

1-1-2008

Test suite prioritization techniques applied to Web-based applications

Vani Kandimalla
University of Nevada, Las Vegas

Follow this and additional works at: <https://digitalscholarship.unlv.edu/rtds>

Repository Citation

Kandimalla, Vani, "Test suite prioritization techniques applied to Web-based applications" (2008). *UNLV Retrospective Theses & Dissertations*. 2376.
<http://dx.doi.org/10.25669/29kj-d23a>

This Thesis is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

TEST SUITE PRIORITIZATION TECHNIQUES APPLIED TO
WEB-BASED APPLICATIONS

by

Vani Kandimalla

B.Tech, Electronics and Communication Engineering
Jawaharlal Nehru Technological University, Hyderabad, India,
2004

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2008

UMI Number: 1460533

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 1460533

Copyright 2009 by ProQuest LLC.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 E. Eisenhower Parkway
PO Box 1346
Ann Arbor, MI 48106-1346



Thesis Approval
The Graduate College
University of Nevada, Las Vegas

JULY 24TH, 2008

The Thesis prepared by

VANI KANDIMALLA


Entitled

TESTCASE PRIORITIZATION TECHNIQUES APPLIED FOR WEB-BASED APPLICATIONS.

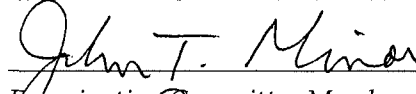
is approved in partial fulfillment of the requirements for the degree of

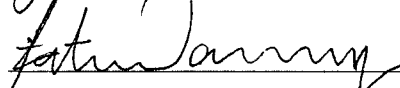
MASTER OF SCIENCE IN COMPUTER SCIENCE


Examination Committee Chair


Dean of the Graduate College


Examination Committee Member


Examination Committee Member


Graduate College Faculty Representative

ABSTRACT

Test Suite Prioritization Techniques applied to Web-Based Applications

by

Vani Kandimalla

Dr. Renee Bryce, Examination Committee Chair
Assistant Professor, School of Computer Science
University of Nevada, Las Vegas

Web applications have rapidly gained importance in many businesses. The increased usage of web applications has created a challenging need for efficient and effective web application testing strategies. This thesis examines one aspect of web testing, that of test suite prioritization. We examine new test suite prioritization strategies that may improve the rate of fault detection for user-session based test suites. These techniques consider test-lengths and systematic coverage of parameter-values and their interactions. Experimental results show that some of these prioritization strategies often improve the rate of fault detection of test suites when compared to random ordering of the test cases. In general the most effective prioritization strategies consider the systematic coverage of the combinations of parameter-values as early as possible.

TABLE OF CONTENTS

ABSTRACT.....	iii
LIST OF TABLES.....	vi
LIST OF FIGURES.....	vii
ACKNOWLEDGEMENTS	viii
CHAPTER 1 INTRODUCTION	1
1.1. Goal and Scope.....	4
CHAPTER 2 BACKGROUND AND RELATED WORK.....	6
2.1. Web Applications	6
2.2. User-session-based Testing	6
2.3. Test Case Prioritization	8
2.4. Related Work	10
CHAPTER 3 TEST CASE PRIORITIZATION STRATEGIES	15
3.1. Generation of test cases	16
3.2. Test Lengths	17
3.3. Systematic Prioritization by Parameter-Values.....	18
3.3.1. Unique parameter-value coverage.....	19
3.3.2. Parameter-value Interaction Coverage	19
3.3.3. Length by parameter-value counts	21
3.4. Random	21
CHAPTER 4 EXPERIMENTAL EVALUATION	22
4.1. Parsing Tool	22
4.1.1. Test set generation	22
4.1.2. General Layout of GUI.....	28
4.1.3. Major Operations	29
4.2. Experiments.....	40
CHAPTER 5 EXPERIMENTAL RESULTS	45
5.1. CPM.....	45

5.2. MASPLAS.....	46
5.3. BOOKS	48
CHAPTER 6 SUMMARY AND CONCLUSION	50
6.1. Summary of Results.....	50
6.2. Conclusion.....	51
BIBLIOGRAPHY	53
VITA	56

LIST OF TABLES

Table 1.	Example Test Case.....	18
Table 2.	Four parameters can take on one of three values each	19
Table 3.	A set of test cases	20
Table 4.	2-way parameter-value interaction.....	20
Table 5.	Number of options for each page	25
Table 6.	Unique IDs of Pages	26
Table 7.	Unique IDs parameter-values.....	26
Table 8.	Test case generation	27
Table 9.	Simple URL format types	29
Table 10.	Subject Applications and Test Suite Characteristics	43
Table 11.	APFD for CPM (in percentage)	47
Table 12.	APFD for MASPLAS (in percentage)	47
Table 13.	BOOK: APFD Metric (in percentage)	49
Table 14.	Percent of Test Suite Run (Execution time in seconds).....	49

LIST OF FIGURES

Figure 1.	Pseudo code (Generation of test cases).....	16
Figure 2.	User Session captures	23
Figure 3.	Page names along with the associated data.....	24
Figure 4.	Front end of the Parsing Tool.....	28
Figure 5.	Simple URL format	30
Figure 6.	Uploading the User-session directory.....	31
Figure 7.	Files uploaded for parsing	32
Figure 8.	Display of the parsing Result.....	33
Figure 9.	Display of the Resultant test cases	34
Figure 10.	Error message when uploaded wrong directory	35
Figure 11.	Selecting the Prioritization type	36
Figure 12.	Test cases prioritized by length Longest to Shortest	37
Figure 13.	Test cases prioritized by length Shortest to Longest	37
Figure 14.	Status Bar, displaying the status of operation	38
Figure 15.	Confirmation window to clear all the fields.....	38
Figure 16.	Help window with Quick start guide	39

ACKNOWLEDGEMENTS

I am very much thankful to my advisor Dr. Renee Bryce for her constant support, encouragement and advice in my academic as well in my personal life. I am thankful to Dr. Evangelos Yfantis, Dr. John Minor and Dr. Fatma Nasoz for serving as my thesis advisors committee. Special thanks are extended to Dr. Sreedevi Sampath and Dr. Bryce for helping me throughout my research and allowing me to serve as co-author for the paper submitted to ICST. Also, thank you to Dr. Sampath for providing the test suites for the experiments summarized in this thesis and for computing the APFD for the results. I would like to thank Dr. Ajoy Datta for his advice and help throughout my academic career and my course work. I am also thankful to the members in the registrar's office for letting me be a part of their team; it was a great learning experience working at the registrar's office. Thanks to the faculty of the Computer Science Department for helping me pursue my academic goals.

I express my profound thanks to my Mom, Dad and my family for their constant support and encouragement throughout my life.

CHAPTER 1

INTRODUCTION

Web applications are critical to the day-to-day operations of businesses. Web applications may experience permanent, intermittent, or transient failures that may affect a web site. Failures in this domain result in losses of millions of dollars to organizations [17, 21]. A single hour of down time can cost a retailer lost sales. Most web applications must be available 24/7 and undergo continual modification throughout their lifetime. This requires testers to fix bugs in an application and deploy a new version quickly. As changes occur, the problem of testing modified versions of the application with respect to these changes efficiently is important. Regression testing is the activity of testing modified versions of software to increase the confidence that the changes behave as intended and do not adversely affect the rest of the software. Regression testing consumes approximately 50% of maintenance costs for software applications [3, 15]. Several tasks are involved in regression testing, such as selecting a subset of test cases to execute, prioritizing

test cases to achieve a performance goal, and augmenting a test suite with test cases to test the modified parts of the software. This testing activity has always been challenging because developers need to check not only the intended functionality of the changes themselves, but also the intended functionality of the rest of the software that interacts with the changes. Testers can benefit from test suites that can detect faults early in the test execution cycle. This thesis focuses on test suite prioritization.

As applications evolve, test cases from a previous version are reused to test the new version of the application. Usually, a large number of test cases accumulate over the life cycle of an application, which makes the reuse of all of these test cases to test the new version impractical. The tester is often required to select and execute a small subset of test cases. Test case prioritization is one such selection methodology where test cases are selected according to some criterion to meet a performance goal. While several strategies have been proposed and evaluated to prioritize test cases for C programs [8, 9, 24] and Java programs [4, 5], to our knowledge, little work has been done to prioritize test suites for web applications.

Usage data from web applications which can be converted into the test cases is easily available to the testers [25]. This conversion process is known as user-session-based testing [9, 25, 26]. Prioritizing test cases becomes particularly significant in user-session-based testing because a

large number of usage-based test cases can be present for a frequently used application. The data from the user log provides information about the user navigation through the web site along with the user induced events, which are ideal test cases. User sessions contain events that are typically base requests and name value pairs (for example, form field data) sent as requests to the web server. A base request for a Web application is the request type and resource location without the associated data (for example, GET /apps/bookstore/Login.jsp). The ability to record these requests is often built into the Web server, so little effort is needed to record the desired events. Cookies and the information about the IP address can be used to convert the requests in the web server log into a sequence of user-session based test cases. These requests exercise the complex interactions between the application components, for example a parameter-value specified by the user may access back-end code or retrieve the stored data in the databases. 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.

