University of Denver
Digital Commons @ DU

Electronic Theses and Dissertations | Graduate Studies

3-2023

A Unified Approach to Regression Testing for Mobile Apps

Zeinab Saad Abdalla
University of Denver

Follow this and additional works at: https://digitalcommons.du.edu/etd

Part of the Software Engineering Commons

Recommended Citation

Abdalla, Zeinab Saad, "A Unified Approach to Regression Testing for Mobile Apps" (2023). *Electronic Theses and Dissertations*. 2170.
https://digitalcommons.du.edu/etd/2170

This Dissertation is brought to you for free and open access by the Graduate Studies at Digital Commons @ DU. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of Digital Commons @ DU. For more information, please contact jennifer.cox@du.edu,dig-commons@du.edu.

A Unified Approach to Regression Testing for Mobile Apps

Abstract

Mobile Applications have been widely used in recent years daily all over the world and are essential in our personal lives and at work. Because Mobile Applications update frequently, it is important that developers perform regression testing to ensure their quality. In addition, the Mobile Applications market has been growing rapidly, allowing anyone to write and publish an application without appropriate validation. A need for regression testing has arisen with the growth of different Mobile Apps and the added functionalities and complexities. In this dissertation, we adapted the FSMWeb [14] approach for selective regression testing to allow for selective regression testing of Mobile Apps. We applied rules to classify the original set of tests of the Mobile App into obsolete, retestable, and reusable tests based on the types of changes to the model of Mobile Apps. New tests are added to cover portions that have not been tested.

As regression test suites change, we want to ensure that required tests are included to satisfy testing criteria, but also that redundant tests are removed, so as not to bloat the regression tests suite. In the dissertation, we developed a test case minimization approach for FSMApp, based on concept analysis that removes redundant test cases.

Next, we proposed an approach to prioritize test cases for Mobile Apps. Naturally, it is desirable to select those test cases that are most likely to reveal defects in the App under test. We prioritized test paths for Mobile Apps based on input complexity, since more inputs might be associated with a more complex functionality which in turn would make it more fault-prone.

As we knew, regression testing is an important activity in software maintenance and enhancement. Combining several regression testing techniques can lead to a more efficient and effective regression test suite. In this dissertation, we presented guidelines for combining regression testing approaches based on a systematic approach. We outlined all possible situations that can occur and showed how each of them influences which combination to use.

Also, we validated the newly proposed regression testing approaches for Mobile Apps and the guidelines for combining regression testing approaches via a case study. The results show that FSMApp approaches are applicable, efficient, and effective.

Document Type

Dissertation

Degree Name

Ph.D.

Department

Computer Science

First Advisor

Kerstin Haring

Second Advisor

Anneliese Andrews

Third Advisor
Chris GauthierDickey

Keywords
Mobile applications, Regression testing

Subject Categories
Computer Sciences | Physical Sciences and Mathematics | Software Engineering

Publication Statement
Copyright is held by the author. User is responsible for all copyright compliance.

This dissertation is available at Digital Commons @ DU: https://digitalcommons.du.edu/etd/2170

A Unified Approach to Regression Testing for Mobile Apps

A Dissertation
Presented to
the Faculty of the Daniel Felix Ritchie School of
Engineering and Computer Science
University of Denver

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

---

by
Zeinab Saad Abdalla
March 2023
Advisor: Dr. Kerstin Haring

© Copyright by Zeinab Saad Abdalla, 2023

All Rights Reserved

Author: Zeinab Saad Abdalla
Title: A Unified Approach to Regression Testing for Mobile Apps
Advisor: Dr. Kerstin Haring
Degree Date: March 2023

Abstract

Mobile Applications have been widely used in recent years daily all over the world and are essential in our personal lives and at work. Because Mobile Applications update frequently, it is important that developers perform regression testing to ensure their quality. In addition, the Mobile Applications market has been growing rapidly, allowing anyone to write and publish an application without appropriate validation. A need for regression testing has arisen with the growth of different Mobile Apps and the added functionalities and complexities. In this dissertation, we adapted the FSMWeb [14] approach for selective regression testing to allow for selective regression testing of Mobile Apps. We applied rules to classify the original set of tests of the Mobile App into obsolete, retestable, and reusable tests based on the types of changes to the model of Mobile Apps. New tests are added to cover portions that have not been tested.

As regression test suites change, we want to ensure that required tests are included to satisfy testing criteria, but also that redundant tests are removed, so as not to bloat the regression tests suite. In the dissertation, we developed a test case minimization approach for FSMApp, based on concept analysis that removes redundant test cases.

Next, we proposed an approach to prioritize test cases for Mobile Apps. Naturally, it is desirable to select those test cases that are most likely to reveal defects in the App under test. We prioritized test paths for Mobile Apps based on input com-

plexity, since more inputs might be associated with a more complex functionality which in turn would make it more fault-prone.

As we knew, regression testing is an important activity in software maintenance and enhancement. Combining several regression testing techniques can lead to a more efficient and effective regression test suite. In this dissertation, we presented guidelines for combining regression testing approaches based on a systematic approach. We outlined all possible situations that can occur and showed how each of them influences which combination to use.

Also, we validated the newly proposed regression testing approaches for Mobile Apps and the guidelines for combining regression testing approaches via a case study. The results show that FSMApp approaches are applicable, efficient, and effective.

Acknowledgements

I would like to express my gratitude to my advisor, Professor Anneliese Andrews for the unlimited support and guidance she provided in my PhD study journey and conducting research. You have invested a great effort in me and you have all my respect and appreciation. Thank you so very much.

Special thanks to my second advisor Dr. Kerstin Haring for the support and guidance she provided in the last year of my PhD study, and also for her believing in me being a research assistant.

I would also like to thank Dr. Chris GauthierDickey and Dr. Chip Reichardt for serving as my dissertation defense committee members.

Special thanks to the DiscoverU team, especially Dr. Daniel Pittman and Lombe Chileshe for believing in me as a software tester for their project and for all the data they gave me to help validate my work.

Also, I would like to express my sincere gratitude to the Computer Science Department faculty and staff for their support.

Special thanks to my parents, my husband, kids, siblings, friends, and colleagues for their fantastic support during my academic journey.

Table of Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 1
- 1.2 Research Scope . . . 5
- 1.3 Research Agenda and Contribution . . . 6

2 Background **9**
- 2.1 Black-Box Model-Based Testing . . . 9
  - 2.1.1 MBT Techniques for Mobile Apps . . . 11
  - 2.1.2 Black-Box Testing with FSMApp . . . 17
- 2.2 Regression Testing . . . 25
  - 2.2.1 Selective Regression Testing . . . 26
  - 2.2.2 Test Case Minimization . . . 28
  - 2.2.3 Test Case Prioritization . . . 32
  - 2.2.4 Regression Testing Techniques for Mobile Apps . . . 42
  - 2.2.5 Combination of Regression Testing Approaches . . . 45

3 A Unified Approach to Regression Testing **50**
- 3.1 Selective Regression Testing of Mobile Apps . . . 52
  - 3.1.1 Selective Regression Testing Process . . . 54
  - 3.1.2 Example Used to Illustrate Approach . . . 59
- 3.2 Test Case Minimization . . . 81
  - 3.2.1 Determine requirements and test cases based on selective regression testing for FSMApp . . . 82
  - 3.2.2 Apply Concept Analysis . . . 85
  - 3.2.3 Example to Illustrate Test Case Minimization of FSMApp using Concept Analysis . . . 90
- 3.3 Test Case Prioritization . . . 98
  - 3.3.1 Test Case Prioritization Process . . . 99
  - 3.3.2 Example Used to Illustrate Approach . . . 100
  - 3.3.3 Example to Combining Test Case Prioritization with Selective Regression Testing and Test Case Minimization . . . 112

4 Guidelines for Combining Regression Testing Approaches **115**
4.1 Combination of Regression Testing Strategies 117
4.2 Situations for Combining Regression Testing Approaches 120
4.3 Example for Using The Guidelines to Combining Three Types of Regression Testing (Selective, Minimization, and Prioritization) 125

5 Validation via Case Study **132**
5.1 Rationale for the Case Study 132
5.2 Case Study Objective 133
5.3 Preparation for Case Study 134
5.4 Case Study Research Questions 134
5.5 Units of Analysis 135
5.6 Case Study General Descriptions 136
5.6.1 First Version of the DiscoverU App ($DiscoverU_{V1}$) 138
5.6.2 Second Version of the DiscoverU App ($DiscoverU_{V2}$) 141
5.6.3 Third Version of the DiscoverU App $DiscoverU_{V3}(APK1)$ . . 142
5.6.4 Fourth Version of DiscoverU App $DiscoverU_{V4}(APK2)$ 144
5.7 Validation Results and Discussion 146
5.7.1 Applicability 146
5.7.2 Effectiveness 148
5.7.3 Efficiency 149
5.8 Threats to Validity 155
5.9 Challenges and Successes in our Case Study 156

6 Future Work **158**
6.1 Minimizing Test Cases with Guarantee of the Aggregation Test Requirements 159
6.2 Prioritizing Test Cases by weighting the Input Types 159
6.3 Applying our Proposed Approaches and Guidelines in Different System Domains 160
6.4 Building Tools 160
6.5 Finish and Execute Test Paths for the Case Study DiscoverU App Automatically 161

7 Conclusion **162**

Bibliography **167**

A Components of Android Apps **185**

B DiscoverU Case Study Screens **192**

C DiscoverU Mobile App **202**
C.1 Clusters and Nodes 202

D DiscoverU Mobile App FSM Model **216**
D.1 DiscoverU Mobile App HFSM Model 216
D.2 DiscoverU App Case Study Test Paths 233
D.3 DiscoverU App Inputs and Complexity of each Test path 239

List of Tables

1.1 Publication 8

2.1 Classification of Prioritization Regression Testing Techniques. 34
2.2 Approach comparison 44
2.3 Summary of existing combination approaches to regression testing . . 46

3.1 The Notations and Conventions 56
3.2 Transitions of Figure 3.3 (Main Page Cluster AFSM) 64
3.3 Transitions of Figure 3.4 (Modify task Cluster) 65
3.4 Transitions of Figure 3.5 (Add task Cluster) 65
3.5 Transitions of Figure 3.6 (Edit task Cluster) 66
3.6 Main Page Test Sequences of Figure 3.3 66
3.7 Modify Task cluster Test Sequences of Figure 3.4 67
3.8 Add Task cluster Test Sequences of Figure 3.5 67
3.9 Edit Task Cluster Test Sequences of Figure 3.6 67
3.10 The aggregated Test Paths of the To Do App 68
3.11 Test Path 1 with Input Values 69
3.12 Main Page Test Sequences of Figure 3.8 76
3.13 Task cluster Test Sequences of Figure 3.9 76
3.14 Add Task cluster Test Sequences of Figure 3.10 77
3.15 Record Task cluster Test Sequences of Figure 3.11 77
3.16 The aggregated test paths of the To Do app 79
3.17 New Test Path 1 with Input Values 80
3.18 The abstract test paths of the To Do app 92
3.19 Test Requirements of To Do App 93
3.20 Context Table 94
3.21 The Table of Concepts 94
3.22 Context Table 96
3.23 Reduced Concepts 96
3.24 Reduced Context Table 97
3.25 Reduced Concepts 97
3.26 The original Test Paths of the To Do App 102
3.27 The Input Types of the To Do App 103

3.28 The selected and new test paths of the To Do app 106
3.29 The Complexity of the Test Path 1 108
3.30 The Complexity of the Test Path 2 109
3.31 The Complexity of the Test Path 3 109
3.32 The Complexity of the Test Path 4 110
3.33 The Complexity of the Test Path 5 110
3.34 The Complexity of the Test Path 6 111
3.35 The priority of the Test Paths 112
3.36 The minimized Test Paths 113
3.37 The priority of the Test Paths 113

4.1 Strategies of Regression Testing Approaches 120
4.2 Situations for Combining Regression Testing Approaches 121
4.3 The selected and new test paths of the To Do app 128
4.4 The minimized Test Paths 129
4.5 The Complexity of the Test Path 1 130
4.6 The priority of the Test Paths 131

5.1 Units of Analysis 136
5.2 Type of used techniques and the situations for each version 147
5.3 Defect Summary 149
5.4 Summary of the Changes 150
5.5 Model Size Summary 151
5.6 Summary of Test Generation and Execution 153
5.7 Summary of Reducing and Minimizing Test Cases 154

A.1 Components of Mobile Application (LAPs) 186

C.1 Clusters and Nodes For **Main** 202
C.2 Clusters and Nodes For **Basecamp** 203
C.3 Clusters and Nodes For **Explore** 203
C.4 Clusters and Nodes For **Journey** 203
C.5 Clusters and Nodes For **Backpack** 204
C.6 Clusters and Nodes For **Menu** 204
C.7 Clusters and Nodes For **How are feeling** 205
C.8 Clusters and Nodes For **Check in Now** 205
C.9 Clusters and Nodes For **Next** 205
C.10 Clusters and Nodes For **Free Write** 205
C.11 Clusters and Nodes For **Add Free Write** 206
C.12 Clusters and Nodes For **Delete** 206
C.13 Clusters and Nodes For **All Content** 206
C.14 Clusters and Nodes For **Filter** 206
C.15 Clusters and Nodes For **Journey** 207

C.16 Clusters and Nodes For **All Journey** . . . 207
C.17 Clusters and Nodes For **Rewords** . . . 207
C.18 Clusters and Nodes For **Resources** . . . 207
C.19 Clusters and Nodes For **Menu-Favourites** . . . 208
C.20 Clusters and Nodes For **Menu-favourites-Basecamp** . . . 208
C.21 Clusters and Nodes For **Menu-Backpack** . . . 208
C.22 Clusters and Nodes For **Menu-Backpack-Backpack** . . . 209
C.23 Clusters and Nodes For **Menu-Tools** . . . 209
C.24 Clusters and Nodes For **Menu-Tools-Journey** . . . 209
C.25 Clusters and Nodes For **Menu-sub-menu** . . . 209
C.26 Clusters and Nodes For **Sub-Menu-Journal-List** . . . 210
C.27 Clusters and Nodes For **Sub-Free Write** . . . 210
C.28 Clusters and Nodes For **sub-AddFreeW** . . . 210
C.29 Clusters and Nodes For **Delete Free Write** . . . 211
C.30 Clusters and Nodes For **Stress Log** . . . 211
C.31 Clusters and Nodes For **Add Stress-Log** . . . 211
C.32 Clusters and Nodes For **Delete Stress-Log** . . . 211
C.33 Clusters and Nodes For **Grief Journal** . . . 212
C.34 Clusters and Nodes For **Add Grief-Journal** . . . 212
C.35 Clusters and Nodes For **Delete Grief-Journal** . . . 212
C.36 Clusters and Nodes For **Emotion Reflections** . . . 213
C.37 Clusters and Nodes For Add **Emotion-Reflection** . . . 213
C.38 Clusters and Nodes For **Delete Emotion-Reflection** . . . 213
C.39 Clusters and Nodes For **Check-ins** . . . 214
C.40 Clusters and Nodes For **Week-Check-ins** . . . 214
C.41 Clusters and Nodes For **Menu-Daily Check-in** . . . 214
C.42 Clusters and Nodes For **Menu-Mood Check-in** . . . 214
C.43 Clusters and Nodes For **Menu-Settings** . . . 215
C.44 Clusters and Nodes For **Menu-Logout** . . . 215

D.1 Test Cases for the Versions of the DiscoverU App. . . . 233
D.2 The Inputs and the Complexity of the Test Paths of the DiscoverU App. . . . 239

List of Figures

1.1 Software Maintenance Process 3
1.2 Condition to Determine Combination of Regression Testing 4

3.1 Combining Regression Testing of FSMApp Process 51
3.2 Main Screens for To Do App 60
3.3 Main page (AFSM) 60
3.4 Modify Task Cluster 61
3.5 Add Task Cluster 61
3.6 Edit Task Cluster 62
3.7 Annotated FSM for Add Task Cluster of Table 3.4 64
3.8 Modified Main Page (AFSM) 71
3.9 Task Cluster 71
3.10 Add Task Cluster 72
3.11 Record Task Cluster 72
3.12 Test case minimization using concept analysis 82
3.13 Main page of ToDo App 90
3.14 Sub-clusters of the To Do App 91
3.15 LAPs of the To Do App 91
3.16 Concept Lattice 95
3.17 Concept Lattice: First Minimization 97
3.18 Concept Lattice: Second Minimization 98
3.19 Test Case Prioritization of FSMApp Process 99
3.20 Original Model HFSM for ToDo App 101
3.21 Modified Model $HFSM'$ for ToDo App 104

4.1 Process of Combining the Three Types of Regression Testing 119
4.2 Decision Tree for Combinations of Regression Testing 122
4.3 Modified Model $HFSM'$ for ToDo App 127

B.1 Entry Page for DiscoverU App 192
B.2 Main Screens for the DiscoverU App 193
B.3 How are you feeling Screens for DiscoverU App 194
B.4 Free Write Screens for DiscoverU App 194
B.5 Demo Theme for DiscoverU App 195

B.6 Explore Screens for DiscoverU App . . . 195
B.7 Journey Screens for DiscoverU App . . . 196
B.8 Backpack Screens for DiscoverU App . . . 197
B.9 Menu Main Screens for DiscoverU App . . . 198
B.10 Sub-Menu (Free Write) Screens for DiscoverU App . . . 198
B.11 Sub-Menu (Stress Log) Screens for DiscoverU App . . . 199
B.12 Sub-Menu (Grief Journal) Screens for DiscoverU App . . . 199
B.13 Sub-Menu (Emotion Reflections) Screens for DiscoverU App . . . 199
B.14 Sub-Menu (Goals) Screens for DiscoverU App . . . 200
B.15 Sub-Menu (Check-ins) Screens for DiscoverU App . . . 200
B.16 Menu (Daily Check-in, Settings, Logout) Screens for DiscoverU App . 201

D.1 Main Cluster (AFSM) for DiscoverU App . . . 216
D.2 Basecamp Cluster for DiscoverU App . . . 217
D.3 Explore Cluster for DiscoverU App . . . 217
D.4 Journey Cluster for DiscoverU App . . . 218
D.5 Backpack Cluster for DiscoverU App . . . 218
D.6 Menu Cluster for DiscoverU App . . . 219
D.7 How Are You Feeling? Cluster for DiscoverU App . . . 219
D.8 Check-in Now Cluster for DiscoverU App . . . 220
D.9 Free Write Cluster for DiscoverU App . . . 220
D.10 Add Free Write Cluster for DiscoverU App . . . 221
D.11 Delete Cluster for DiscoverU App . . . 221
D.12 All Content Cluster for DiscoverU App . . . 222
D.13 Filter Cluster for DiscoverU App . . . 222
D.14 Content Cluster for DiscoverU App . . . 223
D.15 All Journeys Cluster for DiscoverU App . . . 223
D.16 Embark Journeys Cluster for DiscoverU App . . . 224
D.17 Trails Cluster for DiscoverU App . . . 224
D.18 Favorites-Backpack Cluster for DiscoverU App . . . 225
D.19 Rewards-Backpack Cluster for DiscoverU App . . . 225
D.20 Resoutces-Menu Cluster for DiscoverU App . . . 226
D.21 Journal-SubMenu Cluster for DiscoverU App . . . 226
D.22 Emotion Reflections Cluster for DiscoverU App . . . 227
D.23 Add Emotion Reflections Cluster for DiscoverU App . . . 227
D.24 Delete Emotion Reflections for DiscoverU App . . . 228
D.25 Grief Journal Cluster for DiscoverU App . . . 228
D.26 Add Grief Journal Cluster for DiscoverU App . . . 229
D.27 Delete Grief Journal for DiscoverU App . . . 229
D.28 Stress Log Cluster for DiscoverU App . . . 230
D.29 Add Stress Log Cluster for DiscoverU App . . . 230
D.30 Delete Stress Log for DiscoverU App . . . 231

D.31 Goals Cluster for DiscoverU App 231
D.32 Check-ins Cluster for DiscoverU App 232
D.33 Check-ins Responses Cluster for DiscoverU App 232

Chapter 1

Introduction

1.1 Problem Statement

Almost all software products undergo maintenance cycles to correct faults, improve performance, add new features, or adapt to a changed environment [34]. The process of modifying a software system or component is called software maintenance. The four types of software maintenance are [34]:

1. *Corrective maintenance*: Modification of a software product to correct defects or improve code quality (e.g through refactoring).

2. *Adaptive maintenance*: Modification of a software product performed to keep a software product compatible with a changed or changing environment.

3. *Perfective maintenance*: Modification of a software product to improve performance or add new features.

4. *Preventive maintenance*: Modification of a software product to detect and correct latent faults before they become effective faults.

Software maintenance includes a variety of tasks such as adding, deleting, extending, and modifying functions, rewriting documentation, improving performance or improving ease of use. Maintenance can also include different processes for correction vs. enhancement, small changes vs. large changes, etc. According to the IEEE standard, any maintenance request (MR) is classified as corrective, preventive, adaptive, or perfective [34]. However, enhancement is not considered direct maintenance. In some circumstances, we can consider enhancements as part of development, in others as maintenance. Development refers to the main implementation of a new project. After the development project is completed and the customer accepted it, the project moves into either maintenance or enhancement as defined above. Research and surveys over the years have shown that over 80% of the maintenance, the effort is related to non-corrective actions [89] [72]. Users may request major enhancements to the system. The ISO Standard on Software Maintenance [89] classifies Adaptive and Perfective maintenance as enhancements and Corrective and Preventive maintenance as corrections.

Figure 1.1 shows the software maintenance process. Regression testing is a major part of software maintenance and enhancement tasks. Regression testing ensures new functionality works as planned and changes did not break existing functionality. There are three types of regression testing, all of which aim to reduce the regression testing burden. They are (1) selective regression testing, (2) test case prioritization, and (3) test case minimization. This dissertation addresses regression testing for Mobile Apps. We present a novel approach that combines all three approaches and provides guidelines on how to apply and combine them based on the characteristics of maintenance or enhancement tasks.

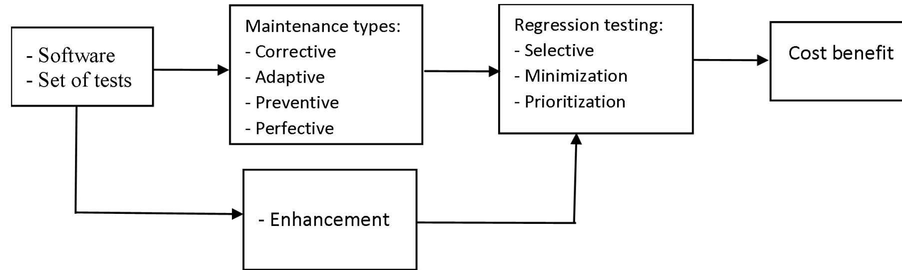


Figure (1.1) Software Maintenance Process

Mobile applications, or Apps, are becoming essential in our personal lives and at work. Mobile Applications like any software, undergo maintenance and enhancement cycles. Mobile Apps also tend to change frequently [9]. This puts a burden on testers since regular regression testing is required. Mobile Applications differ from desktop software in some ways [114]: they interact with other applications, they allow sensor handling via touch screens and cameras, they reside on a multitude of hardware devices and platforms, etc. They also share common technology with other software, especially web applications. Testing Mobile Apps is more complex than testing desktop applications because we need to deal with issues related to mobility, as well as the same issues found in web applications.

According to Muccini et al. [81], [8] testing Mobile Apps differs from testing traditional applications because Mobile connectivity needs to be tested for various connectivity scenarios, networks, resource usage, and performance degradation, possibly resulting in the incorrect system functioning. These items need to be evaluated, as does energy consumption. Varying device screen resolutions, dimensions, etc. affect usability requiring usability testing. The large combination of platforms, operating systems, diversity of devices, and rapid evolution is challenging for a tester. Assessment of the performance is important because many of these testing needs require that a functional test be executed for a number of specific environmental scenarios, set-ups, and devices.

Our goal is to determine a set of regression testing techniques for black box testing of Mobile Apps. Most work on regression testing uses only one of the three approaches to regression testing we defined above. There are only a few papers [111, 106, 107, 103] that try to propose a combination of regression testing (selective, minimization, and prioritization) as more cost-effective [111].

Whether we need to apply multiple regression testing approaches depends on what types of changes were made to the functionality. For example, during corrective maintenance, we fix defects. There is no change in functionality and there are no obsolete test cases so there is no need for selective regression testing because all test cases are still valid. In this case, we use test case minimization and prioritization.

