In *the Eleventh International Conference on World Wide Web* (May 2002).

[4] Bryce, R., and Colbourn, C. The density algorithm for pairwise interaction testing. *Software Testing, Verification, and Reliability 17*, 3 (Aug. 2007), 159–182.

[5] Bryce, R., and Colbourn, C. Prioritized interaction testing for pairwise coverage with seeding and avoids. *Information and Software Technology Journal 48*, 10 (Oct. 2007), 960–970.

[6] Bryce, R., and Colbourn, C. A density-based greedy algorithm for higher strength covering arrays. *Software Testing, Verification, and Reliability 19*, 1 (Mar. 2009), 37–53.

[7] Bryce, R., Colbourn, C., and Cohen, M. A framework of greedy methods for constructing interaction tests. In *International Conference on Software Engineering (ICSE)* (May 2005), pp. 146–155.

[8] Bryce, R., Sampath, S., and Memon, A. Developing a single model and test prioritization strategies for event-driven software. *Transactions on Software Engineering 37*, 1 (Jan. 2011), 48–64.

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

[10] Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering 23*, 7 (Jul. 1997), 437 – 444.

[11] Colbourn, C. J. Combinatorial aspects of covering arrays. *Le Matematiche (Catania) 58* (2004), 121–167.

[12] D.R.Kuhn, and V.Okun. Pseudo-exhaustive testing for software. In *30th NASA/IEEE Software Engineering Workshop* (2006), pp. 153–158.

[13] Elbaum, S., Karre, S., and Rothermel, G. Improving web application testing with user session data. In *International Conference on Software Engineering* (Sep. 2003), pp. 49–59.

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

[15] Guo, Y., and Sampath, S. Web application fault classification - an exploratory study. *Empirical Software Engineering and Measurement* (2008), 303–305.

[16] Halfond, W., and Orso, A. Improving test case generation for web applications using automated interface discovery. In *ESEC / 15. SIGSOFT FSE* (Sep. 2007), pp. 145–154.

[17] HttpUnit. http://httpunit.sourceforge.net/, accessed on Dec. 19, 2011.

[18] Kuhn, D., Kacker, R., and Lei, Y. Practical combinatorial testing. In *NIST Tech Report 800-142* (Oct. 2010), pp. 1–70.

[19] Kuhn, D., and Reilly, M. An investigation of the applicability of design of experiments to software testing. In *27th NASA/IEEE Software Engineering Workshop* (Dec. 2006).

[20] Kuhn, D., Wallace, D., and Gallo, A. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering 30*, 6 (2004), 418–421.

[21] Kuhn, D. R., Wallace, D. R., and Gallo, A. M. Software fault interactions and implications for software testing. *Transactions on Software Engineering 30*, 6 (Oct. 2004), 418–421.

[22] Kung, D. C., Liu, C.-H., and Hsia, P. An object-oriented web test model for testing web applications. In *the Asia-Pacific Conference on Quality Software* (Oct. 2000), IEEE Computer Society, pp. 111–120.

[23] K.Z.Bell. *Optimizing Effectiveness and Efficiency of Software Testing: A Hybrid Approach.* PhD thesis, North Carolina State University, 2006.

[24] Lei, Y., Kacker, R., Kuhn, D., Okun, V., and Lawrence, J. Ipog/ipod: Efficient test generation for multi-way software testing. *Journal of Software Testing, Verification, and Reliability 18*, 3 (2008), 125–148.

[25] Liu, C., Kung, D., Hsia, P., and Hsu, C. Structural testing of web applications. In *International Symposium on Software Reliability Engineering* (Oct. 2000), pp. 84–96.

[26] Offutt, J., Wu, Y., Du, X., and Huang, H. Bypass testing of web applications. In *International Symposium on Software Reliability and Engineering* (Nov. 2004), IEEE Computer Society, pp. 187–197.

[27] Qian, Z. Test case generation and optimization for user session-based web application testing. *Journal of Computers 5*, 11 (2010), 1655–1662.

[28] Rational Robot. http://www.ibm.com/software/awdtools/tester/robot/, accessed on Dec. 19, 2011.

[29] Ricca, F., and Tonella, P. Analysis and testing of web applications. In *International Conference on Software Engineering* (May 2001), IEEE Computer Society, pp. 25–34.