With the goal of identifying prioritization criteria that order test cases such that faults in the application are detected as early as possible in the test execution cycle, we examine prioritization metrics that are based on characteristics of a web application and user-session-based tests. Since

web applications exhibit characteristics of GUI applications and are largely driven by user input (i.e., events), we apply the prioritization techniques proposed in [4] to web based applications. We also examine frequency-based metrics that are unique for web application testing. The main contributions of this thesis are: *(1) a tool to parse and prioritize test suites and (2) a summary of empirical results for three web-based applications.*

Chapter 2 presents the background in web application testing, user-session-based testing, and test case prioritization. In chapter 3, we present prioritization metrics. Chapter 4 presents our subject applications and experimental methodology. We present and analyze the results in chapter 5 and conclude in chapter 6.

1.1. Goal and Scope

The goal of this research is to improve the quality of existing test suites with respect to the rate of fault detection as measured by APFD. To quantify this goal, Rothermel et al. introduced a metric, (Rothermel et al., 2001) [24] *APFD*, which measures the *Average of the Percentage of Faults Detected* (APFD) over the life of the suite. APFD values range from 0 to 100; higher numbers imply faster (better) fault detection rates. Let T be a test suite containing n test cases, and let F be a set of m faults revealed by T . Let TF_i be the first test case in ordering T' of T which reveals fault i . The APFD for test suite T' is given by the equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + \dots + TF_m}{nm} + \frac{1}{2n}$$

To address this goal, various test case prioritization techniques have been developed.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1. Web Applications

Web applications are one of the largest growing types of software. Web application software can be updated and maintained without distributing and installing software on client computers. Web pages can be either static, in which case the content of the page is the same on all client machines and to all the users, or dynamic where the contents of the page depends on input.

A main challenge faced with web applications is that they run on many platforms, use many technologies, and can be written in numerous languages.

2.2. User-session-based Testing

Experiments [9] show that user session data gathered as users operate web applications can be successfully employed in the testing of Web applications. Experiments compare new and existing test generation techniques for web applications, assessing both the adequacy of the generated tests and their ability to detect. Results show that user session

data can produce test suites as effective overall as those produced by existing white-box techniques but at less expense, can be used in automating the regressions testing process, and they also find different types of faults.

User-sessions captured from previous releases of the software can serve as regression tests. A user-session-based test case is the sequence of the HTTP requests containing base requests and the 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. User-session-based test cases can be generated from usage logs. User-session-based test cases begin 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 minutes gap between two requests from a server is considered equivalent to a session timing out in the test cases that we use. Different strategies can construct test cases for the collected user sessions [9, 20, 22, and 27].

Experiments [9] show that user session-based test cases are often efficient at detecting faults; however, a challenge arises on how to manage a large pool of such test cases. There are test suite reduction techniques based on criterion, such as covering all base requests in the

application while maintaining the use case representation. These reduction techniques reduce original suites [26] while maintaining overall fault finding effectiveness, but tests are in no particular order. Whereas, test suite prioritization uses the entire test suit 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

Regression testing of an application is the process of testing whether the recently modified software introduced any new faults into already tested code. Regression testing is very important, yet an expensive and time consuming process. 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 modification [19]. As an application evolves, test engineers run regression tests to validate new features and detect whether any new faults are introduced into previously tested code. There may be a large number of test cases available from testing previous versions of the application, which can be reused to test the new version of the application. However, running all of the test cases in a test suite may take a significant amount of time. For instance, Rothermel et. al. report an example in 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 a subset of test cases which can be executed to achieve the testing

objectives earlier in the testing process. The main testing objective we focused on is the rate of fault detection- a measure of how quickly a test order detects faults as measured by APFD.

One approach to selecting test cases is to schedule the test cases in an order according to some criterion that increases the effectiveness in meeting a performance goal. Scheduling test cases in this manner is known as test case prioritization. To reduce the cost of regression testing and the time involved in it, software testers may prioritize their test cases so that those which are more important, by some measure, are run earlier in the regression testing process. There are many possible goals for prioritization; [24] describes several. One possible goal of test case prioritization is that of increasing the test suit's rate of fault detection. An increased rate of fault detection can provide earlier feedback on the system under regression test and let developers begin locating and correcting faults earlier than might otherwise be possible. Such feedback can also provide evidence that a quality goal is still not met, allowing testers to take strategy decisions about release schedules. Further, an improved rate of fault detection can increase the likelihood that if testing is prematurely halted, those test cases that offer the greatest fault detection ability in the available testing time will have been executed. Other possible goals described in [24] include: early coverage of the code in the application under test, meeting code coverage criterion, increasing the confidence in the application under test at faster rate, and the

likelihood of catching the faults to specific code changes much earlier in the testing process. Additional criteria include code coverage, fault likelihood, and fault exposure potential [8, 9, 24].

Rothermel et. al.[24] define the test case prioritization problem and the issues relevant to the solutions. We will review a small portion of the material here, The test case prioritization problem is defined as follows:

The Test Case Prioritization Problem:

Given: T , a test suite; PT , the set of permutations of T ; and f , a function from PT to the real numbers.

Problem: Find $T' \in PT$ such that $(\forall T'') (T'' \in PT) (T'' \neq T') [f(T') \geq f(T'')]$.

Here, PT represents the set of all possible prioritizations (orderings) of T , and f is a function that, applied to any such ordering, yields an *award value* for that ordering [24].

2.4. Related Work

In recent years, research has been conducted addressing several techniques for the test case prioritization problem. We will review a small portion of the material of previous work done on prioritization in this section.

Wong et al. (Wong et al., 1997) [31] suggested a technique which prioritizes the test cases according to the criterion of “increasing cost per additional coverage”. The authors restricted prioritizing the subset of test cases which are selected from the test suite by a safe test selection

technique, and the subset of test cases selected are the one which reach the modified code, but other test cases can be placed after this subset for further execution. So, this technique is using the modification information, feedback and test cost information.

Rothermel et al. [24] and Elbaum et al. [6, 7] study prioritization. They define several prioritization techniques, which are classified into 2 categories:

- General test case prioritization

Prioritizing the test cases for finding the order that will be effective over a succession of subsequent versions of software.

- Version-specific test case prioritization

Prioritizing the test cases in a manner that will be most effective for a particular version of the software.

They restricted their attention to the version-specific test case prioritization operated at relatively fine granularity- that is, they involved instrumentation, analysis, and prioritization at the level of source code statements. An alternative is to operate at a relatively coarse granularity; prioritization, at the function level.

They summarized several techniques which are classified into 3 categories and present the results of several empirical studies of those techniques.

- Comparator Techniques

The techniques, which use the random ordering or the optimal ordering of the test cases, come under this category.

- Statement Level Techniques

This category consists of the techniques that prioritize the test cases by considering the attributes of the program at the statement level.

- Function Level techniques

This category consists of the techniques that prioritize the test cases by considering the attributes of the program at the functional level.

All the techniques suggested in this research improve the rate of fault detection, including the simplest one. The improvement in rate of fault detection occurs for both functional and statement level techniques.

Jones et al. (Jones and Harrold, 2001) [12] describe a technique for prioritizing test cases which can be used with the modified condition/decision coverage (MCDC) criteria, this technique uses feedback, but no modification information.

Srivastava and Thiagarajan (Srivastava and Thiagarajan, 2002) [30] present a technique for prioritizing the test cases based on the basic block coverage, which uses both feedback and the change information. This technique is different from the others as this computes the flow

graph and the coverage from the binaries, and tries to predict the possible affects on the control flow following the code modifications.

Jeffrey and Neelam [11] prioritize using relevant slices. Techniques used before for prioritizing the test cases were based on the total number of coverage requirements and additional requirement coverage exercised by the test cases. Total statement coverage prioritization orders the test cases in the decreasing order of the number of statements they exercise, and additional statement coverage prioritization orders the test cases in the decreasing order of the additional statements they exercise that have not been covered earlier in the prioritized sequence. This new test case prioritizing approach based on the relevant slices not only takes into account the total statement coverage, but also the number of statements executed that influence or have the potential to influence the output produced by the test cases.

Additional criteria exist for GUI-based programs. For instance, Bryce and Memon [4] prioritize preexisting 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,

prioritization by event-interactions results in the fastest rate of fault detection.

CHAPTER 3

TEST CASE PRIORITIZATION STRATEGIES

In this section, we examine prioritization functions for user-session-based testing. The functions include:

Test length based on number of base requests (LtoS, StoL): order test cases by the number of HTTP requests that they contain. Orderings include longest to shortest (LtoS) and shortest to longest (StoL).

Unique coverage of parameter-values (1-way): Order test cases by the number of unique parameter-values covered by each of the test cases.

2-way parameter-value interaction coverage (2- way): Order test cases by the count of pair wise combinations of parameter-values between pages.

Test length based on number of parameter-values (PV-L to S, PV- S to L): Ordered according to the number of parameter-values used in a test case. Orderings include Longest to Shortest (PV- L to S) and Shortest to Longest (PV- S to L),

Random: Execute the test cases in random order.

3.1. Generation of test cases

Application usage data is used for test cases. The usage data is captured from previous releases of the software. Converting usage data into test cases for testing web application is known as user-session-based testing.