During enhancement when adding new functions and services, we could have obsolete test cases, but we also need new tests, in this case, we need selective regression testing. Test case minimization may not work well if there are no retestable test cases, but we can still prioritize tests. Figure 1.2 shows when to use which combination of regression testing techniques. Since we are interested in Black-Box model-based testing, changes in functionality equate to changes in the model used for MBT.

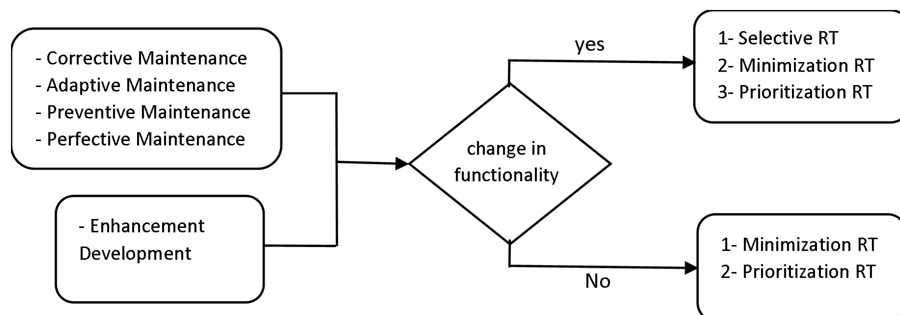


Figure (1.2) Condition to Determine Combination of Regression Testing

Selective regression testing [15] is desirable when model changes are isolated, if there are obsolete tests, and/or if additions have few overlaps with existing parts of the model. Minimization is desirable to remove redundant tests to reduce effort. Prioritization regression testing is desirable to maximize early fault detection [67].

This research contributes a regression testing approach for FSMApp which currently does not have one. We will provide a combination approach for regression testing of Mobile Apps that combines all three types of regression testing (selective, minimization, and prioritization) of Mobile Apps. Our research provides contributions to cover the large gap for regression testing approaches of Mobile Apps (only 4 research papers exist), and the large gap for the combination of regression testing approaches (only 2 types of combinations are covered by existing research). Also, there are no guidelines on how and when to select a particular combination of regression testing approaches based on a systematic approach.

1.2 Research Scope

In this research, our scope is to focus on regression testing for Mobile Apps during software maintenance and evolution. We will investigate how to best combine selective regression testing, test minimization, and test prioritization. In order to identify our scope, we developed a number of research questions. The following research questions define the scope of our work:

- RQ1: What research exists for black-box model-based testing (MBT) for Mobile Apps?
  - RQ1.1: What are the existing MBT techniques for Mobile Apps?
  - RQ1.2: What research exists for regression testing of Mobile Apps?

- RQ2: What research exists for black-box Regression Testing(selective, minimization, and prioritization)?
- RQ3: What research exists for combining the three approaches for black-box regression testing and how do these existing approaches and our approach compare?
- RQ4: Can we extend an existing regression testing for FSMWeb [16] to selective regression testing for Mobile Apps?
- RQ5: How do we best minimize a test suite for Mobile Apps for black-box regression testing?
- RQ6: What is the best way to prioritize test cases for Mobile Apps for regression testing?
- RQ7: Can we combine these approaches for a more efficient and effective regression test suite?
- RQ8: Can we identify guidelines on how to combine them?
- RQ9: How can we validate our new approach?

1.3 Research Agenda and Contribution

Our work focuses on providing a robust Model-Based regression testing approach for Mobile Applications that gives guidance for which combination of regression testing should be used during the maintenance and evolution of Mobile Apps. This work includes answers to our research questions as follows:

- For RQ1, RQ1.1, and RQ1.2: we reviewed the existing body of work related to FSMWeb, MBT techniques for Mobile Apps, and regression testing of Mobile Apps. (Section 2.1)
- For RQ2: we reviewed the existing body of work related to regression testing with a focus on black-box regression testing (selective, minimization, and prioritization). (Section 2.2)
- For RQ3: we reviewed the existing body of work related to combining regression testing approaches and compare the proposed approach to these existing approaches. (subsection 2.2.5)
- RQ4: We provide a model-based black-box selective regression testing technique for Mobile Apps. (Chapter 3)
- RQ5: We provide a test case minimization approach based on concept analysis for Mobile Apps. (Chapter 3)
- RQ6: We provide a test case prioritization approach for Mobile Apps. (Chapter3)
- RQ7 and RQ8: We provide guidelines on how to combine the types of regression testing approaches. (Chapter4)
- RQ9: Validation of the new regression testing approaches of FSMApp (selective, minimization, and prioritization) for Mobile Apps and the guidelines for combining regression testing approaches are performed. Also, we showed our novel contribution to the portability web to Mobile App case study. (Chapter5)

This document is organized as follows: Chapter 2 covers existing work for model-based testing techniques for web applications and Mobile Apps, regression testing for Mobile Apps, regression testing (selective, minimization, and prioritization), and black-box testing with FSMApp. Chapter 3 explains our multi-faceted approach to regression testing. Guidelines for combining regression testing approaches are presented in Chapter 4. Validation of the new regression testing approaches for Mobile Apps and the guidelines for combining regression testing approaches are presented in Chapter 5. Future work is presented in Chapter 6 and finally, the conclusion is in Chapter 7.

Table 1.1 shows the list of submitted and published papers and their location in this dissertation.

Publication	**Venue**	**Status**	**Chapter**
FSMApp: Testing Mobile Apps. [8]	Advances in Computers, Elsevier	Published	Ch2. Section 2.1.2
Regressing Testing of Mobile Apps.[2]	CSCI'21	Published	Ch. 3
Test Minimization for Mobile App Testing with FSMApp. [3]	SERP'22	Published	Ch. 3
Test Case Prioritization for Mobile Apps [5]	CSCI'22	Published	Ch. 3
Guidelines for Combining Regression Testing Approaches [4]	CSCI'22	Published	Ch. 4

Table (1.1) Publication

Chapter 2

Background

2.1 Black-Box Model-Based Testing

Model-based testing techniques use a variety of models such as state charts or sequence diagrams. Utting et al. [112] define six dimensions of Model-Based Testing (MBT) approaches (a taxonomy) in their survey: model scope, characteristics, paradigm, test selection criteria, test generation technology, and test execution. They defined MBT as an automatable derivation of concrete test cases from abstract formal models and their execution. They classify MBT by notation used, such as State-Based, History- Based, Functional, Operational, Stochastic, and Transition-Based. This dissertation proposes a model-based black-box regression testing technique that uses a state-based model (FSMApp [8]). Nguyen et al. [83] define Model-Based Testing (MBT) as a method for generating test cases using an abstraction of the system under test (SUT). The model provides an abstract view of the SUT by focusing on specific system characteristics. Dias-Neto et al. [39] present 219 model-based testing (MBT) approaches after analyzing 271 MBT papers and describing methods that support the selection of MBT techniques for

software projects, including risk factors that may influence the use of these techniques in the industry. This dissertation addresses the use of MBT for functional regression testing of Mobile Applications.

Web Applications have many things in common with Mobile Apps, such as screens, navigation, touch-based use, etc. This makes it important to see what approaches exist for Black-Box Model-Based Testing of Web Applications and whether they could possibly be adapted to test Mobile Apps. Many Model-Based Testing techniques exist for web applications such as [74, 78, 43, 92, 87, 66, 46, 84, 16, 17]. One of these techniques, FSMWeb [16] is the most promising, since it has been cited extensively and provides not just for functional testing [16, 90], but has been evaluated for scalability [17], has been extended to allow for selective regression testing [14, 15], and already has been adapted for testing Mobile Apps [8].

Andrews et al. [16] used FSM to model web applications, they proposed FSMWeb, a black-box model-based testing approach for web applications. It is a hierarchical approach that models both data and behavior. The FSMWeb approach [16] proceeds in two phases. First phase: building a model of the web application, and achieved by these steps: (1) an application is divided into clusters, (2) logical web pages are defined, (3) FSMs are built for each cluster, and (4) for the web application top level (AFSM). Second phase: generating a test suite from the model. It is achieved in three steps: (1) generate test paths for each cluster (2) use path aggregation to generate abstract test paths, and (3) inputs are selected for the abstract test paths. Andrews et al. [17] study the scalability issues of the traditional FSM model of web applications compared to FSMWeb.

The existing MBT approaches use different types of models, both formal and informal, to model software behavior. Informal modeling includes UML State Charts, Sequence and Activity diagrams, etc. Formal modeling includes Petri Nets (PNs),

Finite State Machines (FSMs), Extended Finite State Machines (EFSMs), Specification Description Language (SDL), and Communicating Extended Finite State Machines (CEFSMs). A model is considered formal if its notation includes not only syntax but formal semantics. Very few of these types of models are used to test Mobile Apps. We will discuss these in the next section.

2.1.1 MBT Techniques for Mobile Apps

We focus on model-based testing of Mobile Apps. The Model-based testing approach builds a model of the application under test and uses this model to generate tests. In model-based testing, the model is created by the user manually or is generated automatically by a tool. Tests are generated either manually or by tools and then executed. Our interest is in the Black-Box functional testing of Mobile Apps.

Sahinoglu et al. [98] introduce a mapping study of testing Mobile Applications. They study the research issues in testing Mobile Apps and the most frequent test type and test level of existing studies. In their study, they found only six model-based testing studies out of 123 studied. They present the count of every classification of test levels, test types, and research issues of each study, but they did not discuss or reference the papers themselves. These few studies show a lack of MBT techniques to test Mobile Apps. We focus on the use of a state-based behavioral model.

Mendez-Porras et al. [77] discuss empirical studies of automated testing of Mobile Apps in their systematic mapping study. The categories include Model-based testing, Capture/replay, Model-learning, Systematic testing, Fuzz testing, Random testing, and Script based testing. This dissertation focuses on Black- Box Model-

Based regression testing of Mobile Applications. Therefore, only the category of Model-based testing is related to the scope of this dissertation. We are not interested in white-box testing, grey-box testing, testing security, data-flow analysis, and life cycle testing of Mobile Apps, so these papers are excluded.

Zein et al. [117] present a mapping study of Mobile Application testing techniques. They consider studies of Mobile testing techniques, services, security and usability testing of Mobile Apps, and the challenges of testing Mobile Apps. The used categories are: Usability testing, Test automation, Context-awareness, Security, and a general category. The general category consists of all studies which are not in the other areas. These defined categories are not directly speaking to the scope of this dissertation. The category Test Automation includes two potentially related subcategories: Model-based test automation and Black-box test automation. The other categories are not related to our scope.

After analyzing the papers listed under Model-based testing and Black-box testing in the above studies, we found the following papers fall into our scope [37, 59, 70, 11, 36, 108, 19] in additionally to the FSMApp approach [8] which we interested in adapting it for regression testing of Mobile Apps. We summarise the papers that fall into our scope.

Alhaddad et al. [8] extend FSMWeb to FSMApp for testing Mobile Application. FSMApp proceeds in four phases: Phase 1 build a hierarchical model of HFSM. Phase 2 generates tests from the HFSM. Phase 3 selects the inputs Phase 4 compiles and executes tests through automated Mobile testing tools. Since we are interested in adapting the FSMApp approach for regression testing, We will explain FSMApp in detail in the next subsection.

De Cleva Farto et al. [37] evaluate the use of MBT to verify and validate Mobile Apps through automated tests. They use an Event Sequence Graph (ESG) to build

a test model of the App under test. An ESG expresses the requirements and the functionality of the system under test. ESG is a directed graph that represents the events (nodes) and possible sequences of events (edges). Their approach has three stages: create the ESG model, generate and implement test cases from the ESG model, and execute the test cases with Robotium and collect data. This approach is the closest to FSMApp. It does not address regression testing. Alhaddad et al. compared the FSMApp approach to the ESG approach by applying both approaches to a number of Mobile Apps with FSMApp being more efficient.

Jing et al. [59] present a Model-based Conformance Testing Framework (MCTF) for Android Applications. The model is generated manually from the requirements. Their framework consists of four steps: System Modeling, Test Case Generation, Test Case Translation, and Test Case Execution. The parameters and properties of an App are derived by System Modeling, while abstract tests are generated automatically by Test Case Generation with Alloy Analyzer (a language for describing structures and exploring them). Test Case Translation converts abstract tests into executable tests. Test Case Execution compiles executable test cases to generate test packages. The Android apps or the operating system is then tested with the packages.

Costa et al. [36] adopt a Pattern-Based GUI Testing (PBGT) technique for testing Mobile Apps. Their technique is based on User Interface Test Patterns [82] that are specifically developed for testing web apps. They aim to increase reusability and reduce the effort of modeling and testing Mobile Apps. The PBGT approach for Mobile Apps differs from the PBGT approach for web apps in the mapping and interaction strategy and the fact that Mobile Apps can call other applications. However, this approach does not include some gestures and components like swiping and zooming which the FSMApp approach supports. Also, it does not support

components, varying screen sizes, and loops. It needs a separate model for every function of the Mobile App. They cannot be connected to create a hierarchical model to test the Mobile App.

Takala et al. [108] used a model-based testing tool (TEMA) to represent a test automation solution for testing Android apps. The TEMA tool is a set of model-based tools for distinct phases of MBT: design, generation, and debugging tests. They used TEMA tools with state machines for testing Mobile Applications. In their approach, the model can be divided into small components that are connected. Each component has two levels: an action machine and a refinement machine. However, the model is complex even for small apps because each component is a combination of an action machine and a refinement machine of each function of a Mobile Application. Further, the two levels make testing of large Mobile Apps difficult because a refinement machine state model can only connect to one action machine. In the FSMApp approach, hierarchical levels and clusters are used to mitigate the state explosion problem and simplify the generation of a model for large Mobile Apps.

Baek et al. [19] propose a model-based black-box approach for testing Android apps with a set of multi-level GUI Comparison Criteria (GUICC). It creates a directed graph using ScreenNodes and EventEdges. The GUICC has five levels. They develop a testing framework around GUICC. It consists of three modules: the communication layer connects the desktop with the Mobile device, the EventAgent is a tool to run the test on a Mobile device, and the testing Engine generates a GUI graph with GUICC, test inputs, and test cases. However, their framework is not open to the public and it is not explained in enough detail. This makes it unusable for our purpose.

The next two approaches [70], [11] are based on a crawler, they use an algorithm that traverses the app and searches for possible transitions. This is used to build

the model used for testing. Depending on the technique (and automated inputs) used, the model may be incomplete, hence does not fit our needs.

Lu et al. [70] represent an activity page-based model to automate functional testing for Mobile Apps. The Activity page-based model is a directed graph. An activity page is like a screen and is modeled as a tuple to describe input constraints and related actions. Edges represent a trigger event between the states. The model can be generated either by using a tool (UIAutomation tool) or by creating a model based on an image comparison for every event. They present two modeling methods based on an activity page based model. Android apps contain activity components that help to develop the user interface of an app. An app usually contains one or more activity classes to provide GUI interfaces for the user. This is used to build an activity page based model to automate functional testing for Mobile Apps. However compared to the FSMApp approach, the description of the input constraints for this approach is incomplete, especially related to dependencies between inputs and it is unclear at which step the input constraints are resolved into values.

Amalfitano et al. [11] propose a GUI automated approach to test Android apps. They implement their technique in a tool called Android Ripper. The technique is based on a GUI Ripper: the software's GUI is automatically traversed by opening all its windows and extracting all its widgets, properties, and values. The ripping generates a tree model. Compared to the FSMApp approach, the tool takes a long time to generate test cases for large Mobile Apps, and it does not support some inputs, such as sensors, but the FSMApp tests large apps and includes more input types.

Alhaddad et al. [8] validate FSMApp for ten Mobile Applications. Their case studies cover Mobile Apps from different domains and with different sizes. They investigate the applicability, scalability, efficiency, and effectiveness of FSMApp for

testing Mobile Applications. They also compare the FSMApp approach to the ESG approach [37] with respect to model building, test generation, input selection, making tests executable, and test execution.

FSMApp can be applied to a variety of Mobile Apps in different application domains and of different sizes, showing the applicability of the FSMApp approach. They applied FSMApp to ten Mobile Applications from different categories. The apps fall into five categories: Health and Fitness, Game, Music and Audio, Tools, and Productivity. They were successfully able to apply FSMApp to all ten Mobile Applications.

They also consider the scalability with regards to four phases: Generate the Model, Generate the test sequences, Input selection, and Test execution and validation.

They also studied the efficiency of FSMApp. The efficiency is evaluated in the test generation process, the length of test sequences, and the test execution effort (time). Their study shows the efficiency of FSMApp compared to ESG.

Their case study executes the test cases and captures the number of defects. They were not able to show the full effectiveness of FSMApp because all Mobile Apps were professional, and related product contains very few faults.

Their comparison between the FSMApp and the ESG approach shows that FSMApp and ESG have the same applicability. They compare the scalability of FSMApp and ESG with the model size in terms of edges, model generation time, number of test sequences, test sequences generation time, the total number of inputs and actions, time to choose an input, test lines of code, and execution time. FSMApp is more scalable compared to ESG. They compare the efficiency of FSMApp with ESG. The efficiency is evaluated for all phases of test generation and execution. The FSMApp model is almost half the size of the ESG model. Building the model for

FSMApp takes much less time than building the ESG model. The reason for this is the model for FSMApp is much smaller than the model for ESG. The ESG model has twice the number of nodes and more than double the edges compared to FSMApp. There is a big difference in the number of test steps between FSMApp and ESG. The FSMApp approach takes less than half the time of the ESG approach.
The case study also executes the test cases and captures the number of defects. FSMApp and ESG each found one defect. However, the high quality of the apps used in the case study makes conclusions related to effectiveness limited.

For these reasons, we conclude that FSMApp is more promising for adapting it to regression testing of Mobile Applications. Next, we present the testing approach for Mobile Apps, FSMApp [8] which we will extend for regression testing of Mobile Apps.

2.1.2 Black-Box Testing with FSMApp

This subsection is a summary of the FSMApp approach for testing Mobile Applications [8] using Finite State Machines. FSMApp is an extension of FSMWeb [16]. FSMApp proceeds in four phases:
Phase1: Build a hierarchical model HFSM for Mobile App:

1. Partition Mobile App into clusters.

   Partitioning Mobile Applications into clusters, each of which is composed of App pages and other clusters. The term cluster is referring to collections of software modules/app pages that apply a logical or user level function. Clusters represent functions that can be identified by users, at the highest level of abstraction. The hierarchical model (HFSM) is a collection of models for clusters (FSM_i) and a top level model (AFSM). $HFSM = \{FSM\}_{i=0}^{n}$

with a top level $FSM_0 = AFSM$. Each FSM has nodes that represent either clusters or Logical App Pages (LAPs). Edges are internal or external to each FSM. External nodes span cluster boundaries. (They become internal at the next higher level.) External edges can either enter or leave a cluster FSM [8]. The clusters might be an individual Activity which is a screen that is shown to the application user, and it contains a set of layouts that organizes the item on the page, or software modules that represent a major function. Clusters can be identified from the spot navigation layout, coupling relationships among the components and design information [16].

2. Define logical app page and input constraints.

   The Mobile Apps' screens will be considered as input components, or logical app pages (LAPs). There are several app actives (screens). They are modeled as multiple Logical App Pages (LAPs). A Logical App Page is either a physical app page, physical app component, or the portion of an app activity that receives data from the user via an XML form, and then sends the data to a particular software module [8]. All inputs in a LAP are considered atomic: data entered in a text field is only one user input, regardless of how many characters are entered into the field. The inputs might have rules to enter: some inputs may be required, others may be optional, users might be permitted to enter inputs in any order, or a specific order may be required. The constraints on inputs and the meaning of each are as follows:

   - Required (R): required input must be entered.
   - Required Value (R(parm)): one must enter at least one value.
   - Optional (O): an input may or may not be entered.

- Single Choice (C1): one input should be selected from a set of choices.
- Multiple choice (Cm): more than one input should be selected from a set of choices.

Mobile Apps can have a variety of components that can be modeled via input constraints. Mobile Apps have many more input types than web applications. The difference between the web input types and Mobile input types is swipe (which means it is required to change the value of a component) and scroll (which indicates that the input required is to scroll up or down the content). Table A in (appendix A) shows how typical input types found in Mobile Applications are represented as input constraints on edges in an FSMApp model. The variety of components of Mobile Applications can be modeled through input constraints. Although Mobile Applications differ a little between different Mobile operating systems, they also have many types of components in common [33], [8]. We explain common components of Android applications and what FSMApp's input constraints would look like [1]:

- Bottom Sheet: this is a sheet of material that slides up from the bottom edge of the screen. The action of the bottom sheet is a click. The input constraint is R(<Click>).
- Button: indicates what action will occur when the user touches. The action of the button is click. The input constraints are R(<Click>) or select the button then click C1(Select Button, Click).
- Card: is a sheet of material with unique related data that serves as an entry point to more detailed information. The actions of the card are click, swipe, scroll, and pick-up-and-move. The input constraint is C1(<Click>, <Swipe>, <Scroll>, <Pick>).

- Chips: represent complex entities in small blocks, such as a contact. The action of the chip is a click. The input constraint is R(S(<Select Button>, <Click>)).
- Data tables: are used to represent raw data sets, and usually appear in desktop enterprise products. The actions of the data tables are row hover, row selection, column sorting, column hover, and test editing. The input constraint is C1(<Row hover>, <Selection>, <Sort>,<Column hover>, <Edit>).
- Dialogs: inform users about critical information, require users to make decisions, or encapsulate multiple tasks within a discrete process. The action of the dialog is click. The input constraint is R(<Click>).
- Dividers: group and separate content within lists and page layouts. The action of the divider is click. The input constraint is R(<Click>).
- Grid lists: are an alternative to standard list views. The actions of the grid list are vertical scrolling or filtering. The input constraint is C1(<Scroll>, <Filter>).
- Lists: present multiple line items in a vertical arrangement as a single continuous element. It has a checkbox, a switch, and a reader. The action of the list is sort. The input constraint is R(<Sort>).
- Menus: allow users to take an action by selecting from a list of choices revealed upon opening a temporary, new sheet of material. The actions of the menu are scroll and click. The input constraint is C1(<Scroll>, <Click>).
- Pickers: provide a simple way to select a single value from a pre-determined set. For example, time and date pickers. The actions of the pick-

ers are dropdown and click. The input constraint is C1(<Dropdown>, <Click>).

- Sliders: let the user select a value from a continuous or discrete range of values by moving the slider thumb. The action of slider change is scrolling. The input constraint is R(<Scroll>).
- Snackbars & toasts: provide lightweight feedback about an operation by showing a brief message at the bottom of the screen. The action of snackbars & toasts is click. The input constraint is R(<Click>).
- Progress & activity: indicators are visual indications of an app loading content. The action of Progress & Activity is loading. The input constraint is R(<Load>).
- Selection Controls: allow the user to select options. The action of selection controls is click. The input constraint is R(<Click>).
- Subheaders: these are special list tiles that delineate distinct sections of a list or grid list and are typically related to the current filtering or sorting criteria. The action of subheader is click. The input constraint is R(<Click>).
- Steppers: convey progress through numbered steps. They may also be used for navigation. The action of the steppers is to show the next steps. The input constraint is R(<Follow>).
- Tabs: make an app easy to explore and switch between different views or functional aspects of an app or to browse categorized data sets. The action of the tab is scroll. The input constraint is R(<Scroll>).
- Toolbars: appear above the view affected by their actions. The action of toolbars is scroll. The input constraint is R(<Scroll>).

- Tooltips: are labels that appear on hover and focus when the user hovers over an element with the cursor, focuses on an element using a keyboard (usually through the tab key), or upon touch (without releasing) in a touch UI. The actions of tooltips are click and hover. The input constraint is C1(<Click>, <Hover>).
- Text fields: allow the user to input text, select text (cut, copy, paste), and lookup data via auto-completion. The actions of test fields are look up the table, select the text, and write the text. The input constraint is C1(<Lookup>, <Select>, <Write>).

Transitions connect the nodes and the clusters. They are annotated with input constraints to indicate what inputs and actions lead to the next node or cluster.

3. Build FSM for the clusters.

   After an App application has been partitioned into clusters, a Finite State Machine (FSM) for each cluster is derived. Ultimately, an Application Finite State Machine (AFSM) will define the top level of the Mobile App. Each cluster (except for the lowest level cluster) may contain both LAPs and cluster nodes. We refer to this collection of FSMs as HFSM. Dummy nodes and transitions are added to ensure each FSM_i is single-entry, single-exit.