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

[31] RTI. The economic impacts of inadequate infrastructure for software testing. Tech. Rep. 7007.011, National Institute of Standards and Technology, May 2002.

[32] Sampath, S., Bryce, R., Jain, S., and Manchester, S. A tool for combinatorial-based prioritization and reduction of user-session-based test suites. In *International Conference on Software Maintenance: Tool Demo Track* (Sep. 2011), pp. 574 – 577.

[33] Sampath, S., Bryce, R., Viswanath, G., Kandimalla, V., and Koru, A. G. Prioritizing user-session-based test cases for web application testing. In *International Conference on Software Testing, Verification and Validation* (Apr. 2008), pp. 141–150.

[34] Sampath, S., Mihaylov, V., Souter, A., and Pollock, L. Composing a framework to automate testing of operational web-based software. In *International Conference on Software Maintenance* (Sep. 2004), IEEE Computer Society, pp. 104–113.

[35] Sampath, S., Sprenkle, S., Gibson, E., and Pollock, L. Web application testing with customized test requirements—an experimental comparison study. In *International Symposium on Software Reliability Engineering* (Nov. 2006), pp. 266–278.

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

[37] Sampath, S., Sprenkle, S., Gibson, E., Pollock, L., and Souter, A. Applying concept analysis to user-session-based testing of web applications. Tech. Rep. 2006-329, University of Delaware, 2006.

[38] Sprenkle, S., Gibson, E., Sampath, S., and Pollock, L. Automated replay and failure detection for web applications. In *International Conference of Automated Software Engineering* (Nov. 2005), pp. 253–262.

[39] Wallace, D., and Kuhn, D. Failure modes in medical device software: An analysis of 15 years of recall data. *International Journal of Reliability, Quality, and Safety Engineering 8*, 4 (2001), 351–371.

[40] Wang, W., Sampath, S., Lei, Y., and Kacker, R. An interaction-based test sequence generation approach for testing web applications. In *IEEE International Conference on High Assurance Systems Engineering* (Nanjing, China, 2008), IEEE Computer Society, pp. 209–218.

[41] Xie, Q., and Memon, A. M. Studying the characteristics of a 'good' GUI test suite. In *International Symposium on Software Reliability Engineering (ISSRE 2006)* (Nov. 2006), IEEE Computer Society Press.

APPENDICES

Appendix A

Test Suites

The following figures show a test case from the test suite used in the experiment described in Chapter 4. The test case has 12 GETS/POSTS, 50 parameters, a two-way score of 500 and a three-way score of 623.

```
<testSuite>
<session id="1000091.XML">
<url>
  <request_type>POST</request_type>
  <baseurl>/SchoolMate/index.php</baseurl>
  <param>
    <name>username</name>
    <value>star</value>
  </param>
  <param>
    <name>password</name>
    <value>wars</value>
  </param>
  <param>
    <name>page</name>
    <value>0</value>
  </param>
  <param>
    <name>login</name>
    <value>1</value>
  </param>
</url>
<url>
  <request_type>POST</request_type>
  <baseurl>/SchoolMate/index.php</baseurl>
  <param>
    <name>semester</name>
    <value>3</value>
  </param>
  <param>
    <name>page</name>
    <value>2</value>
  </param>
  <param>
    <name>teacherid</name>
    <value>3</value>
  </param>
</url>
<url>
  <request_type>POST</request_type>
```

Figure A.1. Sample test case (part 1) from test suite described in Section 4.2
.

```
    <baseurl>/SchoolMate/index.php</baseurl>
    <param>
      <name>page2</name>
      <value>0</value>
    </param>
    <param>
      <name>page</name>
      <value>2</value>
    </param>
  </url>
  <url>
    <request_type>POST</request_type>
    <baseurl>/SchoolMate/index.php</baseurl>
    <param>
      <name>semester</name>
      <value>1</value>
    </param>
    <param>
      <name>page2</name>
      <value>1</value>
    </param>
    <param>
      <name>page</name>
      <value>2</value>
    </param>
    <param>
      <name>selectclass</name>
      <value>4</value>
    </param>
    <param>
      <name>teacherid</name>
      <value>3</value>
    </param>
  </url>
  <url>
    <request_type>POST</request_type>
    <baseurl>/SchoolMate/index.php</baseurl>
    <param>
      <name>page2</name>
      <value>8</value>
    </param>
```