Figure 1 explains the Parsing algorithm; conversion of usage data into test cases.

```
// Pseudo code
Input: user-sessions captured previously
int files = no: of user sessions captured .
while (files>0)
{
  Int Urlcount = no: of urls in the file.
  While (urlcount>0)
  {
    If (! page and parameter values already exist in the data structures) then
    {
      //Parse the URL for page name and the unique parameter values and assign them the unique
      // page no: and the param no: and store them in the data structure.
    }
  }
}
// do the same to parse the data and output the unique values from the data structures used for
// storing the page names and the parameter-values
```

Figure 1. Pseudo code (Generation of test cases)

3.2. Test Lengths

This technique orders test cases by selecting the next test case with maximum number of base requests, counting the duplicates. Ordering test cases based on the length of base requests can affect the rate of fault detection of the ordered test suite, since the amount of application code covered is also partially determined by the number of base requests in the test case. Table 1 shows an example of a test case, tc1 where the length tc1 is four i.e., the number of base requests in tc1. Register.jsp, Login.jsp, Search.jsp and Logout.jsp are the base requests covered by the test case tc1.

The test cases can be prioritized in descending order of the number of the base requests, Request-longest to shortest (*Req-L to S*) i.e., executing the test cases with more number of base requests before the test cases with less number of base requests, or in the ascending order of the number of base requests, Request-shortest to longest (*Req-S to L*) i.e., the test case with less number of base requests are covered first than the one with more number. Here the number of base requests also includes counting the duplicates.

Test case tcl 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

3.3. Systematic Prioritization by Parameter-Values

Most of the pages of the web application deal with the parameters for which the user needs to specify the value. For example consider the test case shown in Table 1. The *Login.jsp* page accessed in the test case has two *parameters*, “name” and “pswd” that can take on values. We can prioritize these user-sessions by the discrete number of values that have been specified for these parameters. For instance, test case *tcl* 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 2. Four parameters can take on one of three values each

3.3.1. Unique parameter-value coverage

This technique selects the next test that has the maximum number of the parameter-values that are not in the previously selected test.

3.3.2. Parameter-value Interaction Coverage

The t-way criteria selects the next test that maximizes the number of t -way parameter-value interactions between pages that occur in a test. Here t is set to 2 i.e., $t=2$ for pair wise coverage of parameter-values. Consider the example of 4 parameters as shown in Table 2 that can each take on one of three values from the list. Table 3 shows an example of parameter-values that occur in a set of test cases.

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 3. A set of test cases

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 4. 2-way parameter-value interaction

Table 4 lists the six pair wise parameter-value interactions that occur in Test 1. The number of previously uncovered parameter-values in each test is counted and prioritizes the tests by selecting the test with the maximum number of parameter values next.

3.3.3. Length by parameter-value counts

In this technique, test cases are prioritized according to the number of parameter-value pairs that each test-case contains, counting the duplicates. Selecting those tests with the largest number of parameter-values in a test first is called PV-LtoS (PV-Longest to Shortest). Conversely, selecting those tests with the smallest number of parameter-values first is called PV-StoL (PV- Shortest to Longest).

3.4. Random

We select test cases uniformly at random until there are no remaining test cases.

CHAPTER 4

EXPERIMENTAL EVALUATION

4.1. Parsing Tool

In our experiments the user sessions captured previously serve as tests. We develop a tool for parsing and prioritization. There are 2 major functionalities supported by the tool:

1. Parsing the user sessions for generating the test sets
2. Prioritizing the test set depending upon some criteria.

4.1.1. Test set generation

The input to the tool is the user sessions which contain the captured data from the user log. These user sessions provide information about the user navigation through the web site along with the user invoked events. User sessions contains events that are typically base requests and name value pairs (for example, form field data) sent as requests to the web server. A base request for a Web application is the request type

and resource location without the associated data (for example, GET /apps/bookstore/Login.jsp) and the associated data is the parameter-value pairs.

For example, Figure 2 shows a single captured user session, which is the set of urls through which the user navigates in a session. Session on a server is considered to be the time elapsed between the user login and the logout from a particular web application existing on the server.

We need to identify all windows, parameters and the values, for which the beginning part of the url should be parsed out, as shown in Figure 3.

```
http://dwalin.cis.udel.edu:8080/apps/bookstore/Login.jsp
http://dwalin.cis.udel.edu:8080/apps/bookstore/Login.jsp?
Password=guest&FormName=Login&FormAction=login&Login=guest
http://dwalin.cis.udel.edu:8080/apps/bookstore/ShoppingCart.jsp
http://dwalin.cis.udel.edu:8080/apps/bookstore/ShoppingCartRecord.jsp?order_id=2&
http://dwalin.cis.udel.edu:8080/apps/bookstore/ShoppingCart.jsp
http://dwalin.cis.udel.edu:8080/apps/bookstore/ShoppingCartRecord.jsp?order_id=2&
http://dwalin.cis.udel.edu:8080/apps/bookstore/ShoppingCart.jsp
http://dwalin.cis.udel.edu:8080/apps/bookstore/MyInfo.jsp?
http://dwalin.cis.udel.edu:8080/apps/bookstore/Default.jsp
```

Figure 2. User Session captures
(Base request along with parameter value pairs)

Figure 3 shows the part of the URL with page names along with the associated parameter-value pairs, which are further parsed for different pages.

The list of pages include: Login.jsp, ShoppingCart.jsp, MyInfo.jsp, Default.jsp, and ShoppingCartRecord.jsp.

The parameter-value are:

For page "Login" <Password,guest>, <FormName,Login>,<Login,guest>, <FormAction,login>

For Page "ShoppingCartRecord": <order_id,2>

Login.jsp
Login.jsp?Password=guest&FormName=Login&FormAction=login&Login=guest
ShoppingCart.jsp
ShoppingCartRecord.jsp?order_id=2&
ShoppingCart.jsp
ShoppingCartRecord.jsp?order_id=2&
ShoppingCart.jsp
MyInfo.jsp?
Default.jsp

Figure 3. Page names along with the associated data

The pair (or t-way interaction) are between windows. For example, since parameter-values <Password,guest> and <FormName,Login> are both from the Login page, they are not counted as an interaction.

However parameter-values <Password,guest> and <order_id,2> is considered to be a pair because they are from different windows (Login page and ShoppingCartRecord page).

We assign a unique ID to each page and parameter-value so that we make sure that we are testing interaction between pages.

Number of pages = 5

Number of parameter-values for pages is listed in Table 5.

No of Parameter-Values	No: of Pages	List of Parameter-values
5	1	page "Login" has parameter-value <none, none>, <password, guest>, <FormName, Login>, <FormAction, login>,<Login, guest>
1	1	page "ShoppingCart" has no parameter-values <none, none>
1	1	page "ShopCartRecord" has one parameter-value <order_id, 2>
1	1	page "Myinfo" has no parameter-value <none, none>
1	1	page "Default" has no parameter-value <none, none>

Table 5. Number of options for each page

We assigned a unique id to each of the pages and its associated parameter-values as shown in Table 6 & Table 7.

Page Name	IDs
Login	0
ShoppingCart	1
ShoppingCartRecord	2
MyInfo	3
Default	4

Table 6: Unique IDs of Pages

Page Name	Parameter-Values	IDs
Login page	<none, none>	0
	<Password, guest>	1
	<FormName, Login>	2
	<FormName, login>	3
	<Login,guest>	4
ShoppingCart page	<none, none>	5
ShoppingCartRecord	<order_id, 2>	6
MyInfo page	<none,none>	7
Default page	<none,none>	8

Table 7. Unique IDs parameter-values

Test cases will then be generated as shown in Table 8.

Login.jsp	0
Login.jsp?Password=guest&FormName=Login&FormAction=login&Login=guest	1,2,3,4
ShoppingCart.jsp	5
ShoppingCartRecord.jsp?order_id=2	6
ShoppingCart.jsp	5
ShoppingCartRecord.jsp?order_id=2	6
ShoppingCart.jsp	5
MyInfo.jsp?	7
Default.jsp	8

Table 8. Test case generation

The resultant test case that will be the input to the prioritization algorithm is:

0 1 2 3 4 5 6 5 6 5 7 8

This example is just for one test case, we need to enumerate all pages and parameter-values in the collection of tests, assign them unique IDs and then prioritize.

4.1.2. General Layout of GUI

Figure 4 shows the general layout of the Parsing tool GUI. “Files Uploaded” displays all the files of the user-sessions. “Parsing Result” shown in the figure is the result obtained after the test cases have been parsed and the unique IDs have been assigned to all the different pages and the parameter-values. “Resultant TestCases” displays the actual test cases that we input to the prioritization algorithm.