Phase 2: Generates tests from the HFSM,

1. Generates Paths through FSMs/AFSM.

   In this phase of the FSMApp method, test sequences are generated. The user can select coverage criteria such as node, edge, edge-pair, simple round trip, and prime path coverage [12]. Test paths for clusters are generated by

applying standard graph criteria such as node coverage or edge coverage [12]. The result is a set of test paths for each FSM_i. $T_i = \{T_{i_1}, ..., T_{i_n}\}$, i=0, ..., n.

2. Aggregate paths to form abstract tests.

   The test paths are aggregated into test paths for the model as a whole. Several aggregation criteria have been proposed: all-combinations, each choice, and base choice coverage [12]. For example, if the path aggregation criterion is to use each path at least once, this requires that we need to use each test $t \in T_i$ at least once when replacing C_i with $t \in T_i$. This means starting with the top level model (AFSM), we replace a cluster node C_i in a path with one of the test paths $t \in T_i$ through it. This ultimately results in a test path from the start node in AFSM to the end node in AFSM that only contains LAP nodes. Aggregation criteria address which cluster paths need to be used. The result of this process is a set of aggregate paths, the abstract tests. Algorithm 1 shows the procedure to aggregate test paths with their criterion. The inputs to the algorithm are AFSM and cluster test paths. The output of algorithm 1 is a set of aggregated test paths (abstract tests). In line 1, the algorithm put AFSM test paths into an input List. Then line 2 iterates through every test path from the input List. Line 3 takes one test path from the input List. Then, it sequentially checks each node in the path whether it is a cluster node. If there is a cluster node in the path, then a loop replaces the cluster node with each cluster path and creates as many new partially aggregated paths as there are paths through this cluster node.

```
Input: AFSM and Cluster Test Paths
Result: outputList = Set of Aggregated Paths
 1: inputList = AFSM Paths
 2: while inputList has next path do
 3:    currentPath = get one path from inputList
 4:    pathDone = true
 5:    for i = 1 to Length(currentPath) do
 6:       if nodei is cluster node then
 7:          for j = 1 to Length(cluster paths) do
 8:             Replace node_i with cluster path_j and add new path into inputList
 9:          end for
10:          add list paths to inputList
11:          remove currentPath from inputList
12:          i = length (currentPath) + 1
13:          pathDone = false
14:       end if
15:    end for
16:    if pathDone is true then
17:       Move currentPath into outputList
18:    end if
19: end while
```

Algorithm 1: Aggregated Test Paths for "Each path at least once" Criterion

3. Reduction Step

   Dummy nodes and transitions were added in step 1 to ensure the FSM_i are single-entry, single-exit. The dummy nodes and transitions do not require any inputs as they are not really testing steps. However, they increase the length of the test paths. So, these dummy nodes, and transitions are removed. We replaced each remaining node pair (edge) in an aggregate path with its corresponding input action constraint.

   Algorithm 2 shows the procedure for the reduction step. The input of algorithm 2 is the set of Aggregated Paths and the number of paths. A sequence of input constraints for each aggregated test path is the output of algorithm 2. There are two loops in algorithm 2: the first loop processes all test paths.

The second loop visits each node pair (edge) of the test path and adds the constraint on the edge to the sequence if there is one.

```
Input: Set of Aggregated Paths, Number of Paths n
Result: Sequence of Input constraints for Each Aggregated Test Path
1: for i = 1 to n do
2:     for j = 1 to Length(path_i)-1 do
3:         if edge (node_j, node_{j+1}) has constraint then
4:             add to constraint sequence forpath_i
5:         end if
6:     end for
7: end for
```

Algorithm 2: Test Reduction Step and Input Constraint Sequence generation

Phase 3: Selects the inputs: choose inputs to create abstract tests.

Inputs are selected to replace the input constraints in the aggregated test path. The test designer selects the test values. For example, the test designer can create input values by covering partitions or randomly selecting values from a list of input selection constraints that are met. The test designer selects values for related inputs.

Phase 4: Compiles and executes tests through automated Mobile testing tools.

In this phase, the test cases can be executed manually or we can use an automated tool to execute them. For Mobile Apps, many automated tools are available to run the test cases. For example, tests can be converted to Selenium [91].

2.2 Regression Testing

Regression testing (selective, minimization, and prioritization) plays an important role in software maintenance. Regression testing is the process of validating that the modifications introduced in a software system work and do not adversely affect the unchanged part of the software system. The goal is to provide confidence that changes were implemented correctly. There are three types of regression testing

techniques: selective regression testing, test case minimization (reduction), and test case prioritization. In selective regression testing, a regression test suite is determined by selecting tests from the original test suite and adding new tests to test new paths of the software. Test case minimization techniques attempt to reduce a test suite, usually based on some objective function related to coverage criteria. Test cases prioritization techniques rank and then prioritized the order in which tests are executed to achieve required goals [28].

We are interested in Model-Based regression testing of Mobile Apps. That means we will use the system model(s) for regression testing. The model is analyzed to collect information related to modifications then this information is utilized for determining a regression test suite. We will survey the three types of regression testing approaches and review model-based approaches for each type to determine potentially useful papers that fall in our scope.

2.2.1 Selective Regression Testing

Selective regression testing techniques reduce the cost of testing modified software by reusing existing tests and identifying the portions of the modified software that need new tests [94]. The test case selection problem is defined formally by Rothermel and Harrold [94] as follows:

Given a program, P, a test suite, T (original test suite) to test P, and the modified version of P, P′.

1) Identify changes produced to P by creating a mapping of the changes between P and P′.

2) Select $T' \subseteq T$, a set of tests to execute on P′. T′ may reveal change-related faults in P′.

3) Test P' with T'.

4) Identify if any portions of P' have not been tested adequately and require additional testing, and create T'', a set of new tests for P'.

5) Use T'' to test P'.

Existing techniques for Selective Regression Testing can be classified as follows: Model-Based, Component-Based, Specification-Based, Requirement-Based, Search-based, and Similarity-Based. Some of the selective regression testing techniques are classified into multiple categories. We focus on black-box model-based regression testing. There are many MBT selective regression testing techniques based on UML, FSM, EFSM, etc such as ([14], [15] [110], [88], [45], [57], [52], [13], [53], [56], [51], [73], [25], [68], [30], [86], [24], [23], [44]).

Chen et al. [30] describe regression tests as: A Targeted Test for testing functionality of the changed portions of the software, and a Safety Test for addressing risks. They generate test cases utilizing UML activity diagrams. They categorize two types of changes; code changes and system behavior changes. Orso et al. [86]proposed an approach similar to Chen et al. [30], but instead of UML activity diagrams, they used a Statechart diagram. Their approach compares the new and the old versions of the state charts and finds test cases that are affected by the changes. Briand et al. [24] [23]proposed a regression testing technique using use case, sequence, and class diagrams. They describe the meaning of the types of changes in diagrams (e.g. added/deleted attribute, added/deleted method) and use them to classify tests into reusable, retestable, and obsolete. Iqbal et al. [44] present a regression test selection approach utilizing UML state machines and class diagrams. We are not interested to use UML model in our work.

Andrews et al. [14], [15] proposed an approach for selective regression testing of web applications using FSMWeb and develop a cost-benefit tradeoff framework between brute force and selective regression testing. In regression testing of FSMWeb [14], types of changes to the FSMWeb models are formalized based on changes to the web application (its inputs, Logical Web Pages, and navigation between them). Specifically, the selective regression testing approach identifies obsolete, reusable, and retestable tests. Retestable tests need to be rerun (T' in [94]). Finally, it identifies new tests that are necessary to achieve required coverage (T'' in [94]). In [15] Andrews et al. proposed a tradeoff analysis framework to determine the best regression testing technique by considering different cost factors that are related to software artifact analysis and technique application. Their framework is based on classifying tests as obsolete, reusable, and retestable. They apply the proposed tradeoff framework by utilizing the FSMWeb model as the behavioral model, and analyze the cost-benefit tradeoffs for the two regression testing techniques (brute force and selective techniques), which vary with the amount of change and the impact of model changes.

Since selective regression testing may produce redundant tests [26], test case minimization may be helpful to eliminate them.

2.2.2 Test Case Minimization

Test case minimization techniques aim to identify redundant test cases and to remove them from the test suite to reduce the size of the test suite while still meeting all requirements [109]. The test minimization problem is equivalent to the minimum set-cover problem [69].

Given: A test suite, T, a set of test requirements R $\{r_1, ..., r_n\}$, that must be satisfied to provide the desired testing of the system, and subsets of T, $\{T_1, ..., T_n\}$, such that $T_i = \{t|t \in T \wedge t \text{ tests } r_i\}$. In other words T_i contains tests that meet test requirement r_i.

Problem: Find a minimal cardinality subset of T that meets all test requirements in R.

A classical approach for this problem [32, 35] uses a straightforward greedy algorithm: Select the test cases that cover the most test requirements until all test requirements are covered, removing redundant test cases along the way. In the following, let $T = \{t_1, \ldots, t_m\}$ be set test suite to be minimized. $R = \{r_1, \ldots, r_n\}$ is the set of requirements covered by T. $T_i = \{t|t \in T, t \text{ covers } r_i\}$. The algorithm selects test cases with the largest number of test requirements until all required elements are covered. First, it will select the test case that covers most requirements. Second, it will remove these requirements that are covered by the selected test case. Then, it will select new test cases that cover most of the remaining requirements. This continues until all requirements are covered. Algorithm 3 shows the pseudo code for this algorithm. This approach will not always produce a minimal set.

Input: $T = \{T_1, \ldots, T_n\}$, $R = \{r_1, \ldots, r_n\}$
Result: : $minT$: minimal test set
algorithm minT (T,R)

1: $minT=$ empty
2: $T = \{t_1, \ldots, t_n\}$
3: $R = \{R_1, \ldots, R_n\}$
4: **for** i=1 to n **do**
5: $\quad R_i = \{r | t_i \text{ meets } r \in R\}$
6: $\quad TR = \{r_1, \ldots, r_m\} = \cup R_i$
7: **end for**
8: Find index k for which $max(|R_i|)$
9: $T = T \setminus t_k$
10: $minT = minT \cup t_k$
11: $R = R \setminus R_i$
12: $TR = TR \setminus R_i$
13: **if** TR = empty **then**
14: $\quad$ return minT
15: **else**
16: $\quad$ **for** j=1 to n **do**
17: $\quad\quad R_j = R_j \setminus R_i$
18: $\quad$ **end for**
19: $\quad minT = minT \cup mingreedy(T, R)$
20: $\quad$ return minT
21: **end if**

Algorithm 3: minimal set

Harrold et al. [50] developed a variant of this greedy algorithm. It starts by selecting $t \in T$ that are included in the most T_i of a given size, and adds those r_i to the requirements covered until the largest set of $T_i s$ has been processed. If there are any sets T_i from which no tests have been chosen, they are considered next. Algorithm 4 (Reduce Test Suite) shows it.

Input: $T = \{T_1, \ldots, T_n\}$, $R = \{r_1, \ldots, r_n\}$
Result: : T_{min}: a representative set of $T_1, T_2, \ldots, T_n$
algorithm ReduceTestSuite (T,R)

1: T_{min}= empty, CurCard= Min(Card(T_i))
2: $T = \{t_1, \ldots, t_n\}$
3: $R = \{r_1, \ldots, r_n\}$
4: $T_i = \{t|t \text{ meets } r_i\}$ i=1,..., m
5: **for** $|T_i|$=1 to m **do**
6: $T_{min} = T_{min} \cup T_i$
7: Mark T_i
8: $R = R \setminus r_i$
9: $\forall j, j \neq i$ if $t \in T_i$ is in T_j then
{Mark T_j
$R = R \setminus R_j$}
10: **end for**
11: $C_i = 2$ is cardinality of T_i
12: **while** $R \neq empty$ **do**
13: select $T_{i_1}.....T_{i_k}$ with cardinality C_i
14: determine t_k such that t_k occurs most often in $T_{i_1}....T_{i_k}$
15: if there is a tie, consider $|T_i| = C_i + 1$
16: if no decision, select t_k randomly
17: $T_{min} = T_{min} \cup t_k$
18: $\forall j$: if $t_k \in T_j$ and T_j unmarked
Mark T_i , $R = R \setminus r_j$
$C_i = C_i + 1$
19: **end while**
20: return T_{min}

Algorithm 4: Reduce Test Suite

This algorithm does not guarantee a minimal selection (e.g. when a test is selected too early). Other approaches using a greedy algorithm to reduce the test suite include Agrawal [7, 6] and Marre and Bertolino [76].

Tallam et al. [109] suggest using concept analysis. Concept analysis classifies objects (tests) based on the overlap among their attributes (test requirements), to obtain a heuristic for test suite minimization. They attempt to prevent selecting a test too early which can cause non-minimized tests in [50, 109]. They present a greedy heuristic algorithm for selecting a minimal subset of a test suite T that covers all requirements covered by T. Similarly, Sampath et al. [99] present an approach for user-session based testing of web applications using concept analysis. Specifically, they are reducing user-session tests. These two approaches are promising to reduce test cases for regression testing of FSMApp, but cannot be directly applied, since FSMApp has two types of requirements: graph coverage requirements for individual cluster FSMs and path aggregation coverage requirements to form abstract test cases. While we could simply use both types of requirements by combining them into a large set of requirements $R = R_{graph} \cup R_{aggregation}$, this ignores the relationship between both types of requirements. We will discuss a more efficient approach in Chapter 3.

2.2.3 Test Case Prioritization

Test case prioritization techniques schedule test cases for execution to increase the rate of fault detection and provide faster feedback on the system under test (SUT), so faults can be corrected earlier. Rothermel et al. [97] formally define the test case prioritization problem as follows:

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 $(\forall T'')$ $(T'' \in PT)$ $(T'' \neq T')$ $[f(T') \geq f(T'')]$. PT represents the set of all possible orderings of T, and f is a function that, when applied utilized to any such ordering, produces an award value for that ordering.

Test case prioritization is an effective and practical technique that helps to increase the rate of regression fault detection as software evolves. Test case prioritization (TCP) techniques aim to improve regression testing cost effectiveness, by helping to reveal faults earlier in testing, which allows to start defect fixes earlier, and provides earlier feedback to testers. Test cases are ranked or prioritized to achieve required goals. The simplest test prioritization method is random test prioritization where test cases are ordered randomly. For a test suite of size N, there are N! possible test sequences. Random prioritization selects randomly one of these sequences.

The existing prioritization techniques can be classified as follows: Model-Based, Coverage-Based, Requirements-Based, Search-Based, Fault-based, History-based, and Learning-based. We focus on test case prioritization for black-box model-based regression testing. The exist model-based test case prioritization are: ([62], [61, 60], [55], [20], [21], [75], [119, 120], [100, 101], [103] , [18], [63], [29], [64], [47], [49], [80], [118], [54] , [105]).

Table (2.1) Classification of Prioritization Regression Testing Techniques.

Begin of Table				
Study	Approach	Test type	Model type	Evaluation Metric
Korel et al. [62], [61], [60]	model dependence-based test prioritization	Black box	EFSM	RP (Most Likely Relative Position)
Chen et al. [29]	dependence analysis	Black-box	Business model (SOA)	
Mariani et al.[75]	component integration	White box	Component-Based	
Belli et al. [20], [21]	Fuzzy Clustering	White box	Event Sequence Graphs (ESG)	
Huang et al. [55]	weight-based (scoring system)	Black box	Event flow graph (EFG)	APFD
Sapna et al. [100], [101]	weight-based (scoring system)	White-box	UML Activity Diagrams, Use case diagrams	Average Percentage of Fault Detected (APFD)

Continuation of Table 2.1				
Study	Approach	Test type	Model type	Evaluation Metric
Zhang et al. [119], [120]	JUnit test case prioritization	White box	UML	Average Percentage of Fault Detected (APFD)
Athira et al.[18]	Paths analysis	Black-box	UML activity diagram	
Kundu et al. [63]	weight-based (scoring system)	White box	UML sequence diagrams	
Filho et al. [103]	Concern-based (such as: risk, change impact, and other user-defined properties associated to model elements)	White-Box	UML	
Lachmann et al. [64]	weight-based prioritization	White box	Component-Based	Average Percentage of Faults Detected (APFD)

Continuation of Table 2.1				
Study	Approach	Test type	Model type	Evaluation Metric
Garg et al.[47]	functionality dependency graph	White box	Control flow graph (CFG)	APFD
Han et al. [49]	Heuristic-Based	Black-box	EFSM	Most Likely Average Position
Morozov et al. [80]	fault and error propagation analysis	Black-box	Dual-graph Error Propagation Model (DEPM)	
Hou et al.[54]	Mutation	Black-box	Component-Based	
Srikanth et al. [105]	Historical information to order tests	black box	UML Use Cases	APFD
End of Table				

Korel et al. [62] introduce methods of test prioritization based on state-based models after changes to the model and the system. They present an analytical framework for the evaluation of test prioritization methods. In addition, Korel et al. [61, 60], present model-based prioritization for modifications when models are

not modified, only the source code is modified. The method is based on identifying elements of the model related to source-code modifications and collecting information during the analysis of a model to use them to prioritize tests for execution. They present two model-based test prioritization methods: selective test prioritization and model dependence-based test prioritization. In selective test prioritization, they assign a high priority to tests that test modified transitions in the modified model. A low priority is assigned to tests that do not test any modified transition. In model dependence-based test prioritization they use model dependence analysis to identify different ways in which modified transitions interact with the remaining parts of the model and use this information to prioritize high priority tests. Also, they present some model-based heuristics: one of the heuristic methods is that tests that test a higher number of modified transitions should be given a higher priority than tests that test a smaller number of modified transitions. The idea of another heuristic method is that tests with a higher frequency of testing of modified transitions should be given a higher priority than tests with a lower frequency of testing of modified transitions. And heuristic that each modified transition should have the same opportunity to be tested during software retesting. This heuristic tries to balance the number of testing of modified transitions. In this heuristic, a higher priority is assigned to a test that tests a transition that has been tested the least number of times at the given point of system retesting.

Chen et al. [29] present a dependence analysis-based test case prioritization technique. They analyze the dependence relationship using control and data flow information and construct a weighted graph. They do impact analysis to identify modification-affected elements and prioritize test cases according to covering more modification-affected elements with the highest weight.

Garg et al. [47] proposed an approach for automatically prioritizing and distributing test cases on multiple machines. Their approach is based on a functional dependency graph (FDG) of a web application. They first construct a functionality dependency graph (FDG) of the web application from its UML specification. Then partition the complete test suite into test sets, each test set is associated with a unique functional module or node in the FDG. Next, they prioritize the test cases within each test set using the Control flow graph (CFG) of the corresponding functional module.

Han et al. [49] present a model-based heuristic method to prioritize test cases for regression testing. In their approach, they consider information collected during the execution of the modified model on the test suite. The idea of this heuristic is a test case that tests a larger number of modified transitions has a higher probability of showing a fault and takes a higher priority.

Hou et al. [54] apply mutation on interface contracts. They mutate the interface contracts of each black-box component. Their approach focuses on the faults that could possibly happen when the component users integrated reusable components in their applications. The approach uses mutation operators, which simulate various types of faults in component composition to generate mutants, and uses a mutation score, which defines the capability of killing mutants to measure the quality of test suites. They prioritize test cases according to the mutation score.

Morozov et al. [80] present a method for the automatic prioritization of test cases. Their method is based on two principles: first, a test case should stimulate an error in an updated block. Second, the stimulated error should propagate to the place where it can be detected. The idea of this method is that analyzes the original model and the test suite and identify which test cases should be run first after an update of a particular block. The result of the method is the Priority table.

Kundu et al. [63] present an approach, called System Testing for Object-Oriented systems with test case Prioritization (STOOP) for the automatic generation of test cases, and prioritization of these test cases. STOOP consists of four tasks. The input to STOOP is a set of sequence diagrams. The first task converts a set of sequence diagrams into a graph is called sequence graph (SG). Then test cases are generated from the sequence graph. The output of these two tasks is input to the task that prioritizes the test cases. Finally, STOOP ranked test cases from its preceding task and generates test data. The test case prioritization uses message weight and edge weight as metrics. The message weight of an edge between any two nodes in an SG is defined as the number of messages that occurs between the two messages in the sequence diagram. The edge weight of an edge between any two nodes in an SG is defined as the number of paths in the SG, which include the edge. Computing the values of prioritization metrics and these values are then associated with their corresponding test cases. Test cases are then prioritized according to the decreasing order of these values.

Srikanth et al. [105] present a process for prioritizing tests based on historical field failures. They first, analyze historical data, then identify the rarity of use cases, and tag use cases to provide meta-data. Next, they prioritize use cases which will then become the ordered tests that are executed during the final testing phase. In this step, they use historical data analysis and the tagged use cases to determine the best order for running tests. Prioritization can be performed either on the number of services involved in a use case or by the most fault-prone use cases first.

Athira et al. [18] present a model-based test prioritization using an activity diagram. Their approach identifies the difference between the original model and the modified model and uses this information to identify the most promising paths.

The test case which covers these paths is considered the most beneficial test case, and these test cases are assigned with high priority.

Huang et al. [55] design and analyze a GUI test-case prioritization approach weights. They assign weights for each input and action, resulting in a scoring system for test cases. The more important inputs have a higher weight. They proposed three methods for assigning a weight to each input: equal weight, fault-prone weight, and random weight. They used weight-based Event Flow Graph (EFG) methods to prioritize the test cases. They consider the importance of each event. That is if the event is more important and more faults could be detected, so they assign a high weight to this event.

Belli et al. [20], [21] proposed a model-based approach to prioritizing test cases. Their prioritized testing approach is based on the ESG-based testing and coverage based. The ordering of the coverage event sequences (CESs) is based on their importance degree which is defined by the estimation of events that are the nodes of ESG. The groups of events are constructed by using a clustering algorithm and then these groups are ordered on the importance degree according to the rule: The greater the value of attributes the more important group. Finally, the CESs are ordered based on the events which join the importance group(s). The test case has the highest degree if it contains the events which belong to the group(s) with the greatest importance degrees. The test case has the lowest degree if it contains the events which belong to the group(s) that are within the lowest part of the group(s) with the least importance degree.

Mariani et at. [75], proposed a technique to prioritize test suites. It is based on a behavioral model that represents component interactions. They prioritize test cases according to the complexity of the interactions between the system and the target component that are caused by the test. High priority is assigned to test

cases that cover components that have more complexity of the interactions, and then the priority of the remaining test cases is computed as the number of distinct interactions between the system and the target component that are covered.

Lachmann et at. [64] proposed a fine-grained test case prioritization for integration testing for software product lines. Their approach is based on the analysis of structural and behavioral deltas. Test cases are defined for all product variants under test. They analyze the differences between state machines of an architecture model for product variants to derive a behavioral component weight. They compute behavioral component weights and compute structural component weights. Then sum all weights of components contained in a test case to be prioritized.

Zhang et al. [119, 120] proposed an approach called Jupta for prioritizing JUnit test cases in the absence of coverage information. Jtop prioritizes test cases based on finding relevant test cases for certain elements of the program under test. A test case that has more relevant elements has high priority.

Sapna P.G et al. [100, 101] proposed a prioritization technique based on UML. The constructs of an activity diagram are used for test scenario prioritization. In [100] Activity diagrams represent the scenarios of the system under test. A scenario is a complete path through the activity diagram. They prioritize scenarios by assigning weights to nodes and to edges then calculate the weight of the path(scenario). They assign weights to each of these nodes based on the complexity and possibility of the occurrence of defects. And they assign weights to each of these edges based on the number of incoming dependencies of the node and outgoing dependencies of the node. Their approach [101] is to prioritize use cases from UML use case diagrams. The developer assigns the priority of each actor on a scale of 0-10. Then Compute use case priority from the use case diagram. Next, obtain customer prioritization of use cases. Customers rate uses cases as very low, low, medium, high, or very

high (scale 1-10). Finally, calculate use case priority. The priority of a use case is computed as a weighted sum of Customer Priority and Technical Priority.

Filho et al. [103] proposed a model-based regression test prioritization approach that selects test cases for regression testing based on different concerns. They illustrate how to support these concerns using TDE/UML, an extensible model-based testing environment. Their approach includes different user-defined concerns such as the last change date, requirements, risk, and features, are represented as properties in the model. Furthermore, through traceability links between requirements, model, test cases, and code artifacts, these concerns are used to automatically select and prioritize test procedures, before they are used for code generation and execution.

Next, we present the regression testing approaches for Mobile Apps.