Figure A.2. Sample test case (part 2) from test suite described in Section 4.2.

```
  <param>
    <name>page</name>
    <value>2</value>
  </param>
  <param>
    <name>selectclass</name>
    <value>4</value>
  </param>
</url>
<url>
  <request_type>POST</request_type>
  <baseurl>/SchoolMate/index.php</baseurl>
  <param>
    <name>page2</name>
    <value>1</value>
  </param>
  <param>
    <name>page</name>
    <value>2</value>
  </param>
  <param>
    <name>selectclass</name>
    <value>4</value>
  </param>
</url>
<url>
  <request_type>POST</request_type>
  <baseurl>/SchoolMate/index.php</baseurl>
  <param>
    <name>page2</name>
    <value>2</value>
  </param>
  <param>
    <name>page</name>
    <value>2</value>
  </param>
  <param>
    <name>selectclass</name>
    <value>4</value>
  </param>
</url>
<url>
  <request_type>POST</request_type>
```

Figure A.3. Sample test case (part 3) from test suite described in Section 4.2.

```
<baseurl>/SchoolMate/index.php</baseurl>
<param>
  <name>delete%5B%5D</name>
  <value>31</value>
</param>
<param>
  <name>delete%5B%5D</name>
  <value>29</value>
</param>
<param>
  <name>delete%5B%5D</name>
  <value>30</value>
</param>
<param>
  <name>delete%5B%5D</name>
  <value>28</value>
</param>
<param>
  <name>delete%5B%5D</name>
  <value>32</value>
</param>
<param>
  <name>deleteassignment</name>
  <value>0</value>
</param>
<param>
  <name>selectassignment</name>
  <value>5</value>
</param>
<param>
  <name>page2</name>
  <value>2</value>
</param>
<param>
  <name>onpage</name>
  <value>2</value>
</param>
<param>
  <name>selectclass</name>
  <value>4</value>
</param>
<param>
  <name>page</name>
```

Figure A.4. Sample test case (part 4) from test suite described in Section 4.2.

```
      <value>2</value>
    </param>
  </url>
  <url>
    <request_type>POST</request_type>
    <baseurl>/SchoolMate/index.php</baseurl>
    <param>
      <name>deleteassignment</name>
      <value>0</value>
    </param>
    <param>
      <name>page2</name>
      <value>2</value>
    </param>
    <param>
      <name>onpage</name>
      <value>3</value>
    </param>
    <param>
      <name>selectclass</name>
      <value>4</value>
    </param>
    <param>
      <name>page</name>
      <value>2</value>
    </param>
  </url>
  <url>
    <request_type>POST</request_type>
    <baseurl>/SchoolMate/index.php</baseurl>
    <param>
      <name>deleteassignment</name>
      <value>0</value>
    </param>
    <param>
      <name>page2</name>
      <value>2</value>
    </param>
    <param>
      <name>onpage</name>
      <value>3</value>
    </param>
```

Figure A.5. Sample test case (part 5) from test suite described in Section 4.2.

```
  <param>
    <name>selectclass</name>
    <value>4</value>
  </param>
  <param>
    <name>page</name>
    <value>2</value>
  </param>
</url>
<url>
  <request_type>POST</request_type>
  <baseurl>/SchoolMate/index.php</baseurl>
  <param>
    <name>page2</name>
    <value>3</value>
  </param>
  <param>
    <name>page</name>
    <value>2</value>
  </param>
  <param>
    <name>selectclass</name>
    <value>4</value>
  </param>
</url>
<url>
  <request_type>POST</request_type>
  <baseurl>/SchoolMate/index.php</baseurl>
  <param>
    <name>page2</name>
    <value>9</value>
  </param>
  <param>
    <name>page</name>
    <value>2</value>
  </param>
  <param>
    <name>selectclass</name>
    <value>4</value>
  </param>
</url>
</session>
</testSuite>
```

Figure A.6. Sample test case (part 6) from test suite described in Section 4.2.

Appendix B

Code Coverage

The following tables summarize the code coverage in terms of lines of code and percent covered for the entire test suite used for the experiment described in Chapter 4.

Table B.1. Summary (Part 1) of Percent Code Coverage and Number of Lines Covered by Each Individual Test Case.