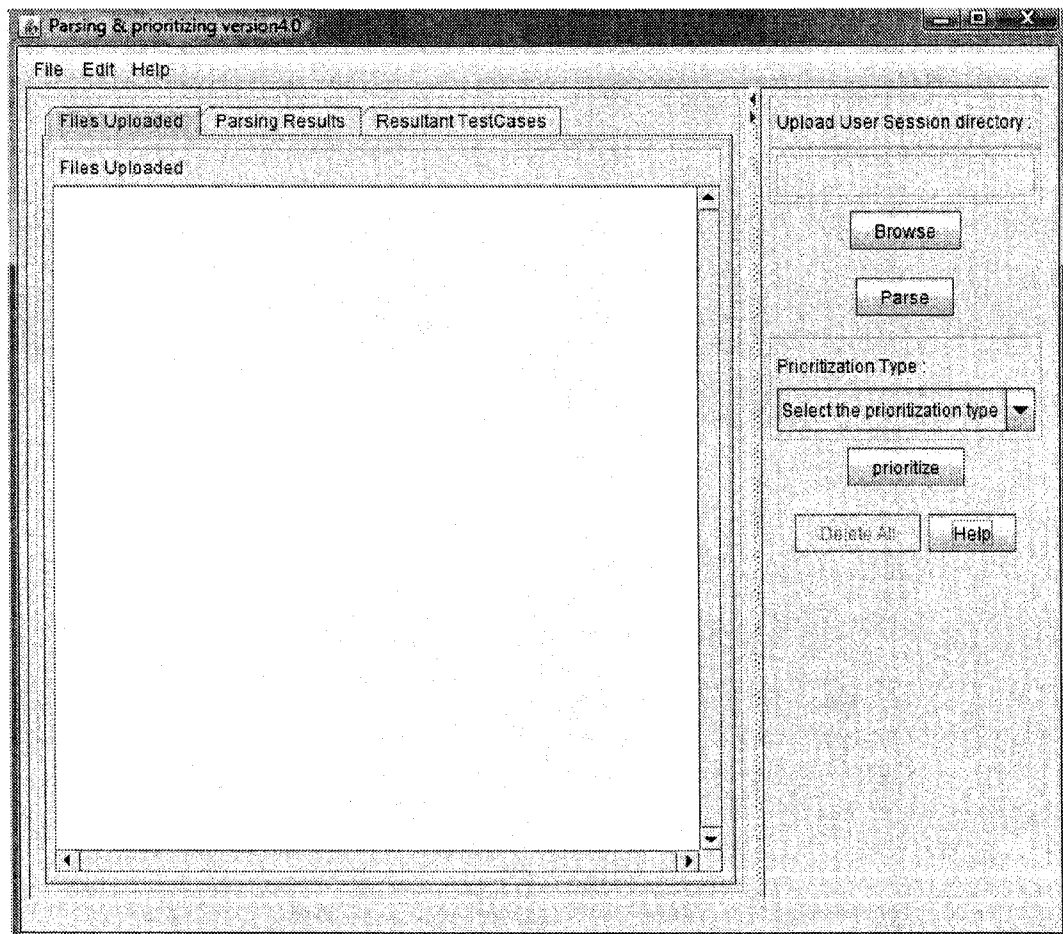


Figure 4. Front end of the Parsing Tool

4.1.3. Major Operations

There are 2 major functionalities supported by the tool

1. Parsing the user sessions for generating the test sets
2. Prioritizing the test set depending upon some criteria.

Currently the tool supports uploading files with the extensions: ".wget", ".us", and ".tc"

The user-sessions captured are with the file extensions listed above. In the experiments conducted on the user-sessions of three web applications,

1. ".wget" extension files have URLs of a simple format as explained below.

Simple URL format: This can be categorized into two types as shown listed in Table 9.

URL format	Example
Directory format	(http://dwalin.cis.udel.edu:8080/apps/bookstore)
File name format	(http://dwalin.cis.udel.edu:8080/apps/bookstore/Default.jsp)

Table 9: Simple URL format types

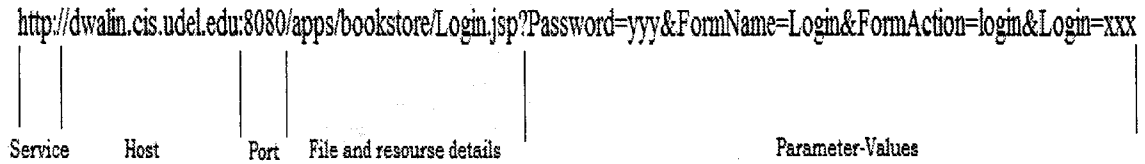


Figure 5. Simple Url format

2. ".us" and ".tc" extension files have URLs with GET and POST variables

I. URLs with GET and POST variable

Example:

- a. GET /scheduler/
- b. GET /scheduler/grader.html

II. URLs with GET and POST variable

Example:

- a. GET /masplas05/index.html POST
`/masplas05/FinalSubmission.do --post- data=`
`"&email=yyy%40g.y.j&last_name=mmm&first_name=t+"`

In example I, the URL "GET /scheduler/" has no parameter-values.

Parsing the user session for generating the test sets:

To start, upload the directory where the user-sessions reside, by clicking the "Browse" button as shown in Figure 6.

Select the proper directory where the user sessions reside and, by clicking the "Parse" button, sessions will be parsed and the result of parsing the user sessions, assigning the unique identifier to the page

names and the parameter-values, and the test sets are displayed in the respective fields provided as shown in Figure 7, Figure 8 and Figure 9.

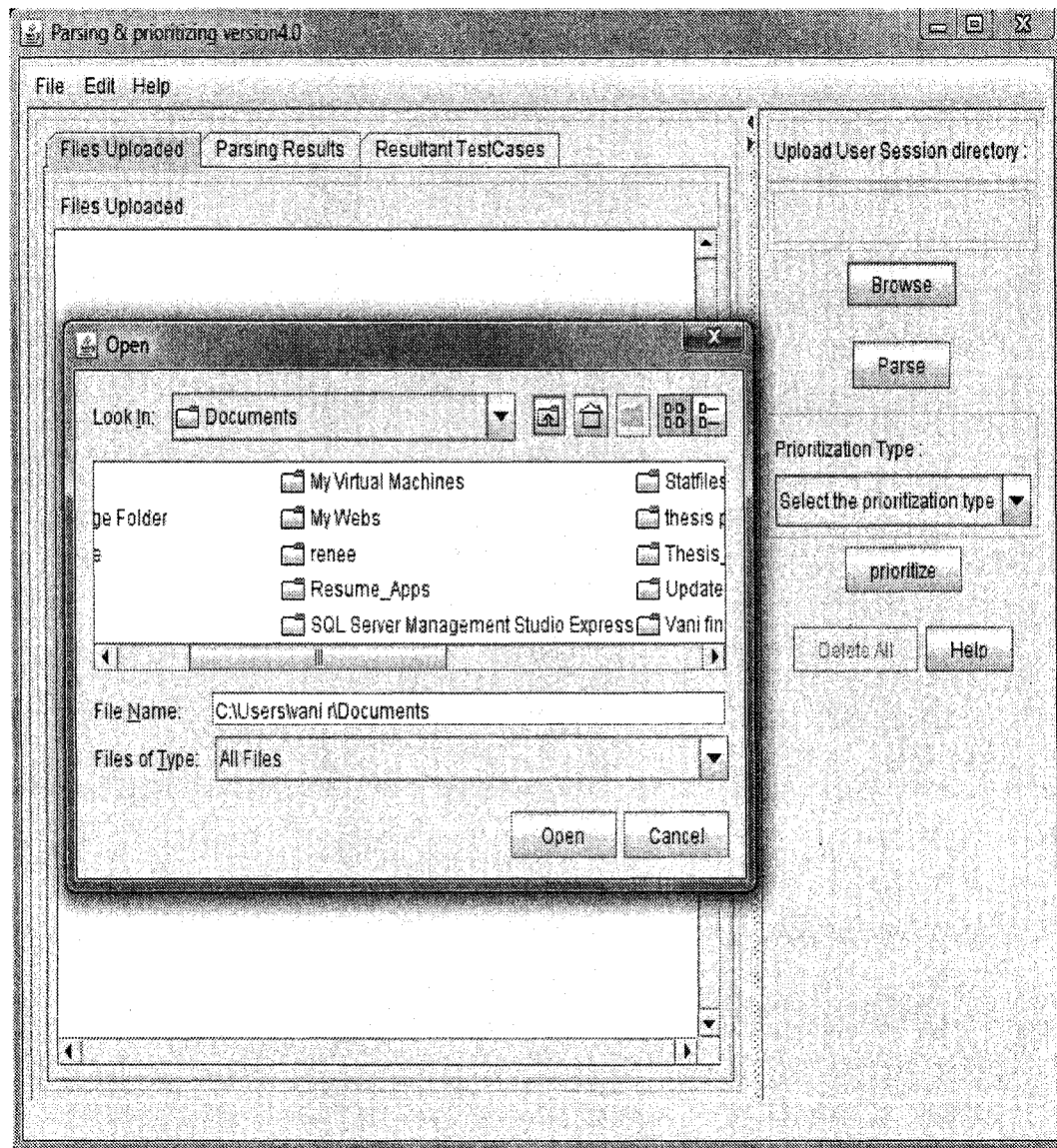


Figure 6. Uploading the User-session directory

Files Uploaded

List of all the files uploaded will be displayed in the “Files Uploaded” field provided; it also displays the path of the directory where the user-sessions reside as shown in Figure 7.