2.2.4 Regression Testing Techniques for Mobile Apps

Amalfitano et al. [10] use a crawler-based technique to generate tests for Android apps. The crawler builds an event-based GUI tree and generates tests from this tree. This is not the same as a behavioral model, as it does not allow for looping behavior like for example FSMApp. When using this approach for regression testing, they rerun previous tests and check whether program behavior has changed. This check is based on comparing sequences of user interfaces obtained in both test runs. They add specific assertions to the original tests for this purpose. This approach does not fall into any of the regression testing approaches discussed in Sections (2.2.1, 2.2.2, and 2.2.3).

Do et al. [41] provide an approach for limited selective regression testing of Android apps based on code impact analysis and coverage information from rerunning

the original test suite. Impact analysis determines where the code has changed. Coverage information for each test case determines whether they reached the code that has changed and hence must be rerun. The implicit assumption is that none of the test cases identified for re-execution have become obsolete. There is also no attempt at determining new test cases.

By contrast, Chang et al. [27] emphasize repairing obsolete test scripts for Android Apps. They build a model of the possible GUI event sequences semi-automatically, first analyzing APK files of Android Apps, they extract information on the screen, and widgets, and even building the base model; second they execute the app to confirm the model. This uses base version test scripts to exercise the app and mark observed behaviors as feasible; third, they manually check the model to add missing behaviors. When changes to screens, widgets, or events occur, they are extracted by analyzing changes to the binary files. They use test simulation to identify which screens and widgets a test covers. This helps in identifying ranges within test scripts that need to be repaired. Once specific test actions in a test script have been identified as affected by changes, they are repaired. For added functionality, test scripts are extended by a simple random approach, triggering events on added widgets or screens. No coverage criteria are used.

Jiang et al. [58] aim to improve the safety of regression test selection. Safe regression testing approaches ensure that in the test selection, no tests are removed that would reveal a failure. Their approach concentrates on models of asynchronous task invocations, but also considers native code. This is a white box approach that uses impact analysis based on an inter procedural control flow graph that models what is considered dangerous edges. Test case selection uses edge coverage information for each of the original tests. Tests that cover edges flagged as dangerous by the impact analysis are selected.

Table 2.2: Shows a comparison of the four approaches. None addresses all aspects of selective regression testing. Models are either built via reverse engineering ([10], [27]) or code is directly analyzed ([27], [58]). When coverage criteria are used, they refer to white box coverage ([41], [58]). None of the papers address test prioritization or minimization.

Table (2.2) Approach comparison

Paper	Approach	Obsolete	Retestable	Reusable	Rerun tests	Coverage Criteria
[10]	Crawler				x	No
[41]	Code impact analysis, test coverage matrix		x			Yes
[27]	Byte code analysis, test coverage information	Repair				No
[58]	Inter procedural control flow graph, impact analysis, test coverage matrix		x			Yes

2.2.5 Combination of Regression Testing Approaches

So far we have only described regression testing using one of the three approaches. Theoretically, they could be combined in four possible ways: three for combining two approaches and one for combining all three. Table 2.3 summarizes the existing work that combines regression testing approaches. There are two types of combinations: combining test case prioritization and test case minimization [111, 106], and combining selective regression testing and test case prioritization [107, 103]. All are white-box techniques, hence not directly applicable to our research.

Table (2.3) Summary of existing combination approaches to regression testing

Paper	**Selective**	**Minimization**	**Prioritization**	type of testing
[106]		Minimizing the test cases using CMIMX, the path coverage is used	prioritize test cases by giving high priority to the test case that has a maximum number of changed statements.	White-Box
[111]		code coverage-based redundant test cases is used to minimize the test cases	prioritize test cases by using statements coverage (the test case that has the maximum number of cover statement has high priority)	White-Box
[107]	Select test cases based on impact analysis		using troubleshooting capabilities and coverage abilities criterion to prioritize test cases.	White-Box
[103]	Select test cases based on impact analysis		prioritize test cases based on different attributes such as: risk, change impact, and other user-defined properties associated with model elements.	White-Box

Prioritization and Minimization

Rehman et al. [111] investigate the impact of test case reduction and prioritization techniques on software testing effectiveness by analyzing previous empirical studies. They reviewed three experiential studies for test case reduction (coverage-based techniques) [50], [116], [95] to investigate testing effectiveness of TCR. And they reviewed [97], [42] coverage-based techniques, and [96],[40] distribution-based techniques to investigate testing effectiveness of TCP. Also, they presented a case study and suggested different useful combinations of these techniques. The first combination starts with test case reduction (TCR) and then test case prioritization (TCP). In this combination, first, they identify a set of test cases using TCR techniques then they determine their execution order using TCP techniques. They minimize the test suite by selecting the test case that covers the largest number of statements, then selecting the test case that covers the largest number of statements not covered by the first one, and so on. Next, they prioritize tests based on their statement-coverage. This implies that a test with high statement-coverage has a high probability of determining a fault. These priorities determine the execution order of all test cases.

Next combination, they start with test case prioritization (TCP) followed by test case reduction (TCR). In this combination, first, they identify high priority test cases utilizing TCP techniques and finally they determine the representative test cases utilizing TCR techniques from the prioritized test cases. From their experimental study, they observed some conditions, where the combination of these techniques was also helpful to improve the overall testing process. They make the following as conditions:

1. There is a risk of losing the fault-detection capability of test suites when using reduction techniques. The tester can avoid that by using prioritization techniques.

2. If there are limited testing resources and the project is behind schedule, TCR techniques are a good choice.

3. If there is adequate available time to run more test cases and fault detection is critical, prioritization techniques can be used.

In this study, they didn't consider selective regression testing. In our proposed approach we consider selective regression testing especially if there are obsolete tests and /or if additions have few overlaps with existing parts of the software under test.

Srivastava et al. [106] proposed a combination of test case prioritization and test case minimization techniques to improve the rate of fault detection and optimize the set of test cases. They first prioritize the test cases based on their importance. Prioritizing test cases by giving high priority to the test case that has a maximum number of changed statements. They used path coverage as a testing criterion. Next, they minimize the prioritized test cases by using the CMIMX procedure.

Nether [106], nor [111] considered selective regression testing as part of their combination. [106] does not address guidelines on how and when to select a particular combination of regression testing approaches.

Selective regression testing and Prioritization

Sun et al. [107] proposed an approach to combine selective regression testing with test case prioritization. Test cases would be selected during test case selection. The criterion of test case prioritization is coverage ability and troubleshooting capabilities

of test case. They first, select test cases to reduce the number of test cases based on impact analysis of programs which has been modified. Then they prioritize the selected test cases to improve the efficiency of finding faults.

Filho et al. [103] present a model-based regression testing and prioritization approach that selects test cases based on different concerns. They illustrate how to support these concerns using TDE/UML, an extensible model-based testing environment. Their approach includes different user-defined concerns such as the last change date, requirements, risk, and features, are represented as properties in the model. Furthermore, through traceability links between requirements, model, test cases, and code artifacts, these concerns are used to automatically select and prioritize test procedures, before they are used for code generation and execution.

Both [107] and [103] do not consider test case minimization as part of their combination. Also, they do not address guidelines on how and when to select a particular combination of regression testing approaches.

Chapter 3

A Unified Approach to Regression Testing

Our approach for regression testing of Mobile Apps is FSMApp combining regression testing. For that, we combined all three types of regression testing (selective regression testing, test minimization, and test prioritization). We extended the FSMApp approach [8] which is described in Subsection 2.3 and we adapted the roles in regression testing of FSMWeb [14] for selective regression testing of Mobile Apps. Figure 3.1 shows the combination of regression testing of the three of FSMApp processes. It includes three main steps:

- (1) Selective Regression Testing of FSMApp

  We performed selective regression testing in (Section 3.1).

- (2) Test Case Minimization of FSMApp

  We performed test case minimization regression testing (Section 3.2).

- (3) Test Case Prioritization of FSMApp

  We performed test case prioritization in (Section 3.3).

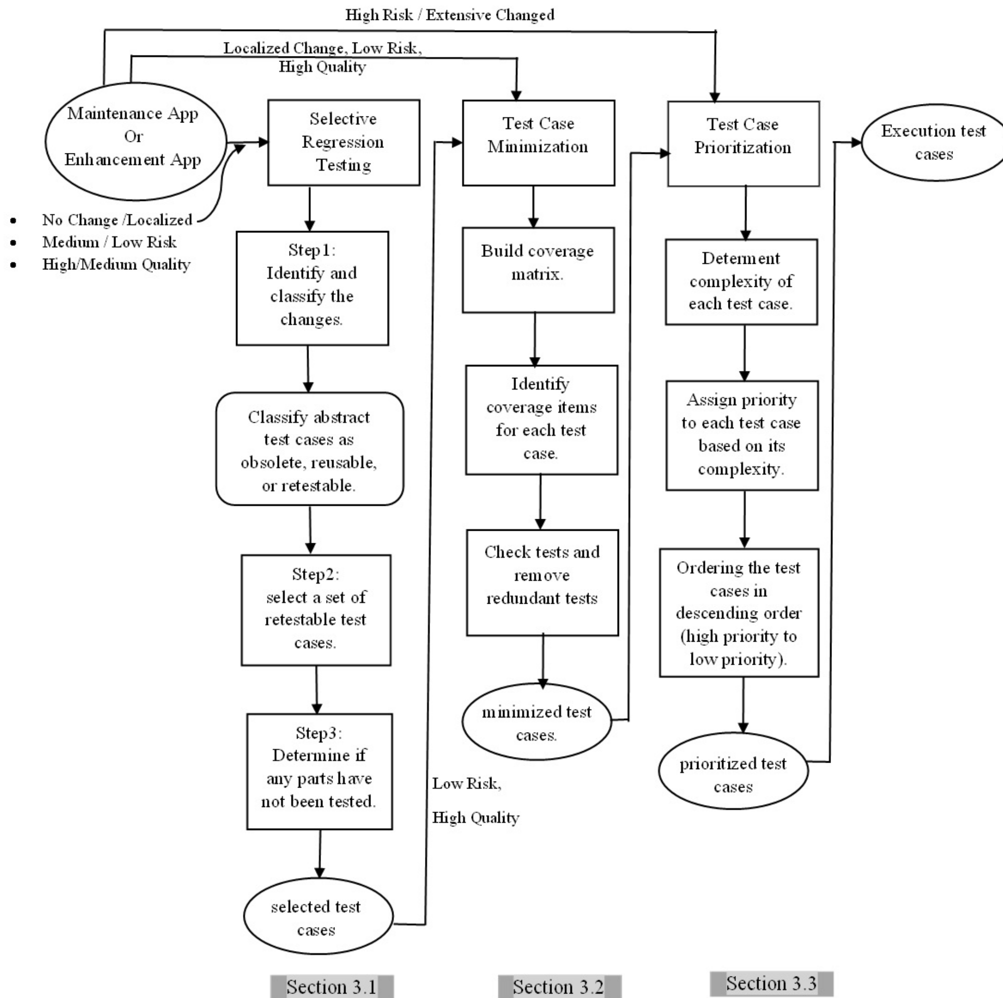


Figure (3.1) Combining Regression Testing of FSMApp Process

There is a connection between these steps. The output of the first step of the selective regression testing is used as input for the second step. (we minimized the selected test cases based on coverage criteria), and the output of the second step of the test case minimization is used as input for the third step. (we prioritized the minimized test suite based on the complexity of test cases).

3.1 Selective Regression Testing of Mobile Apps

We adapted the regression testing of FSMWeb [14] approach for selective regression testing for Mobile Applications. We applied rules to classify the original set of tests into obsolete, retestable, and reusable tests based on the types of changes to the model. New tests are added to cover portions of the App under test that have not been tested.

Before we apply selective regression testing, first we need to identify the changes in the model as:

1. No change in the model.
2. Localized change which means changes are isolated to a few partitions of the model, or
3. Extensive change when there are changes in many partitions of the model.

Then we apply selective regression testing with respect to test coverage criteria and aggregation coverage criteria. We need to identify what aggregation coverage no longer meets the requirements. The test coverage criteria [85] defined the test requirements, which then defined what parts of an FSM model should be tested according to the used criterion. Then we generate test paths that satisfy all the test requirements. A test path is a sequence of consecutive states that start with an initial state and end with a final state, each test path can satisfy multiple test requirements. Test paths can be constructed by creating one test path per test requirement, or by satisfying multiple test requirements in the same path.

Offutt et al. [12] defined different test requirement criteria for graph coverage that testers can use to produce test paths including: Edge-pair coverage, Prime path coverage. Also, the coverage criteria include Node coverage, Edge coverage, Complete coverage, and Specified path coverage. In our work, we used Edge coverage: where each edge has to be covered in test paths.

Moreover, they define aggregation criteria which are used when we need to consider multiple partitions at the same time, which is used for aggregation test requirements. Aggregation criteria include: All combination coverage and at least one coverage. In our work, we used at least one coverage. If the path aggregation criterion is to use each path at least once, this requires that we need to use each test $t \in T_i$ at least once when replacing C_i with $t \in T_i$. This means starting with the top level model (AFSM), we replace a cluster node C_i in a path with one of the test path $t \in T_i$ through it. This ultimately results in a test path from the start node in AFSM to the end node in AFSM that only contains LAP nodes. Aggregation criteria address which cluster paths need to be used.

Determine the amount of the changes as:

- Extensive change: when there are changes in many FSM_i and/or at high levels of the hierarchy (many parts of the model are changed), this is extensive change and will consider full regeneration.

  In this case, we need to rebuild the HFSM and identify test coverage criteria and aggregation coverage criteria to generate new abstract test paths. Since there is no test suite to classify or minimize it, the test case prioritization is a better choice for regression testing of the App under test.

- No change: when there are no changes in the model, we need to identify parts of the affected model to determine retestable tests.

  In this case, If testers have no model changes that means testers have corrective maintenance that we still need to identify the part that affects the model by changes and then do selective regression testing in terms of retestable tests. We don't need to generate new tests, so the task of doing selective regression testing is going to be easier because we classify the test cases and we only select subsets that are retestable to rerun.

- Localized change: the case is when changes are isolated to a few FSM_i (few partitions of the model are changed). We will consider partial regeneration.

  In this case, because of the hierarchical nature of the FSMApp models and the associated stages of test generation, it is possible to use partial regeneration to replace obsolete test cases. Selective regression testing will be used as explained in the next section.

3.1.1 Selective Regression Testing Process

The regression testing approach for FSMWeb [14] is the basis of the Selective Regression Testing approach for FSMApp. In regression testing of FSMWeb [14], types of changes to the FSMWeb models are formalized based on changes to the web application (its inputs, Logical Web Pages, and navigation between them). The approach represents an abstract test case as a sequence of states and input predicates. An abstract test case t_a is a test path through the HFSM. In the FSMApp model, we adapted the classification roles in [14] as follows:

- Determine the type of changes:
  - Node change: delete, modify (i. e. Change the node type from logical app page (LAP) to cluster or from cluster to LAP). Node addition is covered by edge changes.
  - Edge change: modify in/out edge (i. e. modify the edge's input-action constraint), delete in/out edge, add in/out the edge with or without new node.
  - Classify model changes as path affecting (PC) or non-path affecting (NC) change.
- Classify abstract test cases as obsolete, reusable, or retestable. Map this classification to the associated tests.
- Identify aggregation coverage criteria that no longer meet the requirements. We will use at least one aggregation coverage criteria in our work because we do not need to use aggregation coverage that limit minimizing the test cases such as all combination.
- Rerun retestable tests. The set of retestable tests might provide duplicate coverage for some coverage items. This may reduce the selected regression tests.
- Identify new nodes and edges in HFSM that need to be covered. Regenerate paths through changed FSM'_i to cover uncovered edges.
- Determine new test cases and execute them. Reaggregate abstract tests that contain the cluster node that stands for FSM_i with the new paths.

The following Table 3.1 shows the notations and conventions used:

Table (3.1) The Notations and Conventions

Notations	**Explanation**
e:	an edge in one of the FSMs in an HFSM. $E = \{e\}$ is the set of all such edges.
n:	a node in one of the FSMs in an HFSM. $N = \{n\}$ is the set of all such nodes.
e′:	an edge in one of the FSMs of HFSM′ (created through modifications to HFSM). E′ are all such edges.
n′:	a node in one of the FSMs of HFSM′ (created through modifications to HFSM). N′ are all such nodes.
n_{source}, n_{sink}:	source and sink nodes of the AFSM; i. e. nodes in the AFSM that have either no incoming or no outgoing edges.
$\hat{e}$:	refers to an edge in one of the FSMs in HFSM that is changed.
$\hat{n}$:	refers to a node in one of the FSMs in HFSM that is changed.
$t_a tours$, $e = (n_i, n_j)$:	means that there is a subsequence $n_i; I_{ij}; n_j$ in ta where $(n_i, n_j) \in E$ is an edge from node n_i to node n_j and I_{ij} is the input constraint annotation on the edge.

The test classification rules are as follows [14]:

Obsolete Test Cases

They are caused by node deletion and node modification. All paths that visit the deleted node are affected. Node modification changes the type of node from LAP to cluster node or vice versa. This type of change also affects paths, since either the node needs to be replaced by a path through the FSM associated with the cluster (change from LAP to cluster node) or a sequence of nodes corresponding to a path through the cluster that was modified to a LAP node needs to be deleted for the abstract test case to be valid for HFSM′.

Let $N_o = \{n|n \in N$; n is deleted or modified node$\}$, then the set of obsolete abstract test cases due to node changes is given by $O_N = \{t_a|\exists n \in N_o : t_a$ visits n$\}$.

Edge deletion renders any path that visits the edge obsolete. Edge modification involves modification of the input constraint associated with it. This could mean adding or removing inputs, or modifying the type of input or action. Any of these will make abstract test cases obsolete.

Let $E_o = \{e|e \in E$; e is deleted or modified$\}$, then the set of obsolete test paths due to edge changes is given by $O_E = \{t_a|t_a$ tours $e \in E_o\}$. Hence, the set of obsolete test cases is given by $O_{AT} = O_N \cup O_E$.

Retestable Tests

Retestable tests are still valid and test parts of the application that might be affected by the change. These are abstract test cases that visit changed parts of the FSMApp model. For example, it would be reasonable that any node n that is one edge away from a modified or deleted node must be retested, not including the source and sink nodes of the AFSM:

$N_{r_{node}} = \{n|\exists e : (\hat{n}, n) or (n, \hat{n}); \hat{n} \in No; n \neq n_{source}; n \neq n_{sink}\}$.

When edges are changed (deleted or modified), the beginning and ending nodes of the changed edges are potentially affected and thus non-obsolete tests that visit these nodes are retestable (Except the source and sink nodes of the AFSM):

$N_{r_{edm}} = \{n|\hat{e} \in E_o; \hat{e} = (n_i, n) \text{ or } \hat{e} = (n, n_i); n \neq n_{source}; n \neq n_{sink}\}$.

Likewise, when edges (and possibly nodes) are added, existing nodes at which these new edges start, or end is considered potentially affected by the change and thus non-obsolete tests that visit these nodes are retestable (Except for the source and sink nodes of the AFSM):

$N_{r_{ea}} = \{n|\exists \text{ n}' \in \text{N}'\text{: e}' = (n, \text{ n}') \text{ or e}' = (\text{n}' , n); n \neq n_{source}; n \neq n_{sink}\}$.

n appears in both HFSM and HFSM' related to edge changes. The set of retestable nodes is then given by $N_r = N_{r_{node}} \cup N_{r_{edm}} \cup N_{r_{ea}}$ and the set of retestable abstract test cases is $R_{AT} = \{t_a|t_a \text{ visits } n \in N_r \; O_{AT}\}$.

Corresponding executable retestable tests are determined as a function of RAT as before.

$RT = \{t|\exists ta \in R_{AT} : t \in T_a\}$.

Reusable Tests

Reusable tests are neither obsolete nor retestable.

$U_{AT} = AT \; (O_{AT} \cup R_{AT})$

$U_T = T \; (O_T \cup R_T)$.

New Tests

New tests are determined for edges that are not covered. since the existing test cases do not cover each change, particularly edge and node additions, so need to generate new test cases. Obsolete test cases leave gaps in coverage, so every obsolete test case needs to be addressed with one or more new tests.

3.1.2 Example Used to Illustrate Approach

We used To Do App example to illustrate our selective regression testing approach. To Do is a simple app to make lists of tasks. The app has these services: add tasks, modify tasks (delete tasks, edit tasks, mark for a task done or undo), and search for a task.

We first applied the FSMApp approach for testing Mobile Apps which is explained in Subsection 2.1.2 to get abstract test cases for the original version of the To Do App. Then we made some modifications to To Do App to apply our approach of selective regression testing on the modified version.

1. FSMApp Model for To Do App Example:

Phase 1: Build Model

a) Partition the To Do app into Clusters:

The term cluster is referring to collections of software modules/app pages that represent a logical or user level function. Clusters represent functions that can be identified by users, at the highest level of abstraction. $HFSM = \{FSM\}_{i=0}^{n}$ with a top level $FSM_0 = AFSM$. Each FSM has nodes that represent either clusters or Logical App Pages (LAPs). Edges are internal or external to an FSM. External nodes span cluster boundaries. External edges can either enter or leave a cluster FSM [8]. The clusters might be an individual Activity which is a screen that is shown to the application user, or software modules that represent a major function [16]. In this example, the app has three subsystem clusters. Figure 3.2 shows the main screens for the To Do App. Figure 3.2 (Main Screen) shows the main screen for the To Do App, Figure 3.2 (a) shows the screen to add tasks, Figure 3.2 (b) shows the screen to modify the task, and Figure 3.2 (c) shows the screen to search

for a task. The main screen is classified into three clusters as shown in Figure 3.3. The AFSM models entry and exit from the app as well as navigation between its three main functions. The clusters (Add Task) and (Modify Task) are described in detail. Figures 3.5 and 3.6 show the navigation among the LAPs.