Test Case ID	**% Code Covered**	**Number Lines Covered**
1000000	14.95%	896
1000001	8.63%	517
1000002	28.50%	1708
1000003	7.74%	464
1000004	9.14%	548
1000005	33.18%	1989
1000006	37.79%	2265
1000007	15.88%	952
1000008	54.20%	3249
1000009	39.66%	2377
1000010	16.95%	1016
1000011	32.52%	1949
1000012	21.04%	1261
1000013	31.75%	1903
1000014	9.18%	550
1000015	10.08%	604
1000016	27.88%	1671
1000017	39.02%	2339
1000018	9.11%	546
1000019	19.84%	1189
1000020	12.93%	775
1000021	16.78%	1006
1000022	37.32%	2237
1000023	34.88%	2091
1000024	27.66%	1658
1000025	12.93%	775
1000026	37.34%	2238
1000027	17.08%	1024
1000028	29.05%	1741
1000029	17.32%	1038
1000030	22.76%	1364

Table B.2. Summary (Part 2) of Percent Code Coverage and Number of Lines Covered by Each Individual Test Case.

Test Case ID	**% Code Covered**	**Number Lines Covered**
1000031	11.49%	689
1000032	47.25%	2832
1000033	13.68%	820
1000034	17.08%	1028
1000035	13.81%	828
1000036	12.35%	740
1000037	19.89%	1192
1000038	18.23%	1093
1000039	59.86%	3588
1000040	29.33%	1758
1000041	41.01%	2458
1000042	18.25%	1094
1000043	13.68%	820
1000044	24.57%	1473
1000045	23.31%	1397
1000046	14.28%	856
1000047	20.05%	1202
1000048	13.25%	794
1000049	16.80%	1007
1000050	26.66%	1598
1000051	50.57%	3031
1000052	16.58%	994
1000053	23.59%	1414
1000054	18.15%	1088
1000055	15.28%	916
1000056	22.86%	1370
1000057	11.90%	713
1000058	26.56%	1592
1000059	16.28%	976
1000060	26.56%	1592
1000061	12.48%	748
1000062	17.12%	1026

Table B.3. Summary (Part 3) of Percent Code Coverage and Number of Lines Covered by Each Individual Test Case.

Test Case ID	**% Code Covered**	**Number Lines Covered**
1000063	29.23%	1752
1000064	18.17%	1089
1000065	25.34%	1519
1000066	13.16%	789
1000067	26.59%	1594
1000068	11.90%	713
1000069	11.93%	715
1000070	16.03%	961
1000071	17.83%	1069
1000072	18.25%	1094
1000073	23.14%	1387
1000074	12.95%	776
1000075	16.68%	1000
1000076	48.65%	2916
1000077	13.16%	789
1000078	31.61%	1895
1000079	26.04%	1561
1000080	17.10%	1025
1000081	13.00%	779
1000082	12.23%	733
1000083	39.14%	2346
1000084	10.11%	606
1000085	44.23%	2651
1000086	12.38%	742
1000087	13.73%	823
1000088	15.67%	939
1000089	44.91%	2692
1000090	13.70%	821
1000091	14.51%	870
1000092	32.33%	1938
1000093	47.81%	2866
1000094	22.91%	1373
1000095	13.80%	827

Table B.4. Summary (Part 4) of Percent Code Coverage and Number of Lines Covered by Each Individual Test Case.

Test Case ID	**% Code Covered**	**Number Lines Covered**
1000096	13.98%	838
1000097	14.30%	857
1000098	14.46%	867
1000099	11.90%	713
1000100	12.50%	749
1000101	32.18%	1929
1000102	42.88%	2570
1000103	13.70%	821
1000104	13.25%	794
1000105	25.83%	1548
1000106	12.31%	738
1000107	15.35%	920
1000108	39.02%	2339
1000109	13.70%	821
1000110	55.29%	3314
1000111	20.19%	1210
1000112	38.89%	2331
1000113	39.32%	2357
1000114	17.50%	1049
1000115	18.80%	1127
1000116	12.51%	750
1000117	20.72%	1242
1000118	28.16%	1688
1000119	17.12%	1026
1000120	31.00%	1858
1000121	21.96%	1316
1000122	13.70%	821
1000123	18.42%	1104
1000124	35.37%	2120

Appendix C

Seeded Fault Summary

The 66 seeded faults used for the experiment described in Chapter 4 are broken up by fault category and user type. In addition, the logic fault category is broken down into five sub-categories.