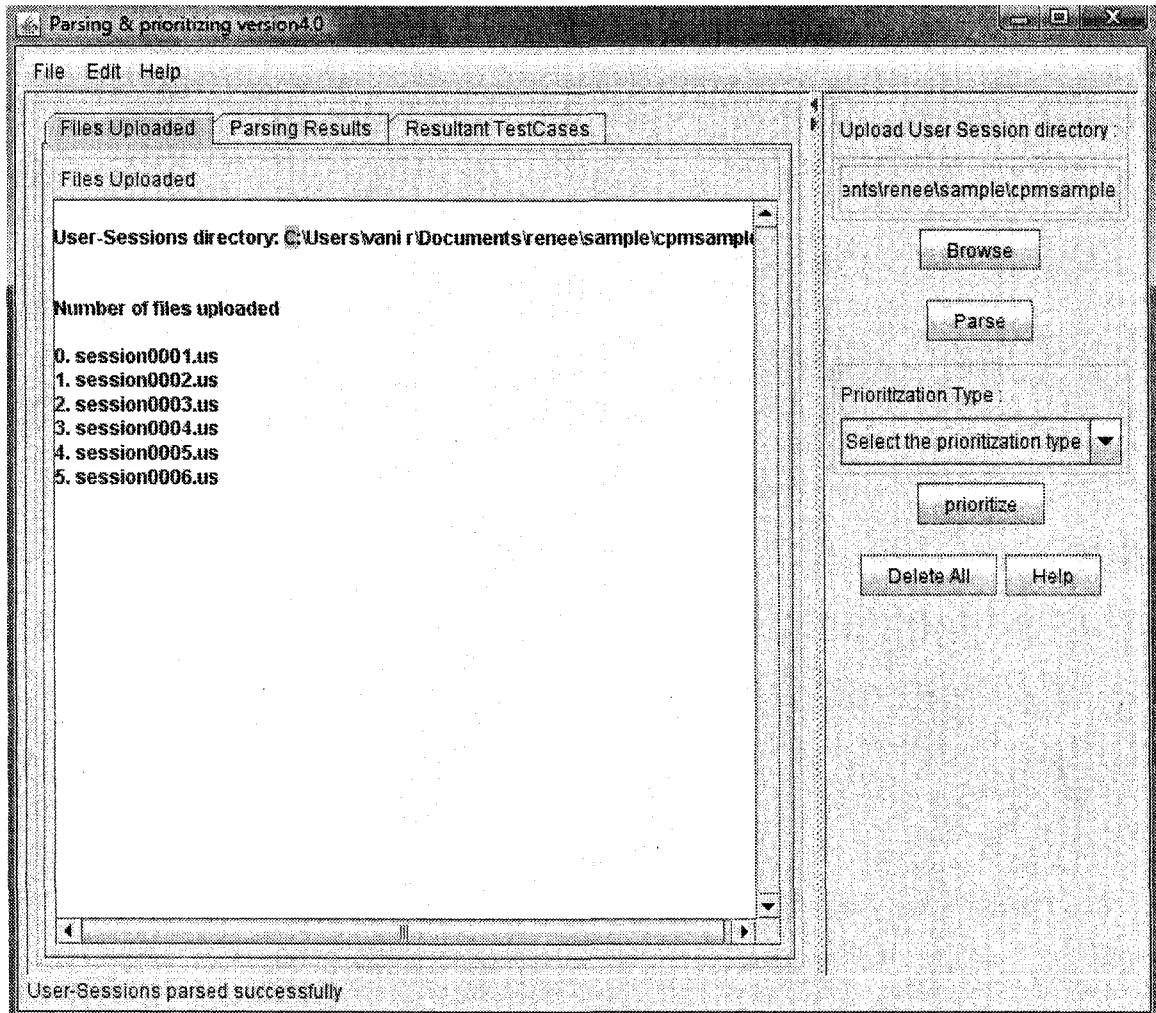


Figure 7. Files uploaded for parsing

Parsing result

The result after parsing the user -sessions and computing the number of options of each page and assigning the unique identifier to each of the page and the parameter-value pairs is displayed in the "Parsing Result" field as shown in Figure 8.

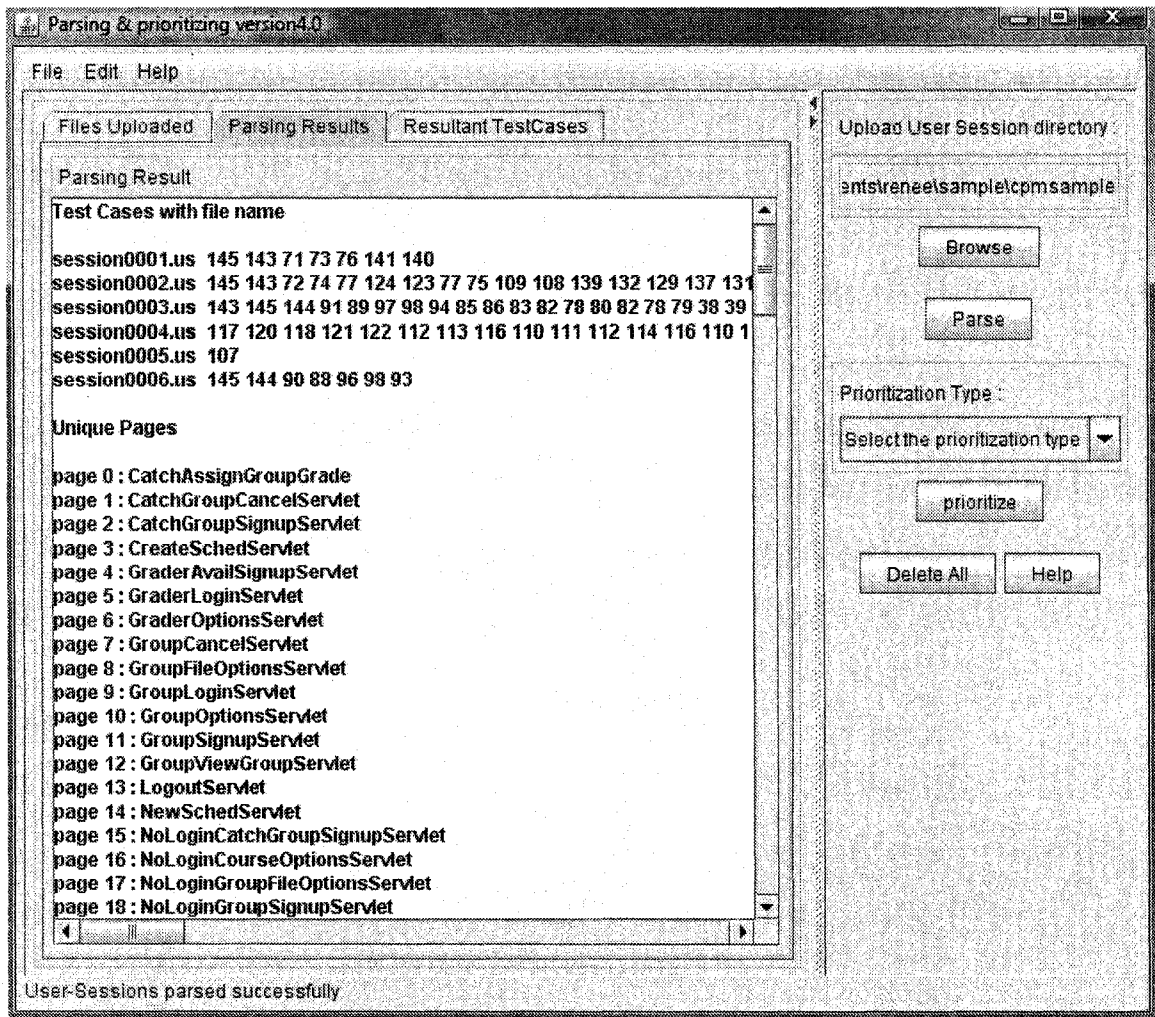


Figure 8. Display of the parsing Result

Resultant test cases

The actual test cases, the sequence of numbers of the parameter-value pairs generated as a result of parsing the user-session, which will be the input to the prioritization tool, will be displayed in the “Resultant TestCases” field as shown in Figure 9.