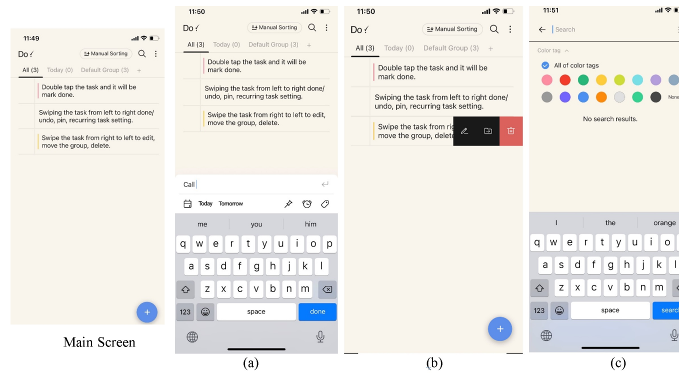


Figure (3.2) Main Screens for To Do App

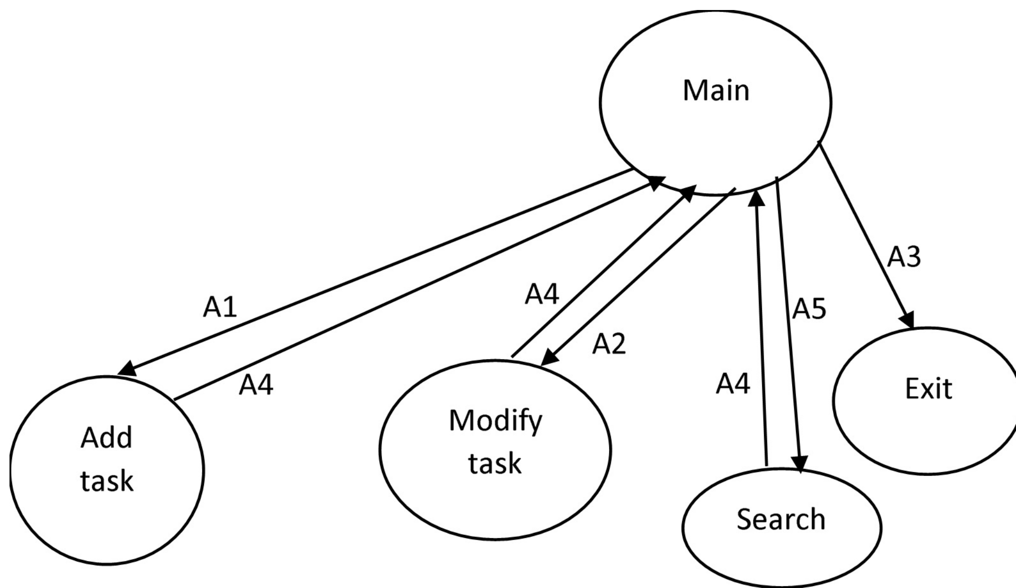


Figure (3.3) Main page (AFSM)

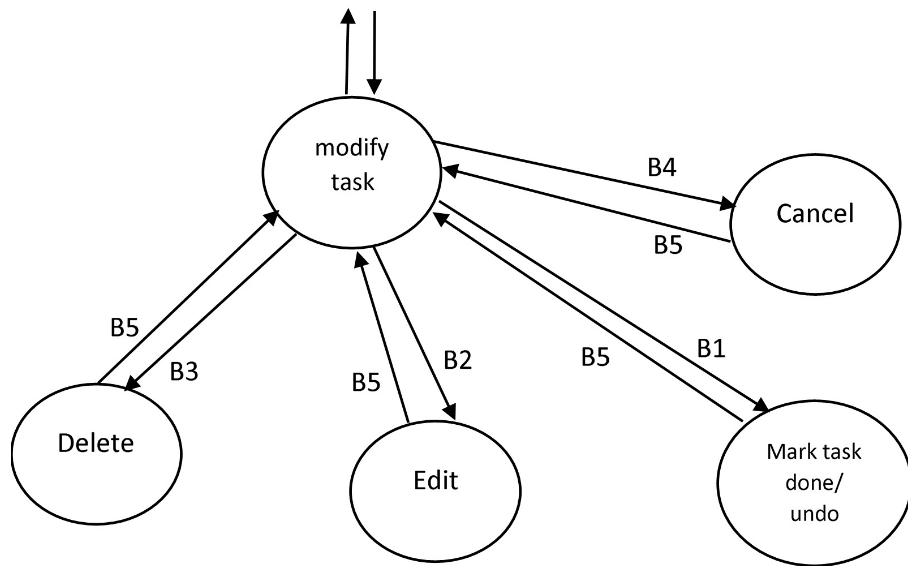


Figure (3.4) Modify Task Cluster

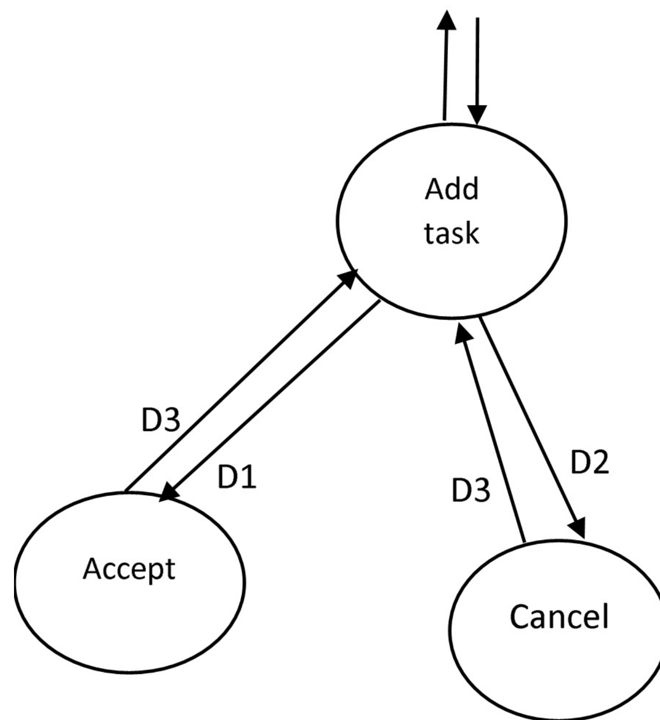


Figure (3.5) Add Task Cluster

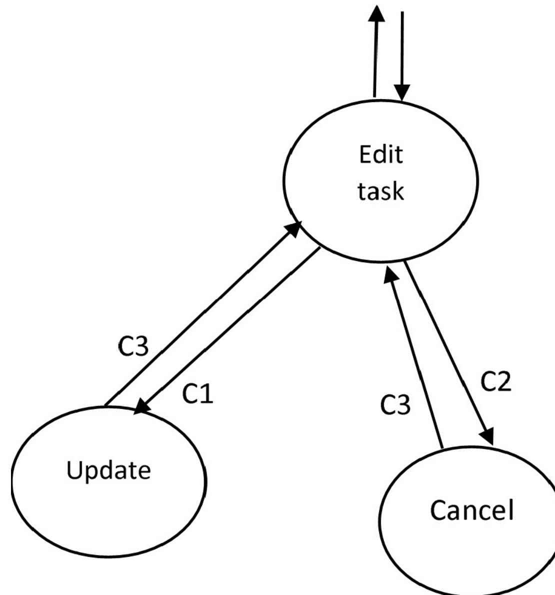


Figure (3.6) Edit Task Cluster

b) Define Logical App Pages (LAPs) and Input-Action Constraints

The Mobile Apps' screens will be considered as input components, or LAPs, and the inputs and their constraints on these LAPs next. There are several app actives (screens). A Logical App Page is either a physical app page, physical app component, or the portion of an app activity that receives data from the user via an XML form, and then sends the data to a particular software module [8]. The inputs might have rules to enter: some inputs may be required, others may be optional, users might be permitted to enter inputs in any order, or a specific order may be required. Example constraints on inputs and their meaning are:

Required (R): required input must be entered.

Required Value (R(parm)): one must enter at least one value.

Optional (O): an input may or may not be entered.

Single Choice (C1): one input should be selected from a set of choices. Multiple choices (Cn) are possible.

Mobile Apps can have a variety of components that can be modeled via input constraints. The difference between the web input types and Mobile input types is swipe and scroll. We explained the types of input (components) and what FSMApp's input constraints we need to use in our example. For the full list of input constraints see Appendix A.

A button: the action of the button is click. The input constraints are R($< Click >$), or select the button then click C1(Select Button, Click). In our example need to Click Add button to add a task.

Pickers: provide a way to select a single value from a pre-determined set. The actions of the pickers are dropdown and click. The input constraint is C1($< Dropdown >, < Click >$). In our example, time and date pickers for doing the task.

Lists: produce multiple line items in a vertical arrangement. The action of the list is sort. The input constraint is R($< Sort >$). In our example list of tasks.

A card: is a sheet of material with unique related data to show more detailed information. The actions of the card are: click, swipe, scroll, and pick-up-and-move. The input constraint is C1($< Click >, < Swipe >, < Scroll >, < Pick >$). In our example need to swipe the task from right to left to edit or delete it.

Edges represent transitions that connect the nodes and the clusters. Transitions are annotated with input constraints to indicate what inputs and actions lead to the next node or cluster. Figure 3.7 shows the Add Task input-action constraints. In the Add Task cluster, there are two states: either you enter the new task info (parTask) and accept it (b-Add), or you cancel the new Task (with or without giving the Task info). Incoming and outgoing edges for this cluster connect to the parent cluster. They don't require any user actions. We also added two dummy transitions to keep the graph single-entry-single exit.

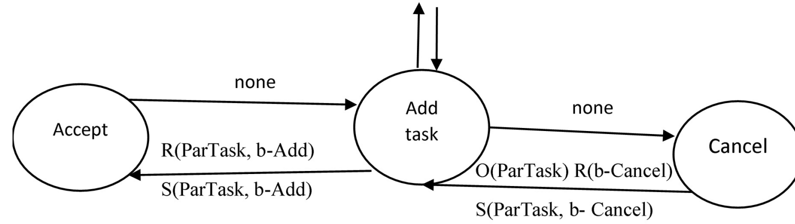


Figure (3.7) Annotated FSM for Add Task Cluster of Table 3.4

Tables 3.2 to 3.5 show the transitions, explanation, and input action constraints for To Do App example. Column 1 identifies each transition. Column 2 shows an explanation of the transition. Column 3 shows all input action constraints with all required or optional inputs. The corresponding graphs are mentioned in the caption of the tables. Table 3.2 shows five transitions for the main page cluster (AFSM). The transitions connect the main page with three clusters and one LAP (Exit App).

Table (3.2) Transitions of Figure 3.3 (Main Page Cluster AFSM)

Transition	**Explanation**	**Constraints**
A1	Add new task	R(buttonANT)
A2	Access Modify task	R(Swiping the task from right to left)
A3	Exit the System	R(buttonBack)
A4	Back to Main Page	none
A5	Search	R(b-search)

Table (3.3) Transitions of Figure 3.4 (Modify task Cluster)

Transition	**Explanation**	**Constraints**
B1	Mark task done/undo	R(double tap the task)
B2	Edit the task	O(Parname, partask) R(b-Edit) S(Parname, partask, b-Edit)
B3	Delete the task	O(Parname, partask) R(b-Delete) S(Parname, partask, b-Delete)
B4	Cancel to Previous Page	O(Parname, partask) R(b-Cancel) S(Parname, partask, b-Cancel)
B5	Back to Main Page	none

Table (3.4) Transitions of Figure 3.5 (Add task Cluster)

Transition	**Explanation**	**Constraints**
D1	Accept the task	O(Parname, partask) R(b-Add) S(Parname, partask, b-Add)
D2	Cancel to Previous Page	O(Parname, partask) R(b-Cancel) S(Parname, partask, b-Cancel)
D3	Back to Main Page	none

Table (3.5) Transitions of Figure 3.6 (Edit task Cluster)

Transition	Explanation	Constraints
C1	Update the task	O(task information) R(b-update)
C2	Cancel to Previous Page	O(Parname, partask) R(b-Cancel) S(Parname, partask, b-Cancel)
C3	Back to Main Page	none

Phase 2: Generate Test Sequences

a) Paths through FSMs/AFSM:

Here test sequences are generated. The user can select coverage criteria such as node, edge, edge-pair, simple round trip, and prime path coverage [12]. A test path is a sequence of nodes through the aggregate FSM and through each lower-level FSM. In this example test sequences for clusters are generated by applying standard graph criteria, edge coverage [12]. Tables 3.6 to 3.9 show test paths for each cluster of the To Do App as sequences of nodes. Nodes in bold indicate the node is a cluster node.

Table (3.6) Main Page Test Sequences of Figure 3.3

ID	Test Sequence	Length
1	[Main, **Add Task**, Main, Exit]	4
2	[Main, **Modify Task**, Main, Exit]	4
3	[Main, **Search**, Main, Exit]	4

Table (3.7) Modify Task cluster Test Sequences of Figure 3.4

ID	Test Sequence	Length
1	[Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Cancel, Modify Task]	7
2	[Modify Task, **Edit Task**, Modify Task]	3

Table (3.8) Add Task cluster Test Sequences of Figure 3.5

ID	Test Sequence	Length
1	[Add Task, Accept, Add Task, Cancel, Add Task]	5

Table (3.9) Edit Task Cluster Test Sequences of Figure 3.6

ID	Test Sequence	Length
1	[Edit Task, Update, Edit Task, Cancel, Edit Task]	5

b) Path Aggregation

The test paths are aggregated into test sequences. Several aggregation criteria have been proposed: all-combinations, each choice, and base choice coverage [12]. We assume that the path aggregation criterion is to use each path at least once, which requires that aggregate paths from the start of the application to the termination of the application are formed such that every path generated for a lower-level FSM is selected at least once on the same path [12]. The result of this process is a set of aggregate paths named abstract tests. For example, the first test path in AFSM [Main, **Add Task**, Main, Exit] in Table 3.6 can be aggregated as follows: The test path has one cluster node (Add Task). The cluster node should be replaced by the Add Task cluster test paths from Table 3.8. The result of this step is this abstract test path: [Main, Add Task, *Accept*, Add Task, *Cancel*, Add Task, Main, *Exit*].

Table 3.10 shows the aggregated test paths of the To Do app as sequences of nodes.

Table (3.10) The aggregated Test Paths of the To Do App

ID	Test Sequence	Length
1	[Main, Add Task, *Accept*, Add Task, *Cancel*, Add Task, Main, *Exit*]	8
2	[Main, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Main, *Exit*]	8
3	[Main, Modify Task, Edit, *Update*, Edit, *Cancel*, Edit, Modify Task, Main, *Exit*]	10
4	[Main, Search, Main, *Exit*]	4

c) Reduction Step:

Dummy nodes and transitions are added when the model is built to ensure single-entry and single-exit graphs. They do not require any inputs as they are not really testing steps. So, these dummy nodes, and transitions will be removed while each remaining node pair (edge) is replaced with its corresponding input action constraint.

Phase 3: Selects the inputs to create abstract tests.

In this step, inputs are selected to replace the input constraints in the test paths constructed above. The test designer selects values for related inputs. In our example, we select inputs to replace the input constraints in the test sequence constructed for To Do App. At the end of this step, we have a set of inputs for the execution phase. Table 3.11 shows the set of inputs for test path1 of Table 3.10. Column one shows the edges, column two shows the constraint sequence, column three shows the input values that meet the constraints, and the last column

explains each value. Test 1 has six inputs and five actions. The inputs are Task name (parname) and Task (Date and time) (parD), (ParT) which occurs twice in test1. The actions are to click Add New Task button (b-ANT), Click Accept button(b-ANTA), Click Cancel (b-Cancel), and Click the back arrow to exit the Mobile App (buttonBack).

Table (3.11) Test Path 1 with Input Values

Edge	Constraint	Value	Explanation
A1	R(b-ANT)	buttonANT= Click	Push AddNewTask to get screen of adding task information.
D1	R(parm(name, D, T), b-ANTA) S(parm(name, D, T), b-ANTA)	Taskname="AA" D=2/2/2022 T=12:00Pm b-ANTA = Click	Enter info task then Push button addnew-task to accept adding info of the task.
D2	O(parm(name, D, T)), R(b-Cancel) S(parm(name, D, T), b- Cancel)	TaskInfo="AA" D=2/2/2022 T=12:00Pm b-ANTC = Click	Enter info task then Push cancel button to cancel adding info of the task.
A4	R(buttonBack)	buttonBack = click	Push back arrow to exit the app

2. Applying Selective Regression Testing of FSMApp for To Do example:

We modify the To Do App that we used in the example above to illustrate selective regression testing needs. We will enhance To Do App by adding a new feature to the app, which is "record task".

Step 1: Identify and Classify Changes for To Do App

First, we will identify types of changes in the To Do App (additions, deletions, or modifications) and determine what portions of the To Do App are affected. Then classify existing test cases as retestable that means must be rerun, reusable will not

rerun them because they do not test parts of the To Do App that are affected by the changes (but will still work), or obsolete, tests that do not work any longer. We apply the rules to determine the type of changes for To Do App and rebuild the modified parts of the To Do App.

Determine types of changes to To Do App: The modifications that we did to enhance the app are as follows:

Modification 1: Node deletion, edge/node additions

We delete Add Task node. Then we add a new cluster node, that will give us options to add new tasks by recording information about the task or by writing the information about the task. So, the functionality changes require additional nodes and edges that could not only be added task information but allow to record of task information. Delete Add Task node, adding the new cluster node (Task), and adding a new FSM for the Task cluster will make the abstract tests associated with aggregate path AP1 in Table 3.10 effected path (PC). Figure 3.8 shows the main model after the modification, and Figures 3.9 to 3.11 show the new clusters and subclusters of the modified App.

Modification 2: Edges additions

Adding edges between cluster node Task and cluster node Modify Task. These edges allow modifying tasks without returning to the main page of the App. This modification makes aggregate paths AP2 and AP3 in Table 3.10 affected paths (PC). Figure 3.8 shows this modification.

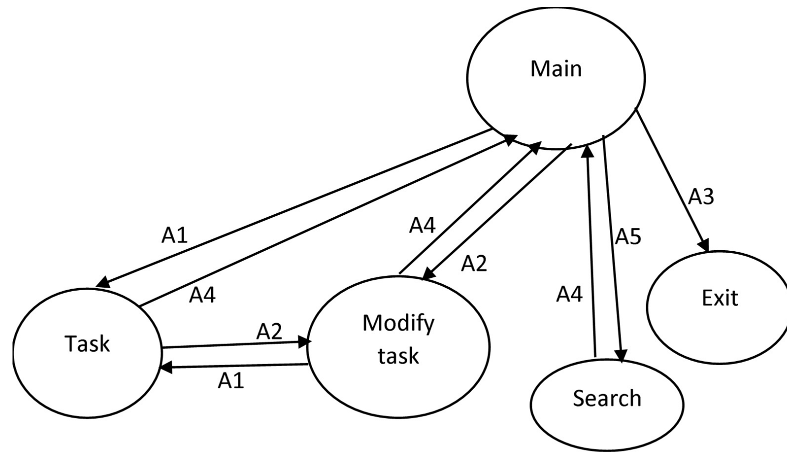


Figure (3.8) Modified Main Page (AFSM)

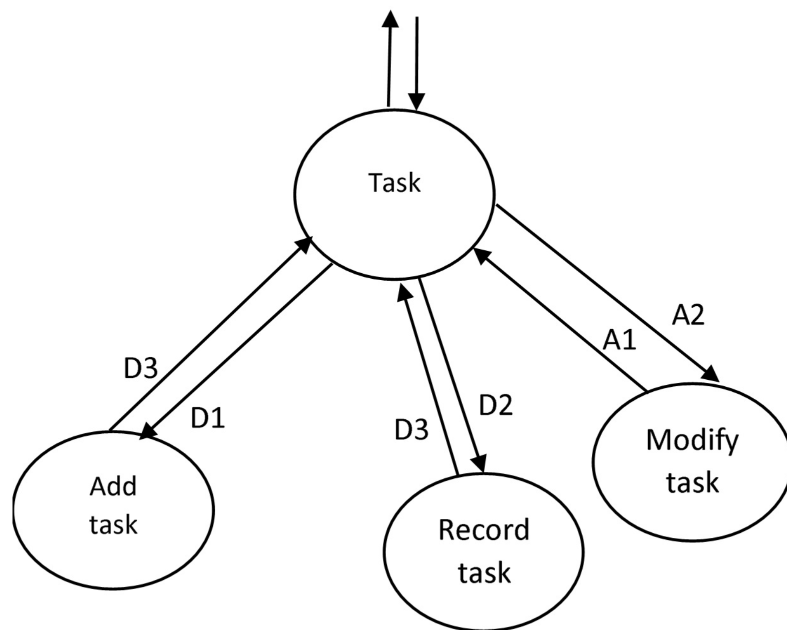


Figure (3.9) Task Cluster

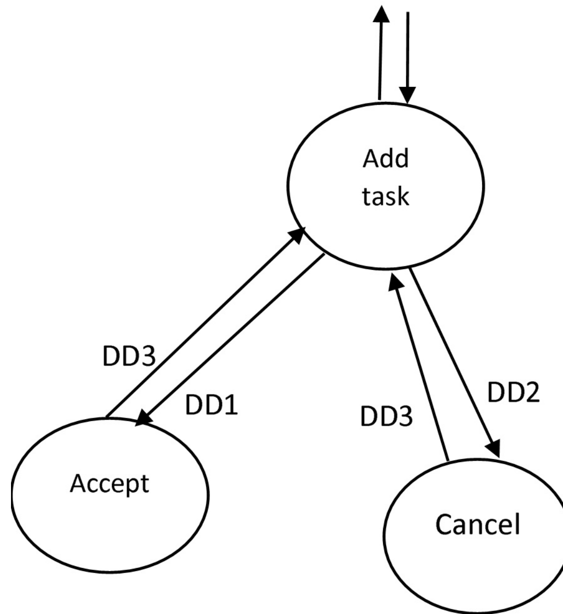


Figure (3.10) Add Task Cluster

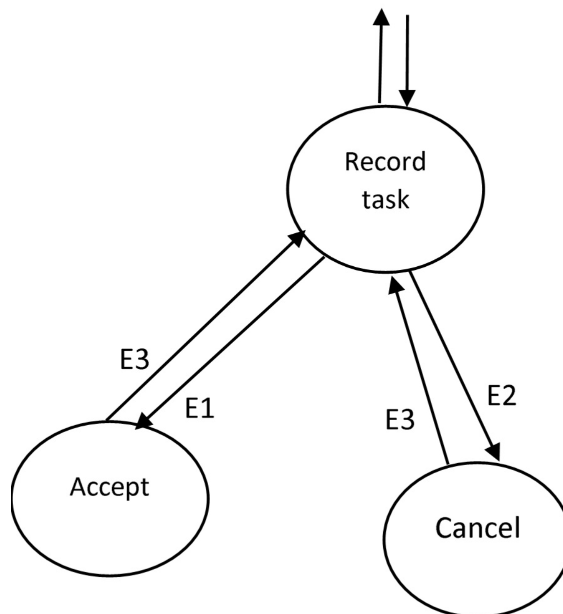


Figure (3.11) Record Task Cluster

We determine the types of changes for the To Do App and rebuild the modified model. Also, we classify model changes as path affecting (PC)= 1 to 3 test paths in Table 3.10, and non-path affecting (NC)= test path 4.

Step 2: Classify abstract test cases into: obsolete, retestable, and reusable.

We apply the set of rules from subsection 3.1.1. obsolete: deleted and modified node makes any test paths that visit the node obsolete. In our example, We deleted the Add Task node, So, $N_o = [AddTask]$. This makes the abstract tests associated with aggregate path AP1 in Table 3.10 obsolete. Edge deletion and edge modification make any test paths that visit the edge obsolete. In our example, we modify edge A1. This renders test paths that visit this edge obsolete. $E_o = (Main, AddTask)$, in test case 1 in Table 3.10.

Retestable: retestable tests are defined as those that are still valid and test portions of the App and visit part of the FSMApp model that are affected by the changes. In our example E$'$ $\backslash E$ ={ (Task, ModifyTask), (ModifyTask, Task)}. N_{rea} = {Task node, ModifyTask node}. So, the retestable tests are test cases 2 and3 in Table 3.10.

Reusable: reusable tests are neither obsolete nor retestable . In our example, reusable tests (test case4 in Table 3.10).

Algorithm 5 shows the procedure to classify the affected test paths into: obsolete and retestable test cases. The inputs of the algorithm are (deleted and modified nodes) N$'$, (deleted and modified edges) E$'$, and the number and affected test paths (that we got when we apply rules to determine the type of changes). The output of the algorithm 5 are sets of test cases: obsolete and retestable. The first loop in line 1, will go through all affected test paths. The loops in lines 3 and 4 iterate through every test path to sequentially check each node in the path whether it deleted or modified node. If there is a deleted or a modified node in the path, then add this test path into obsoletList. In line 11 the algorithm will check whether the path added to obsoleteList. If not added to obsoleteList, the algorithm will

go to the loops in lines 12 and 13 to check whether the path has a deleted or a modified edge, then add it to obsoleteList, but if the test path does not have a (deleted or modified) node or (a deleted or a modified) edge, the algorithm will add this test path to retestableList. Then, the algorithm goes to check another affected test path. The algorithm will continue this process until finish all affected test paths.

Input: N′ (nodes changes) and E′ (edges changes), affected paths (abstract test cases)
Result: obsoleteList = obsolete test cases, retestableList= Retestable test cases

```
1:  for i = 1 to num do
2:     Path = F
3:     for j = 1 to Length(P_i) do
4:        for k = 1 to num(No) do
5:           if nodei == n′_k then
6:              add Pi to obsoleteList
7:              Path = T
8:           end if
9:        end for
10:    end for
11:    if Path ≠ T then
12:       for x = 1 to Length(P_i)- 1 do
13:          for m = 1 to num(Eo) do
14:             if edge(nodex, nodex+1) == e′_m  then
15:                add P_i to obsoleteList
16:                Path = T
17:             end if
18:          end for
19:       end for
20:    else
21:       add P_i to retestableList
22:    end if
23: end for
```

Algorithm 5: Classify Test Cases

We applied the algorithm to classify affected test paths in our example To Do App into obsolete and retestable tests. The algorithm will take every affected test

path and check if it obsolete or retestable test path. For example, it takes test path 1 [Main, Add Task, *Accept*, Add Task, *Cancel*, Add Task, Main, *Exit*], then the algorithm checks each node and edge in the test path 1 whether it (deleted or modified node) or (deleted or modified edge). Since this path has a modified node (Add Task), so the algorithm will move this test path into obsoletList. Then it gets another test path to check if it (obsolete or retestable). The algorithm will continue this process until checks all affected test paths. The output of the algorithm in our example is:

- One obsolete test case: test case 1 in Table 3.10
- Two retestable test cases: test cases 2 and 3 in Table 3.10

Step3: Select Set of Retestable Test Cases

From the results, the retestable test cases for To Do app are: (test case2 and 3 in Table 3.10). We select the set of retestable test cases to use them to reveal change-related faults in HFSM′ for To Do app. Retestable Test Cases are:

1. [Main, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Main, *Exit*]
2. [Main, Modify Task, Edit, *Update*, Edit, *Cancel*, Edit, Modify Task, Main, *Exit*]

Step4: Determine if any parts have not been tested in To Do app:

We needed to determine if any parts of the system have not been tested and generate a new set of tests. The hierarchical characteristic of FSMApp allows doing partial test path regeneration in the face of changes. Since the Obsolete test case (test case1 in Table 3.10) leaves gaps in coverage, needs new tests. The paths of the model that are not covered are related to the test path [Main, **Task**, Main,

Exit] in AFSM′. We needed to do partial regeneration for test paths. Since we have a hierarchical set of models, and test generation proceeds in stages, so, the partial regeneration can start at any of the regeneration steps: the finite state machine levels, the path aggregation phase, or the input value selection phase. In our example, we got an obsolete test case when the Add Task node changed from a LAP node to a cluster node. The obsolete test cases related to Add Task can be fixed by substituting paths through the new FSM Add Task for the old Add Task node and regenerating the test from that point on (i. e. path generation through a lower level FSM and path aggregation).