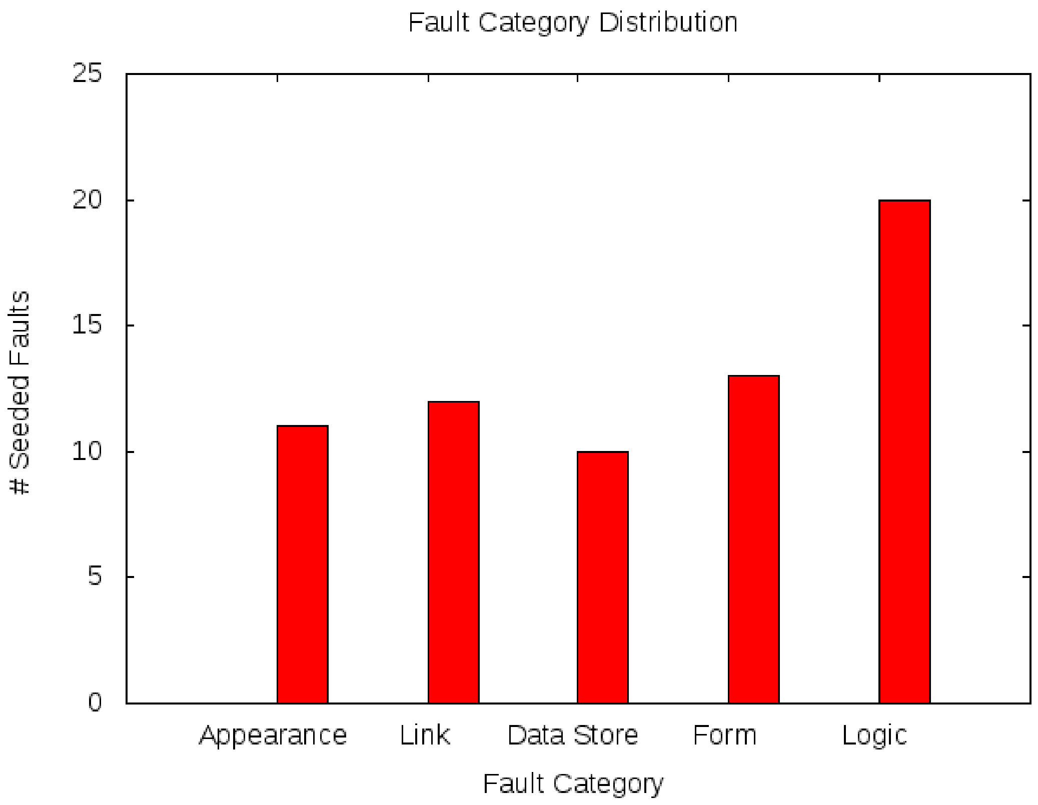


Figure C.1. Fault category distribution for 66 seeded faults discussed in Chapter 4.