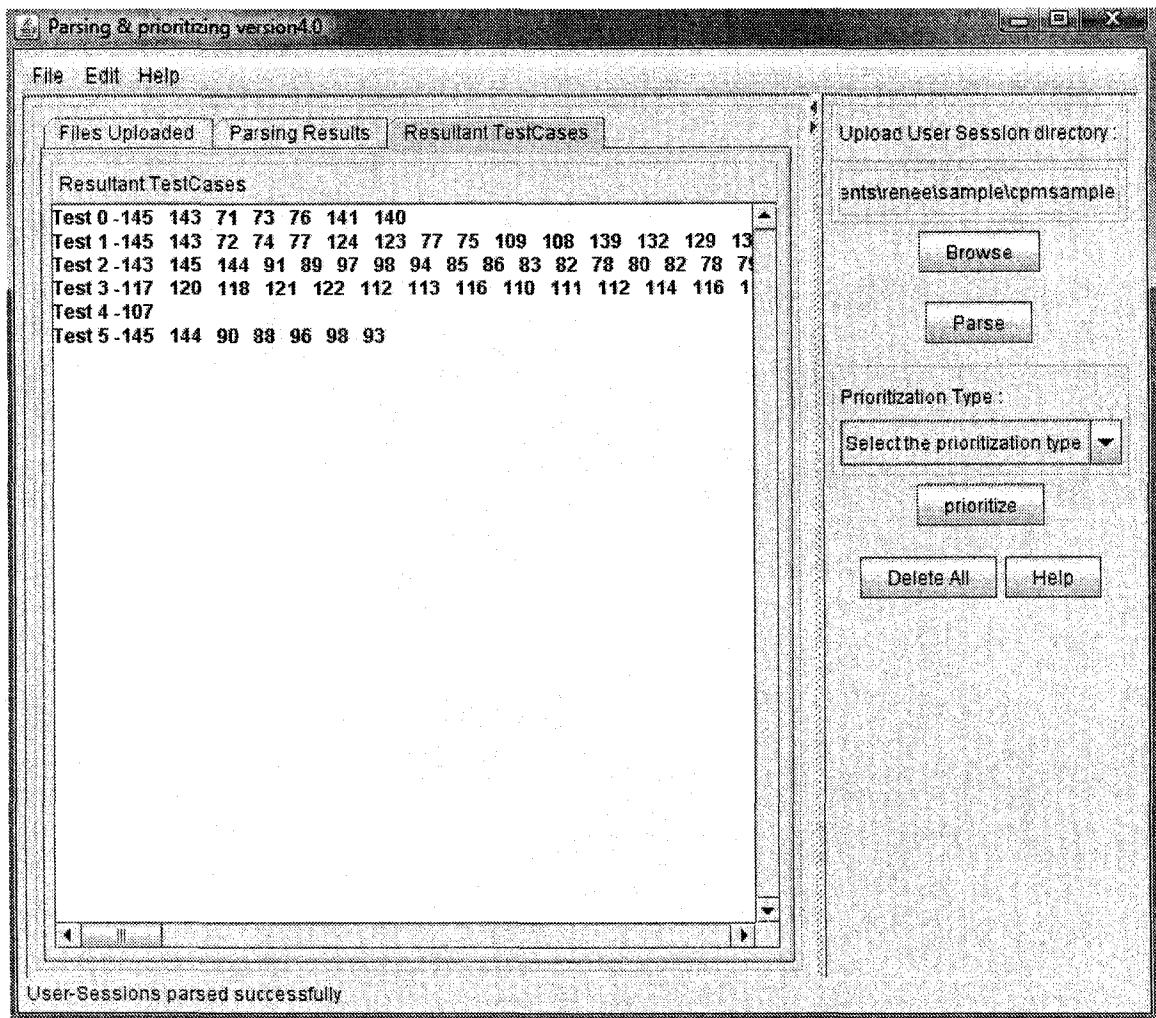


Figure 9. Display of the Resultant test cases

If files other than the extension “.wget” or “.us” or “.tc” are uploaded than an error message will be displayed as shown in Figure 10.

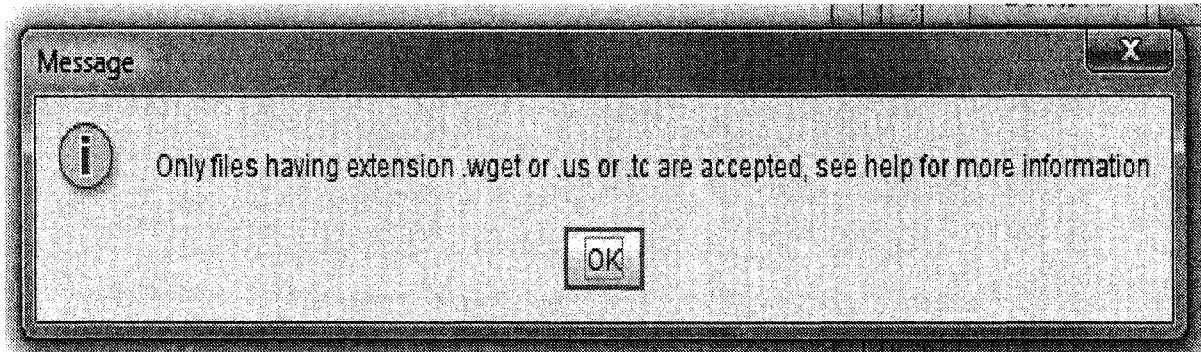


Figure 10. Error message when uploaded wrong directory

Prioritizing the test set depending upon some criteria

Presently the tool supports the prioritization by length in 3 ways

1. Test length -- Longest to Shortest
2. Test length – Shortest to Longest
3. Random

We can prioritize by selecting the prioritization criteria from the drop down box “Prioritization Type” and pressing the “Prioritize” button as shown in the figure below

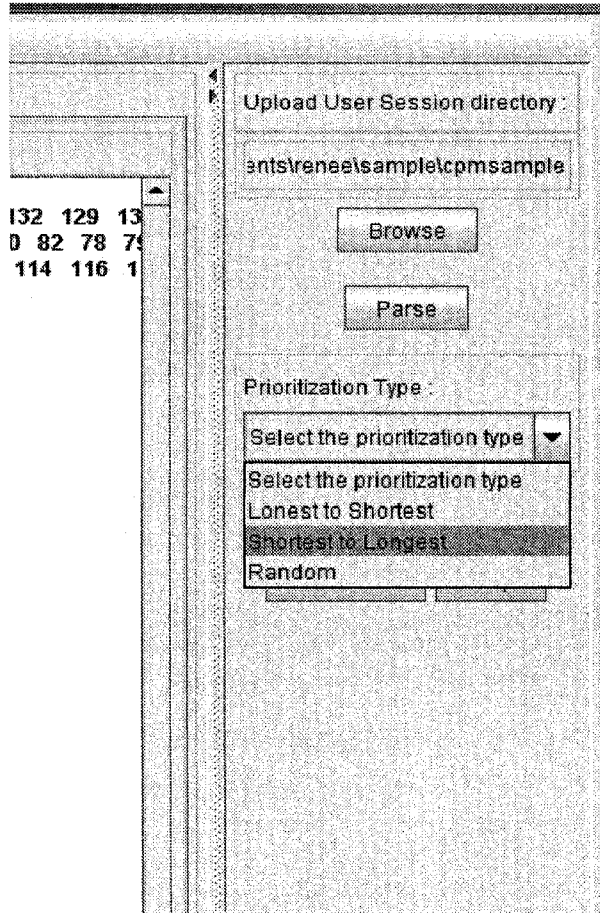


Figure 11. Selecting the Prioritization type

The resultant test cases after prioritization are displayed in the “Resultant TestCases” field provided as shown in Figure 12 and Figure 13.

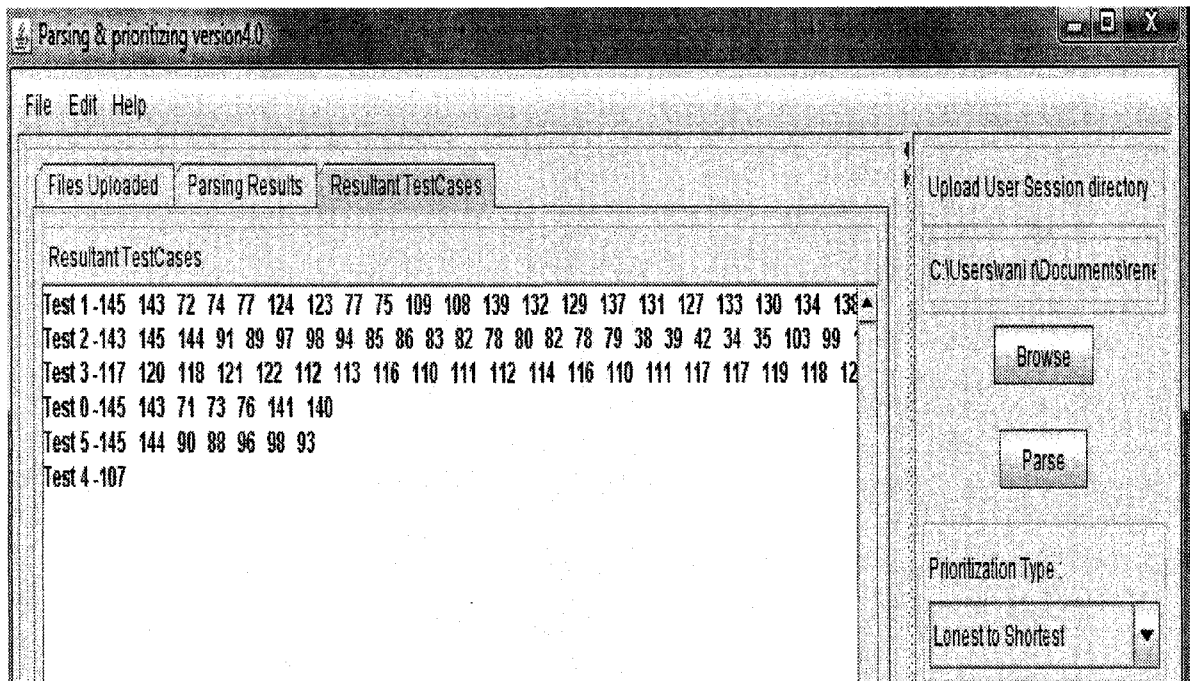


Figure 12. Test cases prioritized by length Longest to Shortest

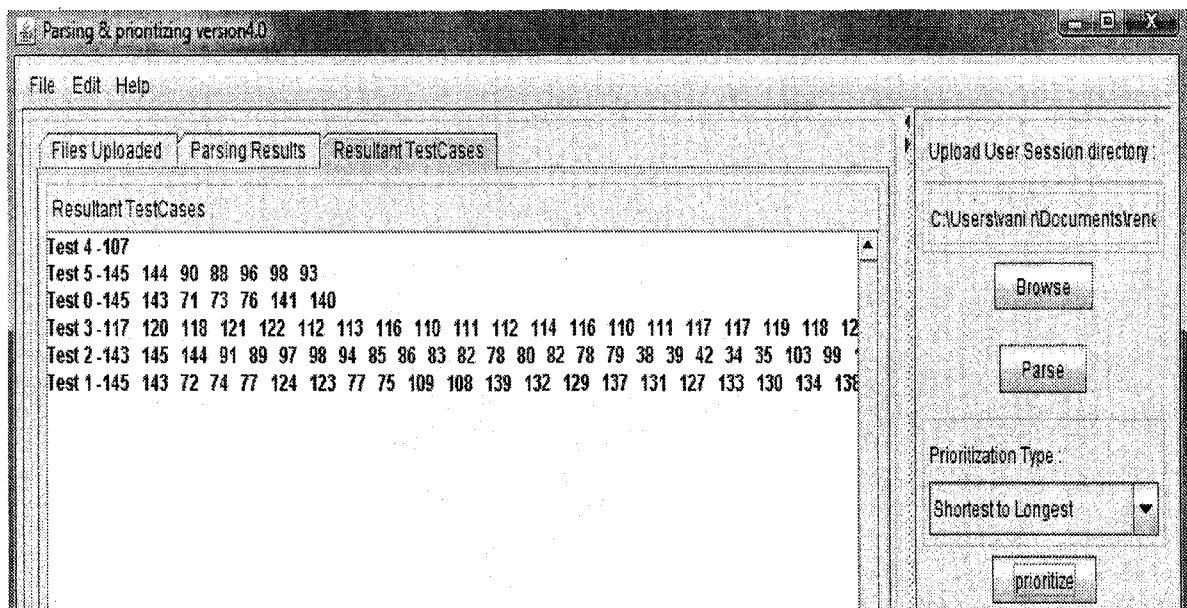


Figure 13. Test cases prioritized by length Shortest to Longest

A Status bar is provided at the bottom of the window which displays the status of parsing and prioritizing.