The paths that we needed to regenerate were related to the [Main, **Task**, Main, Exit] on the main page. Tables 3.12 to 3.15 show the transitions, explanation, and input action constraints for part of To Do App after the modification.

Table (3.12) Main Page Test Sequences of Figure 3.8

ID	Test Sequence	Length
1	[Main, **Task**, Main, Exit]	4

Table (3.13) Task cluster Test Sequences of Figure 3.9

ID	Test Sequence	Length
1	[Task, **Add Task**, Task]	3
2	[Task, **Record Task**, Task]	3
3	[Task, **Modify Task**, Task]	3

Table (3.14) Add Task cluster Test Sequences of Figure 3.10

ID	Test Sequence	Length
1	[Task, Add Task, Accept, Add Task, Cancel, Add Task, Task]	7

Table (3.15) Record Task cluster Test Sequences of Figure 3.11

ID	Test Sequence	Length
1	[Task, Record Task, Accept, Record Task, Cancel, Record Task, Task]	7

Then, we needed to apply the algorithm of aggregate test paths to get new abstract test cases for the modification part of To Do App. To aggregate paths, we assumed that the path aggregation criterion is to use each path at least once, which requires that aggregate paths from the start of the application to the termination of the application are formed such that every path generated for a lower-level FSM is selected at least once on the same path [12]. Algorithm 6 shows the procedure to aggregate test paths for new models. The inputs to the algorithm are $AFSM'$ and cluster test paths. The output of algorithm 6 is a set of aggregated test paths (abstract tests). In line 1, the algorithm put $AFSM'$ test paths into an input List, in our example it will put just one test path (test path1 in Table 3.12). Then line 2 iterates through every test path from the input List. Line 3 takes one test path from the input List. Then, it sequentially checks each node in the path whether it is a cluster node. If there is a cluster node in the path, then a loop replaces the cluster node with each cluster path and creates as many new partially aggregated paths as there are paths through this cluster node. The output of the algorithm is 4 new test paths as follows:

1. [Main, Task, Add Task, *Accept*, Add Task, *Cancel*, Add Task, Task, Main, *Exit*]

2. [Main, Task, Record Task, *Accept*, Record Task, *Cancel*, Record Task, Task, Main, *Exit*]

3. [Main, Task, Modify Task, Edit, *Update*, Edit, *Cancel*, Edit, Modify Task, Task, Main, *Exit*]

4. [Main, Task, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Cancel, Modify Task, Task, Main, *Exit*]

This set of new test cases covered the coverage gap left by the obsolete test case 1 in Table 3.10.

```
Input: AFSM' and Cluster Test Paths
Result: outputList = Set of Aggregated Paths
 1: inputList = AFSM' affected Paths
 2: while inputList has next path do
 3:    currentPath = get one path from inputList
 4:    pathDone = true
 5:    for i = 1 to Length(currentPath) do
 6:       if nodei is cluster node then
 7:          for j = 1 to Length(cluster paths) do
 8:             Replace nodei with cluster pathj and add new path into inputList
 9:          end for
10:          add list paths to inputList
11:          remove currentPath from inputList
12:          i = length (currentPath) + 1
13:          pathDone = false
14:       end if
15:    end for
16:    if pathDone is true then
17:       Move currentPath into outputList
18:    end if
19: end while
```

Algorithm 6: Aggregated Affected Test Paths

Table 3.16 shows the abstract test cases (retestable tests, new tests) that we need to test To Do App after the modification.

Table (3.16) The aggregated test paths of the To Do app

ID	**Test Sequence**	**Length**
1	[Main, Task, Add Task, *Accept*, Add Task, *Cancel*, Add Task, Task, Main, *Exit*]	10
2	[Main, Task, Record Task, *Accept*, Record Task, *Cancel*, Record Task, Task, Main, *Exit*]	10
3	[Main, Task, Modify Task, Edit, *Update*, Edit, *Cancel*, Edit, Modify Task, Task, Main, *Exit*]	12
4	[Main, Task, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Cancel, Modify Task, Task, Main, *Exit*]	13
5	[Main, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Cancel, Modify Task, Main, *Exit*]	10
6	[Main, Modify Task, Edit, *Update*, Edit, *Cancel*, Edit, Modify Task, Main, *Exit*]	10

Now, we need to select inputs to replace the input constraints for the To Do App. Then, we will have a set of inputs for the execution phase. Table 3.17 shows the set of inputs for the new test path1 of Table 3.16. Test 1 has six inputs and five actions. The inputs are Task name (parname), Task Date and time (ParD), (ParT) which occurs twice in test1. The actions are to click Add New Task button (b-ANT), Click Accept button(b-ANTA), Click Cancel (b-Cancel), and Click the back arrow to exit the Mobile App (buttonBack).

Table (3.17) New Test Path 1 with Input Values

Edge	Constraint	Value	Explanation
A1	R(b-Task)	buttonTask= Click	Push Task to select adding task info or record task info.
D1	C1(SelectAddInfo, SelectRecodInfo)	SelectAddInfo= ANT	select AddNewTask to get screen of add.
DD1	R(Par(name,D,T), b-ANTA) S(Par(name, D,T), b-ANTA)	name="AA" Date=9/23/2021 Time=10:00 Am b-ANTA= Click	Enter info of the task then Push button addnewtask to accept adding info the task.
DD2	O(Par(name, D, T), R(b-Cancel) S(Par(name, D, T) b- Cancel)	name="AA" Date=9/23/2021 Time=10:00 Am b-ANTC =Click	Enter info of the task then Push cancel button TO cancel adding info of the task.
A4	R(buttonBack)	buttonBack = click	Push back arrow to exit the app

Step 5: Execute retestable test cases and new test cases:

In this phase, the test cases can be executed manually or using an automatic tool. For Mobile Apps, many automatic tools [31] are available to run the test cases. For example, tests can be converted to Selenium [91] or any of the other candidate execution environments. Selenium is an open-source software testing framework for web and Mobile Applications. It provides a test domain-specific language to write the tests such as Java, Python, and PHP. If we use Selenium, the Appium [48] [71] server executes the Selenium code and reports results. Appium is open-source and easy to set up.

Since we needed to apply the combination of regression testing approaches (selective, minimization, and prioritization) for Mobile Apps, we minimized the selected test cases of the To Do App and prioritize them before executing them.

3.2 Test Case Minimization

As yet, FSMApp has not addressed regression testing in general, nor test case minimization in particular. We developed a test case minimization approach for FSMApp. FSMApp generates a test suite T that satisfies test requirements based on graph coverage criteria and test aggregation criteria for its hierarchical model. Together, these constitute a set of test requirements R. The method described in this section solves the following problem.

Given a test suite T and a set of test requirements, R finds a minimal subset of T that covers R [109]. Since this is known to be NP-complete [69], we cannot always guarantee an optimal solution. Depending on the test requirements, it also is not always possible to reduce the test suite T. This dissertation explores both under which conditions minimization is not possible and what test coverage criteria are likely to lead to no reduction in tests.

Figure 3.12 shows the process of minimizing a regression test suite. We assumed that based on the original Mobile App and subsequent changes, a current FSMApp model $HFSM$ exists and that a current set of tests T and test requirements TR for this $HFSM$ is given. We also assumed that the test requirements are based on graph coverage and aggregation coverage as specified in FSMApp [8]. Based on the relationship between T and TR, a concept lattice is created which is subsequently used to minimize the test suite T. We explained each step in a separate subsection.

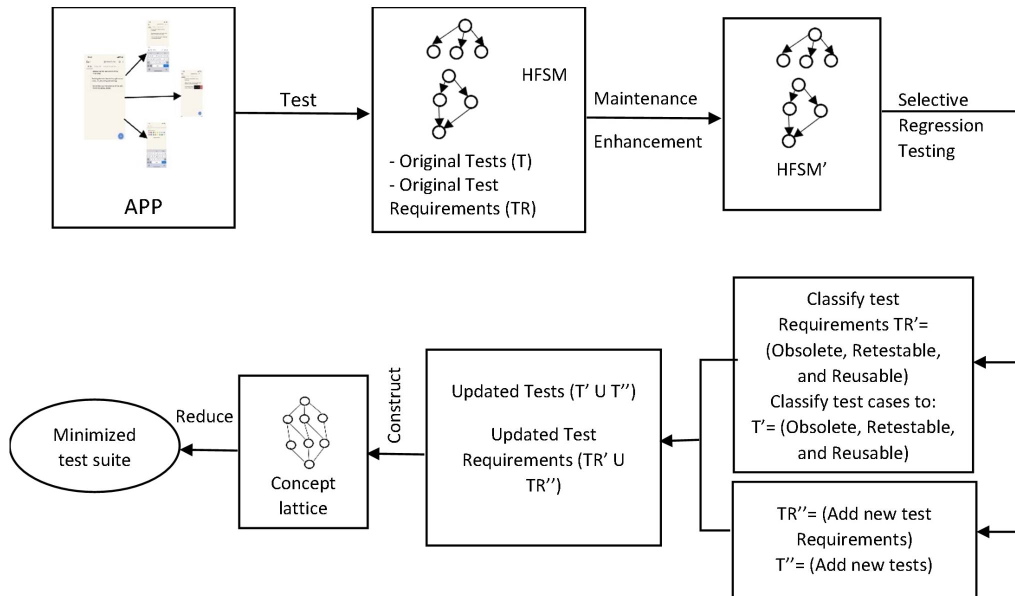


Figure (3.12) Test case minimization using concept analysis

3.2.1 Determine requirements and test cases based on selective regression testing for FSMApp

As using that the HFSM for FSMApp has been built according to the process described in Subsection (2.1.2), changes in the App lead to changes in the model. This can lead to these situations:

- (a) Test paths become obsolete and are removed.
- (b) Test paths still work, some are affected by changes, and some are not.
- (c) New paths need to be generated.

In Section 3.1 we defined a selective regression testing approach that classifies test paths as obsolete, retestable (affected by the change), and reusable (not affected by changes). Specifically, we need to update test requirements based on the type of changes as follows:

- Obsolete requirements: Node deletion affects test requirements related to the deleted node, rendering them obsolete. Node modifications change the type of node from LAP to a cluster node or from a cluster node to LAP. This type of modification affects the test requirements as the node needs to be replaced by another node or a sequence of nodes which means changing the test requirements or adding new requirements. Thus, node deletions and node modifications render test requirements that tour these nodes obsolete.
  Similarly, edge deletion renders any test requirements that related to the edge obsolete. Edge modification involves modification of the input constraint associated with it. This could mean adding or removing inputs, or modifying the type of input or action. Any of these affect test requirements and make them obsolete which is related to the modified edge.

- Retestable requirements: Retestable test requirements are those that still need them after modifying the application, and they exist in portions of the application that may be affected by the change. These are test requirements that visit portions of the FSMApp model that are close to the changes. So, any node n that is one edge away from a modified or deleted node must be retested Except for the source and sink nodes of the AFSM. When edges are changed, the set of retestable edges depends on the type of change (deleted or modified), the starting and ending nodes of the changed edges are potentially affected and hence non-obsolete test requirements that visit these nodes are retestable. Similarly, when edges (and possibly nodes) are added, existing nodes at which these new edges start, or end is considered potentially affected by the change and hence non-obsolete test requirements that visit these nodes are retestable (Except for the source and sink nodes of the AFSM).

- Reusable test requirements: Reusable test requirements are those that are neither obsolete nor retestable. When new edges (and nodes) are added that means new test requirements are added. Also, when the type of a node changes from a logical app page (LAP) to a cluster, that means we add new test requirements.

Therefore, we need to look to affected paths to update the test requirements (remove obsolete requirements, add requirements, and modify requirements). Non-path-affecting change leads to the same requirements.

Test Coverage Criteria

In our work, FSMApp generates a test suite T that satisfies test requirements based on graph coverage criteria and test aggregation criteria for its hierarchical model. Together, these constitute a set of test requirements TR.

Test coverage criteria define test requirements TR, they define which parts of a FSM model should be tested according to the used criterion. Then we generate test paths that meet all the test requirements. A test path is a sequence of transitions through the application FSM and through each lower-level FSM. FSMApp's test generation method first generates paths through each FSM based on some graph coverage criterion such as edge coverage which we used in our work. These paths are then aggregated based on an aggregation criterion for each FSM's paths, such as all combinations or each path at least once which we used. All combinations criterion is not applicable in our cases because it requires that every single path should be covered, then we have less opportunity to minimize.

Aggregation criteria may limit the amount of minimization. Thus, the aggregation criteria that we select have to be preserved for any minimization that we will do. The stronger coverage criteria mean less minimization is possible. So, here

in our work, we assume that the path aggregation criterion is to use each path at least once, which requires that aggregate paths from the start of the application to the termination of the application are formed such that every path generated for a lower-level FSM is selected at least once.

If we have edge coverage in each finite state machine FSM and if we do the aggregation requirements, we automatically have the edge coverage preserved. The following example shows that if we do the aggregation requirements, we automatically have the edge coverage preserved.

Let test path T= n1, n2, c1, n3, n4, and let cluster C1 as:

C1= n3, n4, n3, n5, n3 and it satisfies edge coverage.

Let's say that the edge e1= (n1, n2) is covered by T, and the edges e2= (n3, n4) and e3= (n3, n5) are needed to check if they are covered, then the original path T does preserve edge coverage.

Since the aggregation roles say that we have to substitute all paths for cluster C1. So, we will substitute C1 to the test path T as follows:

T= n1, n2, n3, n4, n3, n5, n3, n4

Therefore, the edges e2= (n3, n4), and e3= (n3, n5) are covered by the test path T, so it preserves the edge coverage.

At the end of this step, we have a test suite that meets all test requirements but may include redundant tests. We now apply test case minimization to this set of tests and test requirements.

3.2.2 Apply Concept Analysis

Concept Analysis is a hierarchical clustering technique [102, 22], for objects with discrete attributes. In general, the input of concept analysis is a set of objects,

a set of attributes, and a relation named the *context* that relates objects to their attributes. Let O be a set of objects. $O = \{o_1, o_2, \ldots, o_n\}$

Let A_i be the attributes of o_i. Then $\cup_{i=1}^{n} A_i = A$, the set of all attributes.

Concept analysis creates a relation R between subsets of O and A.

Let c be a concept where c = (X, Y), $X \subseteq O, Y \subseteq A$ and

Let $C = \{c_1, c_2, \ldots, c_k\}$ be the concepts.

Let $c_1 = (x_1, y_1)$, $c_2 = (x_2, y_2)$ $c1, c2 \in C$. Then $c_1 \leq_R c_2$ iff $X_1 \subseteq X_2$ or $Y_1 \supseteq Y_2$

Then (C, $\leq$) = R is a partial order

A relation R is called a partial ordering if it is reflexive, antisymmetric, and transitive. That is mean, for all $a, b, c \in P$ (partial ordering set), it must satisfy [93]:

1. Reflexivity: $a \leq a$, i.e. every element is related to itself.

2. Antisymmetric: if $a \leq b$ and $b \leq a$ then a = b, i.e. no two distinct elements precede each other.

3. Transitivity: if $a \leq b$ and $b \leq c$ then $a \leq c$.

A concept is an ordered pair (X, Y) where $X \subseteq O$ is a set of objects and $Y \subseteq A$ is a set of attributes satisfying the property that X is the maximal set of objects that are related to all the attributes in Y and Y is the maximal set of attributes that are related to all the objects in X.

if $C_1 = (X, Y_1) \wedge C_2 = (X, Y_2)$ $\wedge Y_1 \subseteq Y_2$ then $Y_1 = Y_2$

The concepts as defined above induce a complete lattice on the concepts, called the concept lattice (all subsets of a poset have a join and meet). The supremum contains all objects and no attributes $C_{Top} = \{O, \emptyset\}$ while the infimum has all attributes and no objects $C_{Bottom} = \{\emptyset, A\}$.

Let C is the set of concepts in a context (O, A, R) [22], [99]. Formally, a lattice is a partially ordered set in which every pair of elements has a unique supremum, also called a least upper bound or join. And a unique infimum, also called a greatest lower bound or meet. In other words, it is a structure with two binary operations: (Join, Meet). For each $(X_1, Y_1), (X_2, Y_2) \in (O, A, R)$, we have $(X_1, Y_1) \leq (X_2, Y_2) if X_1 \subseteq X_2 (or, equivalently, Y_2 \subseteq Y_1)$. The meet $\wedge$ and join $\vee$ operators are defined by:

$(X_1, Y_1) \wedge (X_2, Y_2) = (X_1 \cap X_2, Y_1 \vee Y_2)$

$(X_1, Y_1) \vee (X_2, Y_2) = (X_1 \vee X_2, Y_1 \cap Y_2)$

Where $Y1 \vee Y2$ (respectively, $X1 \vee X2$) is the set of elements of A (respectively, O) in relation with all the elements of $X1 \cap X2$ (respectively, $Y1 \cap Y2$). The set system $\{X \subseteq O$ (respectively, $Y \subseteq A$) such that (X, Y) is a concept of (O, A, R)$\}$ [79].

In our work, the concept analysis considers the test cases as objects O and the requirements as attributes A.

$T = \{t_1, t_2, ..., t_n\}$

$TR = \{r_1, r_2, ..., r_m\}$

Let T = O, TR = A

Then $A_i = \{r | r \in TR : t_i \in T \wedge t_i \text{ meet } r\}$

For every object (test) $t \in T$, there is a unique smallest concept in which it appears. This concept c, the smallest concept for t, is labeled with t. Analogously, for every attribute (test requirement) $r \in TR$, there is a unique largest concept in which it appears. This concept c' , the largest concept for r, is labeled with r.

Based on this labeling of concepts, we can use both object implication and attribute implication to reduce the context table. An object implication $t_i \Rightarrow t_j$ is found when the concept labeled with t_i appears lower in the lattice than the concept labeled with t_j. This implies that the row for t_j can be safely removed because its attributes (test requirements) are covered by t_i [109].

Object Implication: Given two tests $t_i, t_j \in T, t_i \Rightarrow t_j$ if and only if $\forall r \in TR$, (t_j R r) $\Rightarrow$ (t_i R r).

Object reduction rule: For two tests t_i, t_j, if $t_i \Rightarrow t_j$ then all test requirements covered by t_j are also covered by t_i. So, the row corresponding to the test t_j can be removed from the context table.

Attribute Implication: Given two test requirements $r_i, r_j \in TR, r_i \Rightarrow r_j$ if and only if $\forall t \in T, (t \text{ R } r_i) \Rightarrow (t \text{ R } r_j)$.

An attribute implication $r_i \Rightarrow r_j$ is found when the concept labeled with r_i appears lower in the lattice than the concept labeled with r_j. Based on this implication, the column r_j can be safely removed from the context table because when r_i is covered, it implies that r_j is also covered [109]. We call this the attribute reduction rule.

The process of reducing the context table continues until no further implications are found. The minimized test suite is then selected from the final reduced context table.

Algorithm 7 shows minimized test suite:

Input: ContextTable (test cases and test requirements)
Result: minimized test suite Tminum
Procedure MinumSet(ContextTable)
1: Tminum= empty, imp=false
2: **while** (not(imp) and ($ContextTable \neq empty$)) **do**
3: imp= true
4: **for** each two objects $t_i, t_j \in T$ **do**
5: **if** $(\forall r \in TR, (t_j Rr) \Rightarrow (t_i Rr))$ **then**
6: $t_i \Rightarrow t_j$ (Remove row for test case t_j from ContextTable)
7: imp =false
8: **end if**
9: **end for**
10: **for** each two attributes $r_i, r_j \in TR$ **do**
11: **if** $(\forall t \in T, (tRr_i) \Rightarrow (tRr_j))$ **then**
12: $r_i \Rightarrow r_j$ (Remove column for test requirement r_j from ContextTable)
13: imp =false
14: **end if**
15: **end for**
16: **end while**
17: **while** ($ContextTable \neq empty$) **do**
18: Remove row for test case t from ContextTable and add t to Tminum
19: $T_{minum} = T_{minum} \cup \{t\}$
20: **end while**
21: Return T_{minum}
end Procedure

Algorithm 7: Minimized Test Suite

3.2.3 Example to Illustrate Test Case Minimization of FSMApp using Concept Analysis

We will use To Do App to illustrate our approach to minimize test cases of Mobile Apps. To Do is a simple App to do lists of tasks. The app has these services: add tasks, modify task (delete task, edit task, mark for a task done or undo), and search for a task.

Determine requirements and test cases based on selective regression testing for FSMApp

We used the example To Do App that we used to illustrate selective regression testing of Mobile Apps [2] Section 3.1 . We assume that enhance To Do App by adding new feature to the app, it's record task. In this example here, we minimize the selected test cases by using concept analysis. Figure 3.13 shows the top level of the To Do App with the three main clusters (Task, Modify Task, Search) which show the main services for To Do app, while Figures 3.14 and 3.15 show the details for the other clusters of the To Do App.

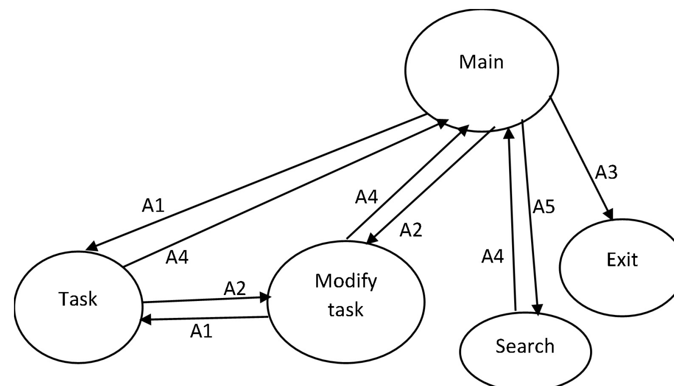
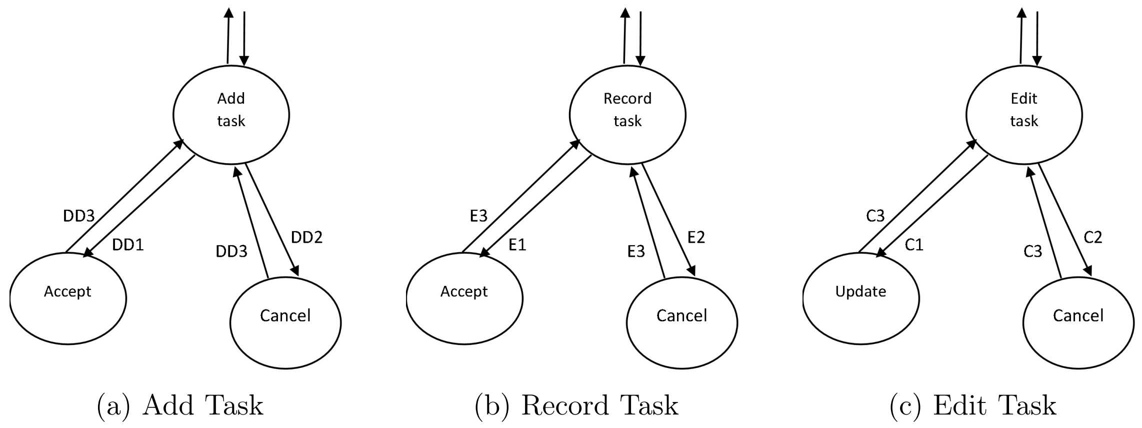


Figure (3.13) Main page of ToDo App

(a) Task Cluster (b) Modify Task Cluster

Figure (3.14) Sub-clusters of the To Do App



(a) Add Task (b) Record Task (c) Edit Task

Figure (3.15) LAPs of the To Do App

After we modified this App and applied the classification rules [2] in Section 3.1 to determine affected paths and non-affected paths we obtained the test cases in Table 3.18 (retestable and new test cases) for the modified To Do App. Details are given in Subsection 3.1.2. We then used the concept lattice approach in an attempt to minimize them. Table 3.19 shows the updated test requirements of the To Do App.