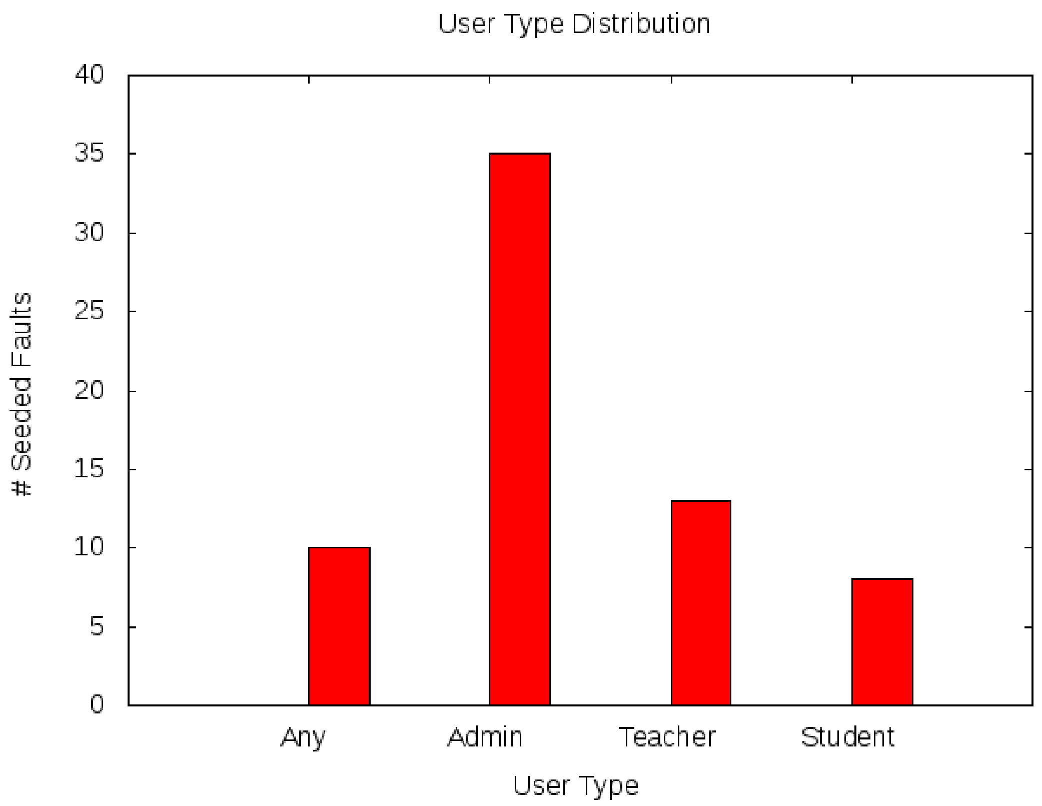


Figure C.2. User type distribution for 66 seeded faults discussed in Chapter 4.

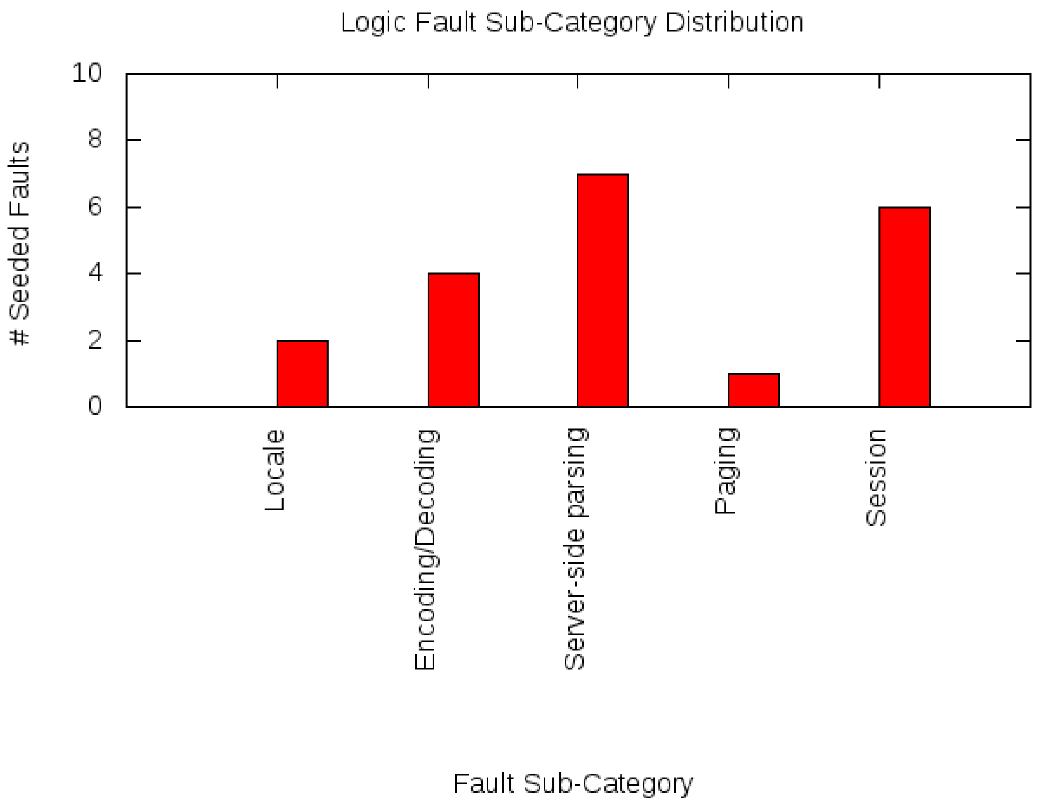


Figure C.3. Logic fault sub-category distribution for 20 seeded logic faults.