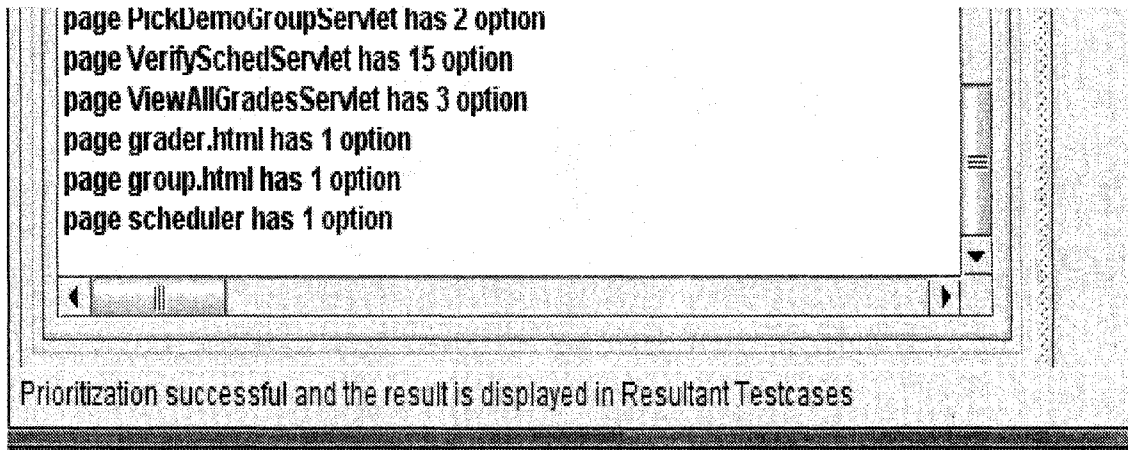


Figure 14. Status Bar, displaying the status of operation

Every computation in the tool can be cleared by clicking “Delete All” button, which need the user confirmation as shown below.

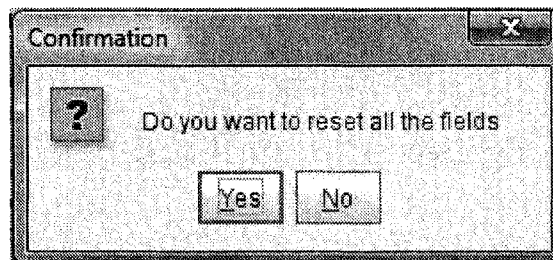


Figure 15. Confirmation window to clear all the fields

Help is provided to the user to guide the tester through the process of parsing and prioritizing.

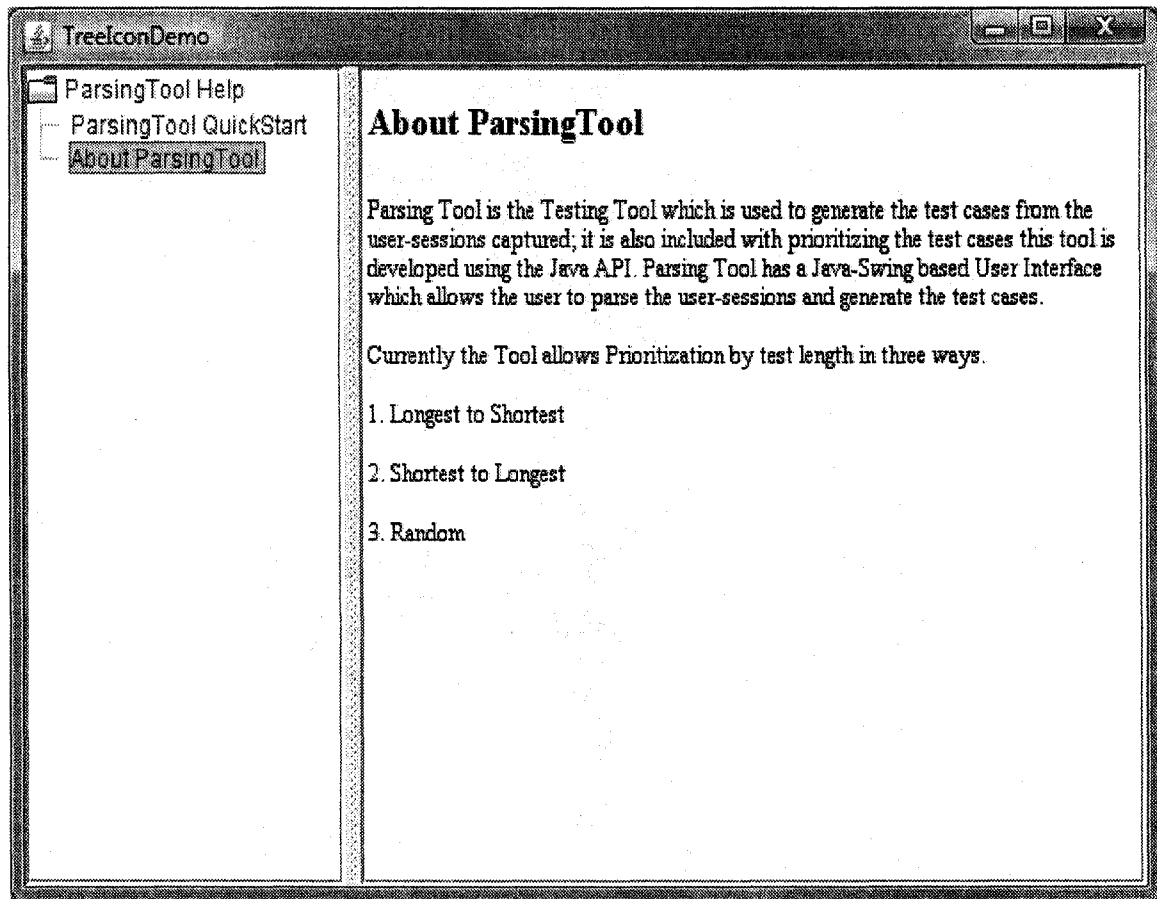


Figure 16. Help window with Quick start guide

The test cases can be directly uploaded into the tool for prioritizing by File → open/upload; the uploaded test cases are shown in the “Resultant TestCases” filed.

To save the resultant test cases, select menu File → Save, give the file name and the location where you intend to save the file. File→ Exit will exit the tool.

Select → Help will bring up the help document, explaining about the tool.

In future work we will consider adding more prioritization techniques in the tool and more features for generating the test cases from scratch.

4.2. Experiments

In the experiments conducted, the effectiveness of each of the prioritization strategies were studied by evaluating their rate of fault detection.

Independent and Dependent Variables. Here the user-session based test suites, prioritization strategies and the faults seeded into the test are considered to be the Independent variables, and the rate of fault detection, average percentage faults detected (APFD)[24] , and the test execution times are the Dependent variables.

Subject Applications and Test Suites. Three web based applications, along with their pre-existing test suites, where the test suites are the previously recorded user-sessions (for experiments in Sampath et al. [26] and Sprenkle et al. [28]) were used for evaluating the proposed prioritization strategies. The subject applications 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). Test suite characteristics and subject programs of the three web applications are shown in Table 10.

Book. Book is a web application which allows users to browse for books, search for a particular book by keyword, rate the books, and purchase the books by adding them to their shopping cart. The users are even allowed to register, login, modify their personal information and

logout. The Book application was designed using JSP for the front-end and MySQL for the back-end database. Since the experiments intend to test the consumer functionality, the administrator code is not included in testing [26]. Sampath et al. [26], by sending emails to local newsgroups and by posting advertisements in the University of Delaware's classifieds web page asking for volunteer users, collected about 125 test cases.

CPM. CPM is the application designed at University of Delaware, which allows course instructors to login and create *grader* accounts for teaching assistants. In turn the Instructors and teaching assistants create *group* accounts for students, assign grades, and create schedules for demonstrated time slots. CPM was designed using Java Servlets and JSPs and the user interface is generated by HTML. It manages the state in a file-based data store. 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 was a web application designed for managing the regional workshop at University of Delaware. 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. Prioritization techniques that are evaluated assume that the tester is aware of the prior knowledge of the faults detected by the regression test suites. As discussed previously these techniques are evaluated with respect to their rate of fault detection, the average percentage 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. The *average percent of faults detected (APFD)* is defined using the notation in [24].