Table (3.18) The abstract test paths of the To Do app

ID	Test Sequence	Length
1	[Main, Task, Add Task, Accept, Add Task, Cancel, Add Task, Task, Main, Exit]	10
2	[Main, Task, Record Task, Accept, Record Task, Cancel, Record Task, Task, Main, Exit]	10
3	[Main, Task, Modify Task, Edit, Update, Edit, Cancel, Edit, Modify Task, Task, Main, Exit]	12
4	[Main, Task, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Task, Cancel, Modify Task, Main, Exit]	13
5	[Main, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Cancel, Modify Task, Main, Exit]	10
6	[Main, Modify Task, Edit, Update, Edit, Cancel, Edit, Modify Task, Main, Exit]	10

The test suite $T' = \{t_1, t_2, t_3, t_4, t_5, t_6\}$, and the set of requirements $TR' = \{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}, r_{11}, r_{12}, r_{13}, r_{14}, r_{15}\}$. The coverage information for each test case is shown by an X in the corresponding column in Table 3.20. Aggregation Requirements:

$R_1 = \{r_1, r_3, r_4, r_{10}, r_{11}\}$, $R_2 = \{r_1, r_3, r_5, r_{12}, r_{13}\}$

$R_3 = \{r_1, r_2, r_3, r_7, r_{14}, r_{15}\}$

$R_4 = \{r_1, r_2, r_3, r_6, r_8, r_9\}$

$R_5 = \{r_2, r_3, r_6, r_8, r_9\}$

$R_6 = \{r_2, r_3, r_7, r_{14}, r_{15}\}$

Table (3.19) Test Requirements of To Do App

Requirements	Edges	Description
r1	A1	Access new task (add or record information of tasks)
r2	A2	Access modify task
r3	A3	Exit the system
r4	D1	Access add task
r5	D2	Access record task
r6	B1	Mark a task done/undo
r7	B2	Access Edit Task
r8	B3	Delete the task
r9	B4	Cancel to Previous Page
r10	DD1	Add the new task
r11	DD2	Cancel to Previous Page
r12	E1	Record the new task
r13	E2	Cancel to Previous Page
r14	C1	Update the task
r15	C2	Cancel to Previous Page

From the context table3.20, we constructed the concept lattice. First, we built a table of the concepts which were objects (test cases of To Do App) and attributes (test requirements of To Do App), then we constructed the concept lattice. For example, a concept C_1 is an ordered pair $(\{t_1\}, \{r_1, r_3, r_4, r_{10}, r_{11}\})$, $X = \{t_1\}$, $Y = \{r_1, r_3, r_4, r_{10}, r_{11}\}$. Table 3.21 shows the concepts of the To Do App. The Top shows all test cases for the To Do App and no test requirements and the Bottom shows all test requirements for the To Do App and no test cases. Figure 3.16 shows Concept Lattice.

Table (3.20) Context Table

	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}	r_{13}	r_{14}	r_{15}
t_1	x		x	x						x	x				
t_2	x		x		x							x	x		
t_3	x	x	x				x							x	x
t_4	x	x	x			x		x	x						
t_5		x	x			x		x	x						
t_6		x	x				x							x	x

Table (3.21) The Table of Concepts

Concept	(objects, attributes)
Top	$(\{t_1, t_2, t_3, t_4, t_5, t_6\}, \{\})$
C_1	$(\{t_1\}, \{r_1, r_3, r_4, r_{10}, r_{11}\})$
C_2	$(\{t_2\}, \{r_1, r_3, r_5, r_{12}, r_{13}\})$
C_3	$(\{t_3\}, \{r_1, r_2, r_3, r_7, r_{14}, r_{15}\})$
C_4	$(\{t_4\}, \{r_1, r_2, r_3, r_6, r_8, r_9\})$
C_5	$(\{t_1, t_2, t_3, t_4\}, \{r_1\})$
C_6	$(\{t_3, t_4, t_5, t_6\}, \{r_2\})$
C_7	$(\{t_1, t_2, t_3, t_4, t_5, t_6\}, \{r_3\})$
C_8	$(\{t_4, t_5\}, \{r_6\})$
C_9	$(\{t_3, t_6\}, \{r_7\})$
C_{10}	$(\{t_4, t_5\}, \{r_8\})$
C_{11}	$(\{t_4, t_5\}, \{r_9\})$
C_{12}	$(\{t_3, t_6\}, \{r_{14}\})$
C_{13}	$(\{t_3, t_6\}, \{r_{15}\})$
BOT	$(\{\}, \{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, \ldots, r_{15}\})$

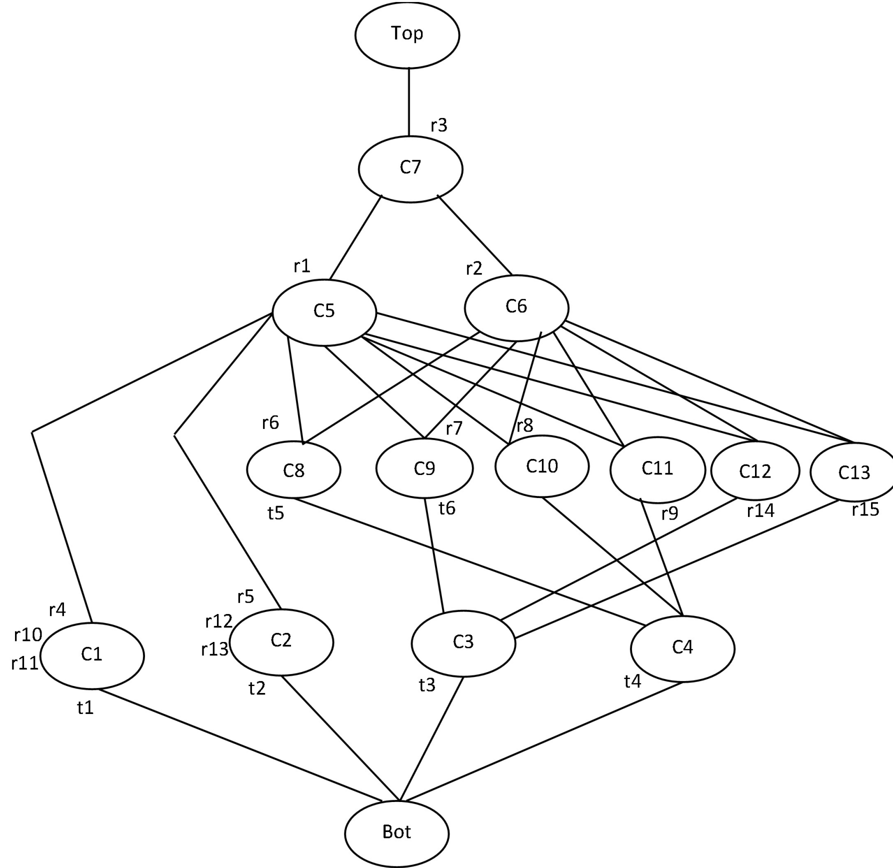


Figure (3.16) Concept Lattice

We then needed to reduce the size of the context table by applying object reductions and attribute reductions. Based on the labeling of concepts, we can use both object implication and attribute implication to reduce the context table. An object implication $t_i \Rightarrow t_j$ is found when the concept labeled with t_i appeared lower in the lattice than the concept labeled with t_j. This implies that the row for t_j can be safely removed because its attributes are covered by t_i.

Based on object implications, we found that $t_4 \Rightarrow t_5$ because it appears lower in the lattice. Similarly, $t_3 \Rightarrow t_6$. Thus, both t_5 and t_6 can be safely removed from the table.

An attribute implication $r_i \Rightarrow r_j$ is found when the concept labeled with r_i appears lower in the lattice than the concept labeled with r_j. Based on this implication, column r_j can be safely removed from the context table because when r_i is covered, it implies that r_j is also covered. For attribute implications, we found that $r_1 \Rightarrow r_3$ and $r_2 \Rightarrow r_3$, hence we can safely remove test requirement r_3. Table 3.22 shows a reduced context table. Table 3.23 shows the reduced concepts. Figure 3.17 shows a concept lattice based on the reduced concept table.

Table (3.22) Context Table

	r_1	r_2	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}	r_{13}	r_{14}	r_{15}
t_1	x		x						x	x				
t_2	x			x							x	x		
t_3	x	x				x							x	x
t_4	x	x			x		x	x						

Table (3.23) Reduced Concepts

Concept	(objects, attributes)
Top	$(\{t_1, t_2, t_3, t_4\}, \{\})$
C_1	$(\{t_1\}, \{r_1, r_4, r_{10}, r_{11}\})$
C_2	$(\{t_2\}, \{r_1, r_5, r_{12}, r_{13}\})$
C_3	$(\{t_3\}, \{r_1, r_2, r_7, r_{14}, r_{15}\})$
C_4	$(\{t_4\}, \{r_1, r_2, r_6, r_8, r_9\})$
C_5	$(\{t_1, t_2, t_3, t_4\}, \{r_1\})$
C_6	$(\{t_3, t_4\}, \{r_2\})$
BOT	$(\{\}, \{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, \ldots, r_{15}\})$

It is possible that after a reduction step, further reductions are possible. Specifically, for attribute implications, we found that $r_4 \Rightarrow r_1$ and $r_{14} \Rightarrow r_2$. We can safely remove the test requirements r_1 and r_2. Table 3.24 shows a reduced context table. Table 3.25 shows the reduced concepts. Figure 3.18 shows the corresponding concept lattice.

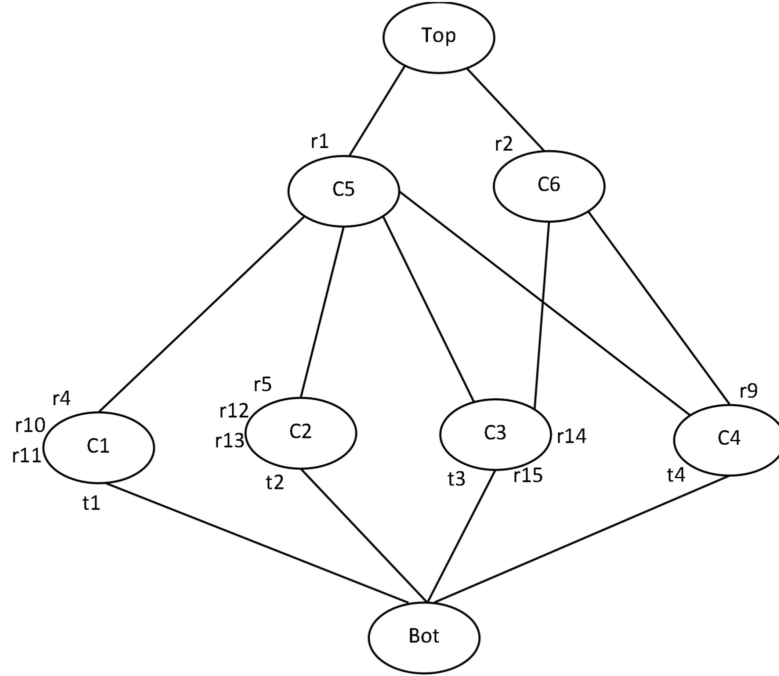


Figure (3.17) Concept Lattice: First Minimization

Table (3.24) Reduced Context Table

	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}	r_{13}	r_{14}	r_{15}
t_1	x						x	x				
t_2		x							x	x		
t_3				x							x	x
t_4			x		x	x						

Table (3.25) Reduced Concepts

Concept	(objects, attributes)
Top	$(\{t_1, t_2, t_3, t_4\}, \{\})$
C_1	$(\{t_1\}, \{r_4, r_{10}, r_{11}\})$
C_2	$(\{t_2\}, \{r_5, r_{12}, r_{13}\})$
C_3	$(\{t_3\}, \{r_7, r_{14}, r_{15}\})$
C_4	$(\{t_4\}, \{r_6, r_8, r_9\})$
BOT	$(\{\}, \{r_4, r_5, r_6, r_7, r_8, \ldots, r_{15}\})$

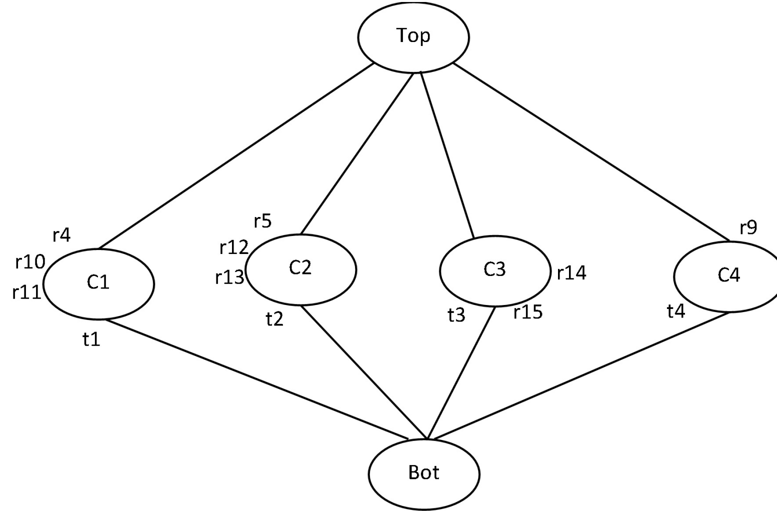


Figure (3.18) Concept Lattice: Second Minimization

Since there were no new attribute reductions or object reductions found, the minimized test suite $T = \{t_1, t_2, t_3, t_4\}$.

In our approach (test case minimization of FSMApp for Mobile Apps) here, we did not guarantee aggregation test requirements. We plan in future work to enhance our approach to guarantee aggregation test requirements.

Next, we need to prioritize these test cases. We will explain it in the next Section 3.3.

3.3 Test Case Prioritization

Since FSMApp has not addressed test case prioritization yet. In this section, we presented a model-based approach to prioritize test cases for Mobile Apps. Due to the resource and time constraints for regression testing, we may need to detect faults in the modified Mobile App as early as possible. Test case prioritization seeks to order test cases in such a way that early fault detection is maximized.
The test case prioritization problem is as follows [97]:

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 $(\forall T'')$ $(T'' \in PT)$ $(T'' \neq T')$ $[f(T') \geq f(T'')]$. PT represents the set of all possible orderings of T, and f is a function that, when applied utilized to any such ordering, produces an award value for that ordering.

3.3.1 Test Case Prioritization Process

The idea of our model-based test prioritization approach is to use the original model and the modified model to identify a difference between these models. The collected information is then used to prioritize the test suite. We suggested prioritizing tests based on input complexity since more inputs might be associated with a more complex functionality which in turn would make it more fault-prone. Figure 3.19 shows test case Prioritization of FSMApp Process.

We prioritized the test suite based on the number of inputs (complexity) for each test path. Identifying which test path that has the biggest number of inputs (values and actions) and giving it a high priority. Then ordering the test paths in descending order, so the test path (with more complexity of inputs) that has the highest priority will execute first.

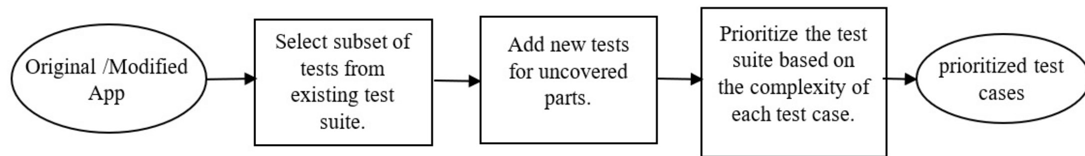


Figure (3.19) Test Case Prioritization of FSMApp Process

The steps of the process are as follows:

Step 1: Select a subset of tests from the existing test suite.

- Builds a hierarchical model HFSM' for the modified Mobile App.
  - Partition the Mobile App into clusters.
  - Define logical App pages (LAP) and input constraints.
  - Build FSM for the clusters.
- Identify a difference between the original model HFSM and the modified model HFSM′.
- Select tests related to the modified parts of the model from the existing test suite.

Step 2: Add new tests for uncovered parts.

- Generate new abstract test cases for uncovered parts.

Step 3: Prioritize the test suite based on the complexity of each test case.

- Determent complexity of each test case.
- Assign priority to each test case based on its complexity. Assign high priority to test path that has the biggest number of inputs (values and actions).
- Ordering the test cases in descending order (high priority to low priority).

3.3.2 Example Used to Illustrate Approach

We used To Do App example to illustrate our approach of FSMApp test case prioritization. To Do is a simple app to make lists of tasks. The app has these

services: add tasks, modify tasks (delete tasks, edit tasks, mark for a task done or undo), and search for a task. For more details about the example see Sections 3.1, 3,2.

Figure 3.20 shows the original model HFSM for the ToDo App. Table 3.26 shows the original test suite for the ToDo App.

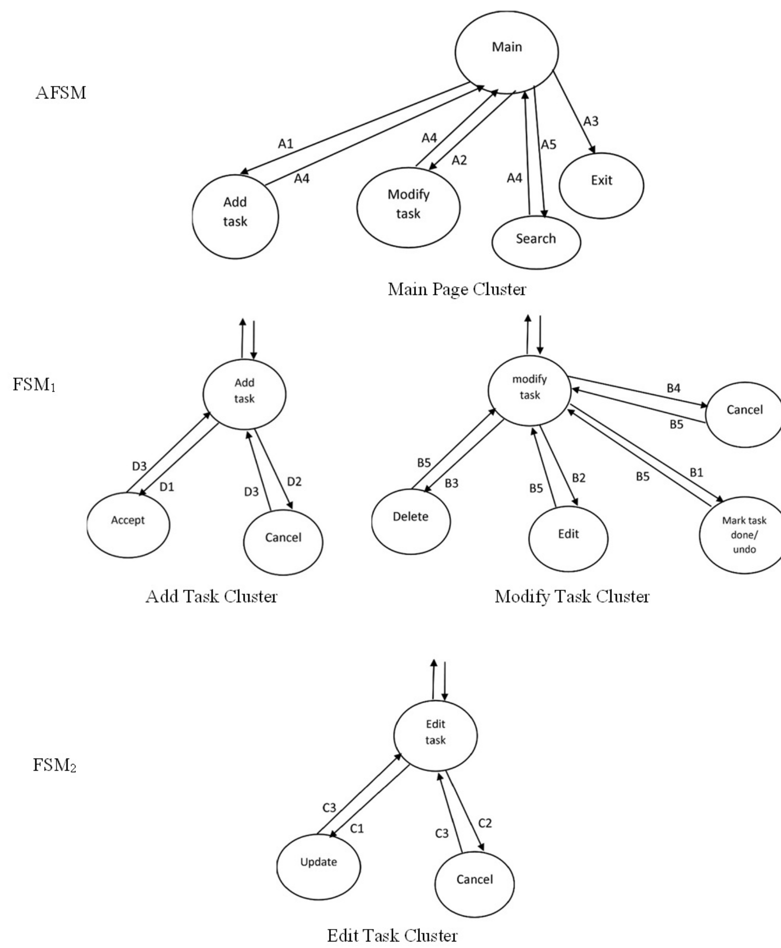


Figure (3.20) Original Model HFSM for ToDo App

Table (3.26) The original Test Paths of the To Do App

ID	Test Sequence	Length
1	[Main, Add Task, *Accept*, Add Task, *Cancel*, Add Task, Main, *Exit*]	8
2	[Main, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Main, *Exit*]	8
3	[Main, Modify Task, Edit, *Update*, Edit, *Cancel*, Edit, Modify Task, Main, *Exit*]	10
4	[Main, Search, Main, *Exit*]	4

Step1: Select a subset of tests from the existing test suite.

- Builds a hierarchical model HFSM' for the modified Mobile App.

  We used FSMApp approach [8] to build the hierarchical model HFSM' for the modified ToDo App. We Partitioned the ToDo Mobile App into clusters, defined the logical App pages (LAP) and input constraints for it, then built FSM for the clusters. The constraints on inputs for ToDo App are:

  - Required (R): required input must be entered.
  - Required Value (R(parm)): one must enter at least one value.
  - Optional (O): an input may or may not be entered.
  - Single Choice (C1): one input should be selected from a set of choices.
  - Multiple choices (Cn) are possible.

We explained the input constraints and input types for ToDo App in detail in Section 3.1. Table 3.27 shows input types for the ToDo App. Figure 3.21 shows the modified hierarchical model for ToDo Mobile App.

Table (3.27) The Input Types of the To Do App

Input Type	**The Action**	**The Input Constraints**
A button	click	R($< Click >$), C1(Select Button, Click)
Pickers	drop-down and click	C1($< Dropdown >, < Click >$)
Lists	sort	R($< Sort >$)
A card	click, swipe, scroll, and pick-up-and-move	C1($< Click >, < Swipe >, < Scroll >, < Pick >$)

- Identify a difference between the original model HFSM and the modified model HFSM'.

  Here we used the original model HFSM for ToDoApp in figure 3.20 and the modified model HFSM' in figure 3.21 to identify a difference between these models.

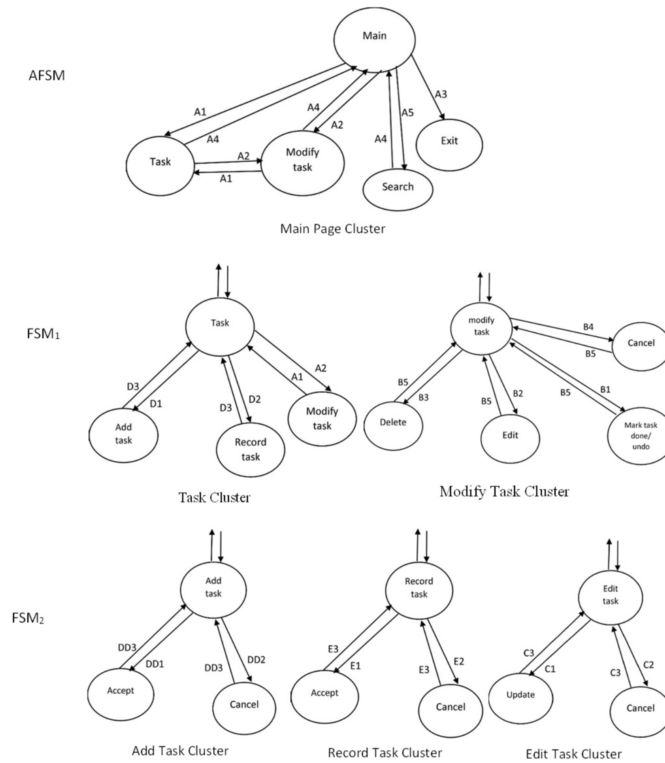


Figure (3.21) Modified Model $HFSM'$ for ToDo App

We identified the following modifications from the original model HFSM and modified model HFSM′:

Modification 1: Node deletion, edge/node additions

Add Task node is deleted. Then a new cluster node is added, which gave options to add new tasks by recording information about the task and by writing the information about the task. Also, modify tasks by editing information, deleting tasks, and marking done or undo of the task. Delete Add Task node, adding the new cluster node (Task), and adding a new FSM for the Task cluster caused the abstract test cases associated with test path 1 in Table 3.26 affected paths.

Modification 2: Edges additions

Adding edges between cluster node Task and cluster node Modify Task. These edges allow modifying tasks without returning to the main page of the App. This modification makes test paths 2 and 3 in Table 3.26 affected paths (PC).

- Select tests related to the modified part of the model.

  Here we used the collected information above to select tests from the existing test suite in Table 3.26 that related to modified parts of the App which are test cases 1, 2, and 3. But test case 1 is invalid.

Step2: Add new tests for uncovered parts

In this step, We determined if any parts of the system have not been tested and generated a new set of tests. We used a partial generation to generate the test cases because the changes in the model are localized, details about generated the new tests in Section 3.1. Since the test case1 in Table 3.10 is not valid anymore, we need new tests to cover untested parts.

The following four new abstract test cases covered the paths of the model that are not covered by selected test cases.

1. [Main, Task, Add Task, *Accept*, Add Task, *Cancel*, Add Task, Task, Main, *Exit*]
2. [Main, Task, Record Task, *Accept*, Record Task, *Cancel*, Record Task, Task, Main, *Exit*]
3. [Main, Task, Modify Task, Edit, *Update*, Edit, *Cancel*, Edit, Modify Task, Task, Main, *Exit*]
4. [Main, Task, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Task, Main, *Exit*]

Table 3.28 shows the abstract test cases (selected set of tests, new tests) that we need to test the modified version of the To Do App.