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 the experiments conducted, *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 are the main concerns.*

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	1415	4146	645
% of 2-way parameter-value interactions Covered in pre-existing test suite	92.5%	97.8%	96.2%

Table 10. Subject Applications and Test Suite Characteristics

Experimental Methodology. The 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, is already available from the previous experiments conducted by Sampath et al. [26] and Sprenkle et al. [28, 29]. The fault matrices used are generated by using the *struct* oracle for CPM and Masplas and the *diff* oracle for Book [28, 29]. In addition to seeding some naturally occurring faults found during the deployment there are some faults manually seeded in the applications by the graduate and the undergraduate students, as described in [26, 28]. In general, there are five types of faults 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 (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].

The implementations of the prioritization techniques are as described in Chapter 3. In case of a tie between two or more tests that meet the prioritization criterion, a random tiebreaking strategy is implemented. To account for the non-determinism introduced by random tie breaking, each of the prioritization techniques is executed five times and the average rate of fault detection, APFD, is reported.

CHAPTER 5

EXPERIMENTAL RESULTS

5.1. CPM

The results for CPM are shown in Table 11. For CPM, the results for the length based on number of base requests (Req-LtoS, Req-StoL), Random and also the rate of fault detection for parameter-value interaction (1-way, 2-way, PV-LtoS, PV-StoL), Random are shown in Table 11.

Table 11 shows the APFD in 10% increments of the number of executed tests. The prioritization techniques with highest APFD for the corresponding percentage of the test suites executed are shown in bold-faced numbers. The same notations are used for showing the results in Masplas and Book.

Finding the most faults in the earliest tests. Prioritization by 2-way parameter-value interaction coverage is the most effective technique as shown in Table 11.

Locating 100% of the faults earliest. After the first 10% of tests are run, the 2-way parameter-value interaction coverage has the fastest rate

of fault detection in the rest of the 90% of test suite. The technique Prioritization by the length based on number of requests – shortest to longest, PV-StoL, is the least effective one. The remaining prioritization techniques fall in between these best and worst cases of APFD. For instance, prioritizations by 1-way and by PV-LtoS are generally the second most effective techniques in the latter 90% of the tests run. Prioritization Random, Req-StoL, and Req-LtoS are less effective than the other techniques.

5.2. MASPLAS

Finding the most faults in the earliest tests. If APFD during the first 30% of the test suite is of primary concern, prioritization by Req-LtoS is the most effective as shown in Table 12.

Locating 100% of the faults earliest. After executing the first 30% of the test suite, the remaining 70% of the test suites has the best APFD if prioritized by 2-way. It can be seen from Table 12 that in the last 70% of the test suite, Req-LtoS and 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.

% of test suite run	LtoS	StoL	Random	1-way	2-way	PV-LtoS	PV-StoL
10	78.17	75.14	48.63	83.79	83.72	83.53	16.38
20	80.34	77.76	57.55	87.78	90.8	88.77	25.6
30	81.77	80.27	64.51	91.54	91.72	88.77	26.44
40	84.58	81.39	69.19	94.79	95.64	92.71	28.76
50	85.58	82.95	73.03	94.79	95.64	92.71	30.33
60	87.14	84.44	75.37	94.79	95.64	94.26	34.64
70	87.74	85.15	77.37	94.79	95.64	94.26	39.15
80	88.27	86.21	78.24	94.79	95.64	94.26	39.58
90	88.3	86.31	78.45	94.99	95.64	94.26	42.18
100	88.36	86.35	78.49	94.99	95.64	94.26	43.09

Table 11 - APFD for CPM (in percentage)

% of test suite run	LtoS	StoL	Random	1-way	2-way	PV-LtoS	PV-StoL
10	95.12	81.5	76.33	89.6	90.98	86.05	4.44
20	95.12	91.06	80.51	93.04	90.98	89.74	4.44
30	95.12	91.06	85.57	93.04	94.28	89.74	26.61
40	95.68	91.59	87.59	95.56	97.06	93.38	30.08
50	95.68	91.59	89.91	95.56	97.06	94.84	50.16
60	95.68	91.59	90.69	95.56	97.06	94.84	53.91
70	95.97	91.89	90.69	95.56	97.06	94.84	57
80	96.14	92.08	90.91	95.56	97.06	94.84	58.1
90	96.22	92.17	90.91	95.56	97.06	94.84	58.85
100	96.22	92.2	90.91	95.56	97.06	94.84	58.85

Table 12 - APFD for MASPLAS (in percentage)

5.3. BOOKS

In Books, for finding the most faults in the earliest tests, prioritization by 1-way has proven to be the best for the first 20% of the test suite execution as shown in Table 13. Prioritization by PV- StoL and Req-StoL are the slow starters during the first 10% of the test 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.

Locating 100% of the faults earliest. Table 13 shows that prioritization by 1-way has a high APFD.

Fault Detection Density. From Table 13, It can be noted that Random creates a reasonably effective test order with APFD comparable to the other techniques.

If the execution time is of primary concern then 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.

% of test suite run	LtoS	StoL	Random	1-way	2-way	PV-LtoS	PV-StoL
10	92.96	70.04	90.34	93.44	93.22	93.11	70.13
20	92.96	86.09	93.7	93.44	93.22	93.11	70.13
30	92.96	88.15	94.52	93.44	93.22	93.11	78.17
40	92.96	88.91	94.86	93.44	93.22	93.11	79.86
50	92.96	88.91	94.86	94.96	94.69	93.11	84.12
60	92.96	89.15	95.11	96.13	94.69	94.47	86.73
70	93.74	89.54	95.27	96.13	95.62	95.56	86.73
80	94.11	89.81	95.56	96.13	95.62	95.56	86.73
90	94.18	89.92	95.56	96.13	95.62	95.56	86.73
100	94.27	89.94	95.57	96.13	95.62	95.56	86.73

Table 13 - Book APFD

Table 14 shows the execution time, time taken to reply test suite. Execution time does not include the time taken to detect faults, i.e., fault detection replay.

Application	1-way	2-way	PV-LtoS	PV-StoL
CPM	83.26(813)	38.88(618)	58.20(746)	100(889)
MASPLAS	93.44	33.73(36)	42.01	97.04(71)
BOOKS	57.60(907)	66.40(1024)	60.80(1002)	54.40(300)

Table 14 – Percent of Test Suite Run (Execution time in seconds) for 100% Fault Detection

CHAPTER 6

SUMMARY AND CONCLUSION

6.1. Summary of Results

The results from the experiments show that none of the prioritization techniques is clearly the “the winner” for all three of the web applications tested. However, for two of our three applications, prioritization techniques that consider parameter-value counts or interactions find 100% of faults before the other techniques. This study shows that 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 as shown in Table 14. In both these applications, 2-way has the highest APFD values overall (after 100% of the test suite is executed). In Book, however, 1-way has the highest overall APFD.

In CPM, prioritization by 2-way parameter-value interaction coverage is generally the most effective. In Masplas, for the first 30% of the test suite Req-LtoS is the best technique and for the remaining 70% of the test suite, giving preference to covering every 2-way parameter-value interaction creates the most effective test suite ordering for finding the

best APFD. In Book, for achieving a good rate of fault detection early (first 10%) in the test cycle, choosing any of the metrics other than Req-StoL, PVStoL, or Random will be good options. However, for achieving 100% fault detection with the smallest test number and with low APFD of overall execution of tests, PV-StoL is the best prioritization technique. Though random appears to create an effective test suite ordering for books, for large number of test cases and low fault detection densities, Random's effectiveness will decrease. Parameter-value interaction coverage and frequency-based techniques can detect more faults early in the test execution cycle.

If we observe the execution time of the tests, 2- way detects 100% of the faults 30% faster than the worst technique, PV-StoL, in CPM, and in Masplas 2-way detects 100% of the faults 40% faster than the worst technique PV-StoL, whereas 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.

6.2. Conclusion

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. This thesis involves studying prioritization of such user-sessions for three web

applications. Several new prioritization criteria are applied to these test suites to identify whether they can be used to increase the rate of fault detection. The experimental results suggest that prioritization by frequency metrics and systematic coverage of parameter-value interactions may increase the rate of fault detection for web applications. Since the conclusion is not clear and there is no clear winner in the prioritization techniques, future work needs to examine additional web based applications, test suites, and prioritization techniques. We can also focus on the hybrid prioritization technique which includes prioritizing by more than one technique for one application.

BIBLIOGRAPHY

- [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-sessionbased 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.
- [31] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 230–238, November 1997.

VITA

Graduate College
University of Nevada, Las Vegas

Vani Kandimalla

Local Address:

4223 Cottage Circle, Apt# 3
Las Vegas, NV – 89119

Degree:

Bachelor of Engineering in Electronics & Communication, 2004
JNTU, Hyderabad, India

Selected Publications:

- S. Sampath, R. Bryce, Gokulanand Viswanath, Vani Kandimalla, A. Gunes Koru. Prioritizing User-Session-Based Test Cases for Web Applications Testing. *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, Lillehammer, Norway, April 2008, pp. 141-150

Thesis Title:

Test Suite Prioritization Techniques applied to Web-Based Applications

Thesis Examination Committee:

Chairperson, Dr. Renee Bryce, Ph.D
Committee Member, Dr. John Minor, Ph.D
Committee Member, Dr. Evangelos Yfantis, Ph.D
Graduate College Representative, Dr. Fatma Nasoz, Ph.D