Table (3.28) The selected and new test paths of the To Do app

ID	**Test Sequence**	**Length**
1	[Main, Task, Add Task, *Accept*, Add Task, *Cancel*, Add Task, Task, Main, *Exit*]	10
2	[Main, Task, Record Task, *Accept*, Record Task, *Cancel*, Record Task, Task, Main, *Exit*]	10
3	[Main, Task, Modify Task, Edit, *Update*, Edit, *Cancel*, Edit, Modify Task, Task, Main, *Exit*]	12
4	[Main, Task, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Task, Cancel, Modify Task, Main, Exit]	13
5	[Main, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Main, *Exit*]	8
6	[Main, Modify Task, Edit, *Update*, Edit, *Cancel*, Edit, Modify Task, Main, *Exit*]	10

Step 3: Prioritize the test suite based on the complexity of each test case.

In this step, we will prioritize test paths in Table 3.28 based on the complexity of each test path.

- Determent complexity of each test case.

  We will determine the complexity of each test path by determining the number of input values and actions. For example, Test Path1 has six inputs and five actions. The inputs are Task name (parname), Task Date and Time (ParD), (ParT) which occurs twice in test path 1. The actions are click Add New Task button (b-ANT), Click Accept button(b-ANTA), Click Cancel (b-Cancel), and click back arrow to exit the Mobile App (buttonBack). The complexity for this test path is (6+5=11).

$$C = \sum_{i=1}^{n} InpValue_i + \sum_{j=1}^{m} InpAction_j$$

  Where,

  C: is the total complexity for the test path.

  InpValue: is the input value.

  InpAction: is the actions.

  Tables 3.29 to 3.32 show the complexity for each test path. Column 1 shows the transaction of the test path, column 2 is the Constraint of the input, column 3 is the number of the input values, column 4 is the number of actions, and column 5 is the complexity of each transaction for the test path.

Table (3.29) The Complexity of the Test Path 1

Edge	**Constraint**	**Input Value**	**Actions**	**Complexity**
A1	R(b-Task)	0	1	1
D1	C1(SelectAddInfo, SelectRecodInfo)	0	1	1
DD1	R(Par(name,D,T), b-ANTA) S(Par(name, D,T), b-ANTA)	3	1	4
DD2	O(Par(name, D, T), R(b-Cancel) S(Par(name, D, T) b- Cancel)	3	1	4
A3	R(buttonBack)	0	1	1
Total Complexity				11

Table (3.30) The Complexity of the Test Path 2

Edge	**Constraint**	**Input Value**	**Actions**	**Complexity**
A1	R(b-Task)	0	1	1
D2	C1(SelectAddInfo, SelectRecodInfo)	0	1	1
E1	R((RecordInfo), b-AR)	1	1	2
E2	O(RecordInfo), R(b-Cancel)	1	1	2
A3	R(buttonBack)	0	1	1
Total Complexity				7

Table (3.31) The Complexity of the Test Path 3

Edge	**Constraint**	**Input Value**	**Actions**	**Complexity**
A1	R(b-Task)	0	1	1
A2	R(Swiping the task from right to left)	0	1	1
B2	R((Par(name,D,T)), b-Update)	3	1	4
B4	O(Par(name,D,T)), R(b-Cancel)	3	1	4
A3	R(buttonBack)	0	1	1
Total Complexity				11

Table (3.32) The Complexity of the Test Path 4

Edge	**Constraint**	**Input Value**	**Actions**	**Complexity**
A1	R(b-Task)	0	1	1
A2	R(Swiping the task from right to left)	0	1	1
B1	R(double tap the task)	0	1	1
B3	R(selectTask), R(b-Delete)	0	2	2
A3	R(buttonBack)	0	1	1
Total Complexity				6

Table (3.33) The Complexity of the Test Path 5

Edge	**Constraint**	**Input Value**	**Actions**	**Complexity**
A2	R(Swiping the task from right to left)	0	1	1
B2	R((Par(name,D,T)), b-Update)	3	1	4
B4	O(Par(name,D,T)), R(b-Cancel)	3	1	4
A3	R(buttonBack)	0	1	1
Total Complexity				10

Table (3.34) The Complexity of the Test Path 6

Edge	Constraint	Input Value	Actions	Complexity
A2	R(Swiping the task from right to left)	0	1	1
B1	R(double tap the task)	0	1	1
B3	R(selectTask), R(b-Delete)	0	2	2
A3	R(buttonBack)	0	1	1
Total Complexity				5

- Assign priority to each test case based on its complexity. Assign high priority to test path that has the biggest number of inputs (values and actions).

  In this step, the test case that has high complexity is assigned with high priority. The priority starts from 1 to N. one means the highest priority. The test path that has priority (1) will execute first. If there are paths that have the same priority, tester randomly chooses which one executes first. Table 3.35 shows the priority for each test path.

Table (3.35) The priority of the Test Paths

No	**Complexity**	**Priority**
1	11	1
2	7	3
3	11	1
4	6	4
5	10	2
6	5	5

- Ordering the test cases in descending order (high priority to low priority). The test case that has high priority will be executed first.

  The prioritized test suite is:

$$T = \{t_1, t_3, t_5, t_2, t_4, t_6\}$$

.

3.3.3 Example to Combining Test Case Prioritization with Selective Regression Testing and Test Case Minimization

Above we applied our test case approach to a modified version of the Mobile App. Since our approach is a combination of regression testing, we also, used our test case prioritization to prioritize test cases that we obtained from applying test case minimization in Section 3.2. Table 3.36 shows the minimized test suite.

Table (3.36) The minimized Test Paths

No	Test Paths	Length
1	[Main, Task, Add Task, Accept, Add Task, Cancel, Add Task, Task, Main, Exit]	10
2	[Main, Task, Record Task, Accept, Record Task, Cancel, Record Task, Task, Main, Exit]	10
3	[Main, Task, Modify Task, Edit, Update, Edit, Cancel, Edit, Modify Task, Task, Main, Exit]	12
4	[Main, Task, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Task, Cancel, Modify Task, Main, Exit]	13

Here we applied step 3 which is the determent complexity of each test case, and we assigned priority to each test case based on its complexity. Table 3.37 shows the complexity and the priority for each test path.

Table (3.37) The priority of the Test Paths

No	Complexity	Priority
1	11	1
2	7	2
3	11	1
4	6	3

Next, we ordered the test cases in descending order (high priority to low priority). The test case that has high priority will be executed first. The prioritized test suite is: T = $\{t_1, t_3, t_2, t_4\}$.

Chapter 4

Guidelines for Combining Regression Testing Approaches

Regression testing is an important activity in software maintenance and software enhancement as it can help developers determine if changes made to the system are handled appropriately without compromising efficiency. While not employed often, it is possible to combine several regression testing techniques. Combining them can lead to a more efficient and effective regression test suite. The three types of regression testing, selective, minimization, and prioritization, can be combined in four different ways: There are three ways to combine two of the approaches and one way to combine all three. However, to efficiently and effectively employ regression testing, it is crucial to select the appropriate combination for the software that is to be tested. For example, the expected quality of the software, the kind of changes made, or the criticality of the software heavily influence which strategies to combine and in what order. So far, literature only addressed two types of combinations: combining test case prioritization and test case minimization [111, 106] (neither study [111, 106] did take selective regression testing into account and guidelines on

how and when to select a particular combination of regression testing approaches are not addressed [106]), and combining selective regression testing and test case prioritization [107, 103]. Both [107] and [103] did not consider test case minimization as part of their combination. They also did not address guidelines on how and when to select a particular combination of regression testing approaches, see Section 2 for more information. However, there is no systematic approach to combining all regression testing approaches. As this could provide a path to more efficient and effective testing, we aim to provide guidelines for combining regression testing approaches systematically.

Mayrhauser et al. [113] presented regression testing support for Sleuth, a test generation tool based on domain-based testing. They explained the rules for building regression tests for a variety of possible regression testing strategies from retesting all strategies to selective regression testing strategies. They described some situations that may call for different types of regression testing approaches. We used their idea in terms of combining regression testing approaches. We described some situations that may call for different combinations of regression testing approaches. So, we can choose which combination of regression testing approaches to use.

As software changes, regression testing is needed. After the maintenance process or the enhancement process of any software, software testers tend to use one of the three regression testing techniques to test the modified software: selective regression testing, test case minimization, or test case prioritization. However, we observed situations where combining these regression testing techniques could improve the overall regression testing process making it more effective and efficient. Theoretically, selective regression testing, test case minimization, and test case prioritization could be combined in four possible ways:

1. Selective regression testing, test case minimization, and test case prioritization.

2. Selective regression testing and test case prioritization.

3. Test case minimization and test case prioritization.

4. Selective regression testing and test case minimization.

We provided guidelines for combining regression testing approaches based on a systematic approach. We outlined all possible situations that can occur and showed how each of them influences which combination to use. Since the software maintenance process, the types of maintenance, software enhancements, coverage criteria, the impact of the changes on the model, time constraints, resource constraints, levels of risk failure, and confidence in the quality of the prior version of the software all are factors that influence which combination of regression testing approaches are used, the combination of these situations needs to be taken into account when selecting the combination of regression testing approaches.

4.1 Combination of Regression Testing Strategies

Several regression testing techniques can be used in combination for a more efficient and effective regression test suite. The challenge is, however, to appropriately combine the regression testing approaches as they each have different strategies and depend on the software being tested. The three types of regression testing, selective, minimization, and prioritization, can be combined using the following strategies:

1. *Apply a selective regression testing technique:* Identify the type of changes and then classify test cases into obsolete, re-testable, and reusable test cases. Add new test cases to cover untested parts. In this combination, the strong coverage criteria are avoided to make minimization possible.

2. *Apply a test case minimization technique:* In this step the updated test cases (re-testable, new test cases) and the updated test requirements from the selective regression testing are used to build a coverage information table. The test cases are reduced by removing redundant tests to get a minimized test suite that meets all the requirements.

3. *Apply a test case prioritization technique:* Prioritize the minimized test suite by assigning a priority to each test case based on some criteria such as the complexity of each test case. Execute the prioritized test cases based on their priority. A test case that has high priority is executed first.

There is a connection between these steps. The output of the first step, selective regression testing, is used as the input for the second step where the selected test cases based on coverage criteria are minimized. The output of the second step, the test case minimization, is used as input for the third step where the minimized test suite based on a specific matrix is prioritized.

How the combination of regression testing is applied depends on the software that is being tested. For example, the combination of strategies is influenced by the level of risk for software failure, the quality of the software, the coverage criteria, and the time and resources available. Figure 4.1 shows an overview of the process of combining regression testing approaches.

The strategy of combining selective regression testing and test minimization combination is as follows: First, apply a selective regression testing technique, then apply a test minimization technique to minimize the selected test suite. The strategy of combining selective regression testing and test prioritization is outlined as follows: First, apply a selective regression testing technique, then use a test prioritization technique to prioritize the selected test suite. The strategy of combining test

case minimization and test case prioritization is outlined as follows: First, apply a test minimization technique, then apply a test prioritization technique to prioritize the minimized test suite. Within these combinations, we can use the existing techniques for selective regression testing, test minimization, and test prioritization. We distinguish between four different combinations of regression testing approaches. Table 4.1 shows their corresponding strategies.

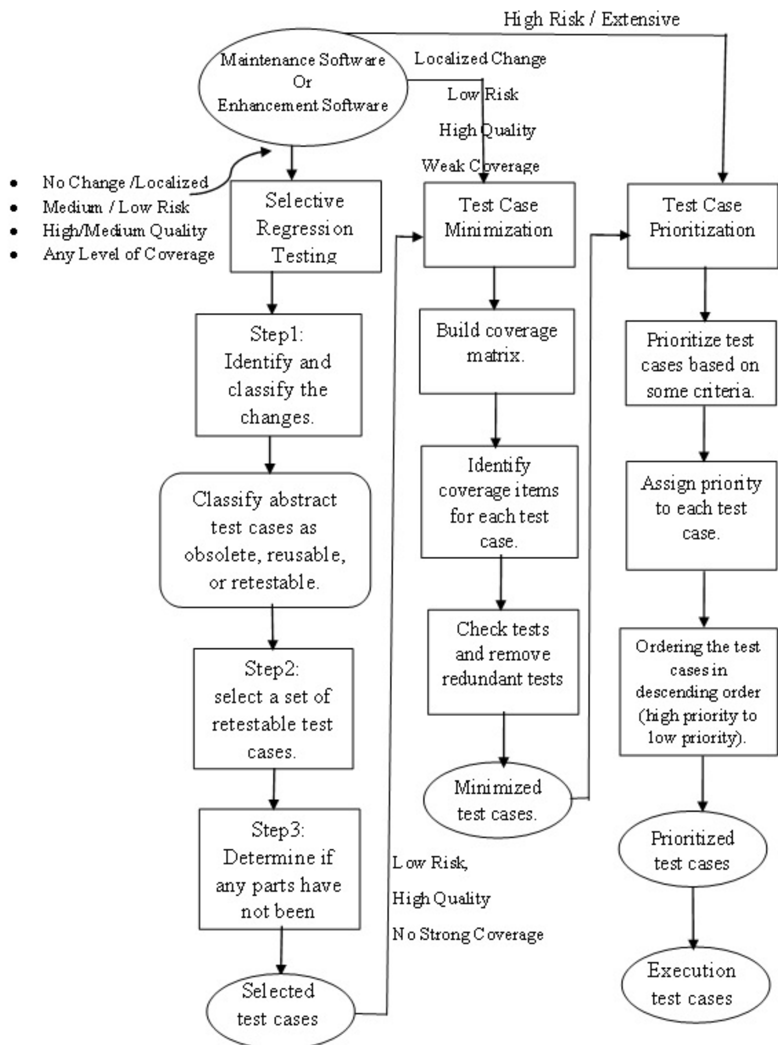


Figure (4.1) Process of Combining the Three Types of Regression Testing

Table (4.1) Strategies of Regression Testing Approaches

Combination	**Strategies to apply**
Selective regression testing, test case minimization, and test case prioritization	(1) Selective regression testing (2) Test case minimization (3) Test case prioritization
Selective regression testing and test case minimization	(1) Selective regression testing (2) Test case minimization
Selective regression testing and test case prioritization	(1) Selective regression testing (2) Test case prioritization
Test case minimization and test case prioritization	(1) Test case minimization (2) Test case prioritization

4.2 Situations for Combining Regression Testing Approaches

The software maintenance process, software enhancements, the types of maintenance, coverage criteria, the impact of the change on the model, time constraints, available resources, level of risk failure, and confidence in the quality of the prior version of the software are all factors that influence which combination of regression testing approaches are used. We described these situations that may call for different combinations of regression testing approaches in Table 4.2.

Table (4.2) Situations for Combining Regression Testing Approaches

	Situations				
Combination Regression Testing	**Coverage Criteria**	**Model Changes**	**Resources or Time Restrictions**	**Failure Risk**	**Software Quality**
Selective Minimization Prioritization	Medium Weak	Local	Some	Low	High
Selective Minimization	Medium Weak	No-Change Local	Few	Low	High
Selective Prioritization	Strong Medium Weak	Local	Some	Medium Low	Medium
Minimization Prioritization	Weak	Local	Few	Low	High

These factors can be used to help in making a decision about which combination is more efficient and effective for regression testing of a given system under test. Figure 4.2 shows a decision tree for the combinations of regression testing approaches. For example, if a system has undergone localized updates in which changes are isolated to a few partitions of the system, the coverage criteria that is used to generate test cases is not a strong coverage and there are time or resource restrictions like a tight deadline or limited resources. In this case may combine selective regression testing, test minimization, and test prioritization are better to use, particularly when its old version quality or the quality of its existing test suite

is high, and the level of risk failure is low. That means we do not need to look at these situations just as a single concern but also, as a combination of concerns. For example, when we look at the situation of a tight deadline, our thinking will tend to use reduction techniques (selective regression testing, test case minimization) to minimize test cases so we can save time. But at the same time, we need to look at other situations such as risk situations, if the system under regression test has high risk, we can not use reduction techniques.

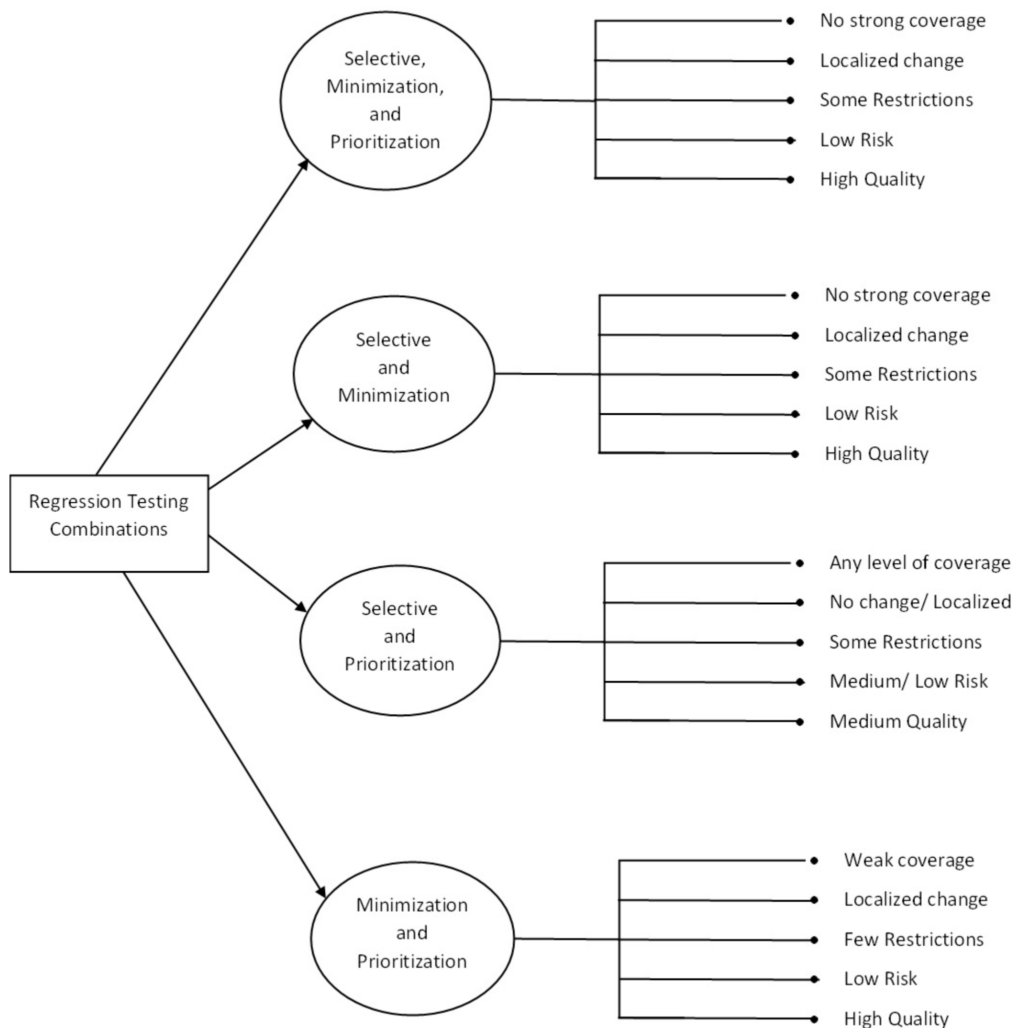


Figure (4.2) Decision Tree for Combinations of Regression Testing

1. Risk of the system under regression test: In a situation where a system under test has a high-risk failure, it is advised to avoid combinations that have a test case minimization. With a high-risk failure comes the risk of losing some of the fault-detection capability in the test suites when using minimization techniques. For example, if we test a safety-critical system, this could result in serious consequences when a fault is missed.

Selective regressing testing may have a similar risk if we are using it to test a system that has a high level of risk failure, such as a safety-critical system. We may use selective regression testing to test a system that has a medium level of risk such as a financial system.

The decision to regression test systems that have a high level of risk is designed to avoid the combinations of regression testing that have reduction techniques (selective regression testing or test minimization). The test case prioritization approach can be used for regression test systems that have a high level of risk.

2. Quality of Existing System: In the situation of the quality system, it is better to avoid combinations that have test case minimization when the quality system under test or the quality of its existing test suite is a concern (low quality) because we do not need to minimize a lot of test cases, so we may get good quality software which is reasonably bug or defects free.

Selective regression testing does not risk losing the quality so we can use selective regression testing to test not just systems that have high quality but also systems that have medium or low quality with using strong coverage criteria.

3. The Coverage Criteria: In the coverage criteria situation, when testers need to use strong coverage criteria to generate test cases, they would not use the combi-

nations that have test case minimization because strong coverage criteria will limit minimizing test cases.

The test coverage criteria [12] defined the test requirements, which then defined what parts of a model should be tested according to the used criterion. We then generate test paths that satisfy all the test requirements. There are different test requirement criteria for graph coverage that testers can use to produce test paths including: Edge-pair coverage, Prime path coverage, Node coverage, Edge coverage, Complete coverage, and Specified path coverage. There are also aggregation criteria that are used when we need to consider multiple partitions at the same time, which are used for aggregation test requirements such as: All combination coverage (strong coverage criteria), and at least one coverage (weak coverage criteria). There are no concerns about limiting the process of selective regression testing or test case prioritization when using any of the coverage criteria to generate test cases.

4. The Impact of the Changes: In situations where the model changes, if the impact of the changes is localized, we can use the combinations that have test case minimization because the changes are isolated to some partitions of the model so we can minimize test cases that are not related to the changed part and we do not need them anymore.

In addition to localized change, selective regression testing can be used if there are no changes in the model which means in case of corrective maintenance we fix defects. Here we need to identify the affected parts related to the defects and use selective regression testing in terms of retestable tests to test affected parts. Therefore, the combination of selective regression testing and test case prioritization can be used if there are no changes in the model.

If the impact of the changes is extensive and that required full regeneration, we can not use any of the combinations of regression testing but we can use the test case prioritization approach to prioritize test cases.

5. Time/Resources of Regression Testing: In situations of Time/Resources, deadlines, and testing resources may restrict using any of the combinations depending on the other factors. For example, if there is adequate available time to run more test cases and fault detection is critical, we need to avoid the combinations that have test case minimization. But if the testers have tight deadlines or limited resources, and the system under test is not safety critical (high risk), testers can use the combination of selective regression testing, test case minimization, and test case prioritization. In contrast, testers can not use the combinations that have reduction techniques (selective regression testing, test case minimization) in case of the system has high risk even if they have tight deadlines.

4.3 Example for Using The Guidelines to Combining Three Types of Regression Testing (Selective, Minimization, and Prioritization)

We used To Do Mobile App, which is a simple app to do lists of tasks. The app has these services: add tasks, modify tasks (delete tasks, edit tasks, mark for a task done or undo), and search for a task. We first applied FSMApp approach for testing Mobile Apps to get abstract test cases for the original version of the To Do App. Then We made some modifications to To Do App to apply a combination of

selective regression testing, test minimization, and test prioritization approach to the modified version of To Do App. For more details about the example see Chapter 3.

We combined all three types of regression testing (selective regression testing, test minimization, and test prioritization) as follows in three main steps: we performed selective regression testing proposed in Chapter 3 (Section 3.1). We then performed test case minimization regression testing proposed in Chapter 3 (Section 3.2). Finally, we performed the test case prioritization proposed in Chapter 3 (Section 3.3). We used our guidelines to choose which combination of regression testing to use. We looked at our situations. There is a localized change in the model of the modified version of the ToDo App. The existing version has high quality so we do not need to use strong coverage criteria. It is not a critical system so there is no high risk. The app is under test behind schedule so we need to reduce the test cases. Therefore, based on these situations, we can use selective regression testing, test minimization, and test case prioritization combination. We need to apply the three main steps of the combination on the modified version of the ToDo App:

Step 1: We applied the selective regression testing of FSMApp [2] on the modified version of the ToDo App. We rebuilt the hierarchical model $HFSM'$ and then identified and classified changes for the modified version of the To Do App. Figure 4.3 shows the modified model $HFSM'$ for the ToDo App. Then we classified abstract test cases into: obsolete, retestable, and reusable for To Do App based on the type of changes and we selected the set of retestable test cases. Next, we determined if any parts have not been tested in To Do app. We used partial generation to generate tests from the $HFSM'$. Paths through FSMs/AFSM generated based on edge coverage. In aggregated paths to form abstract tests, we used at least one

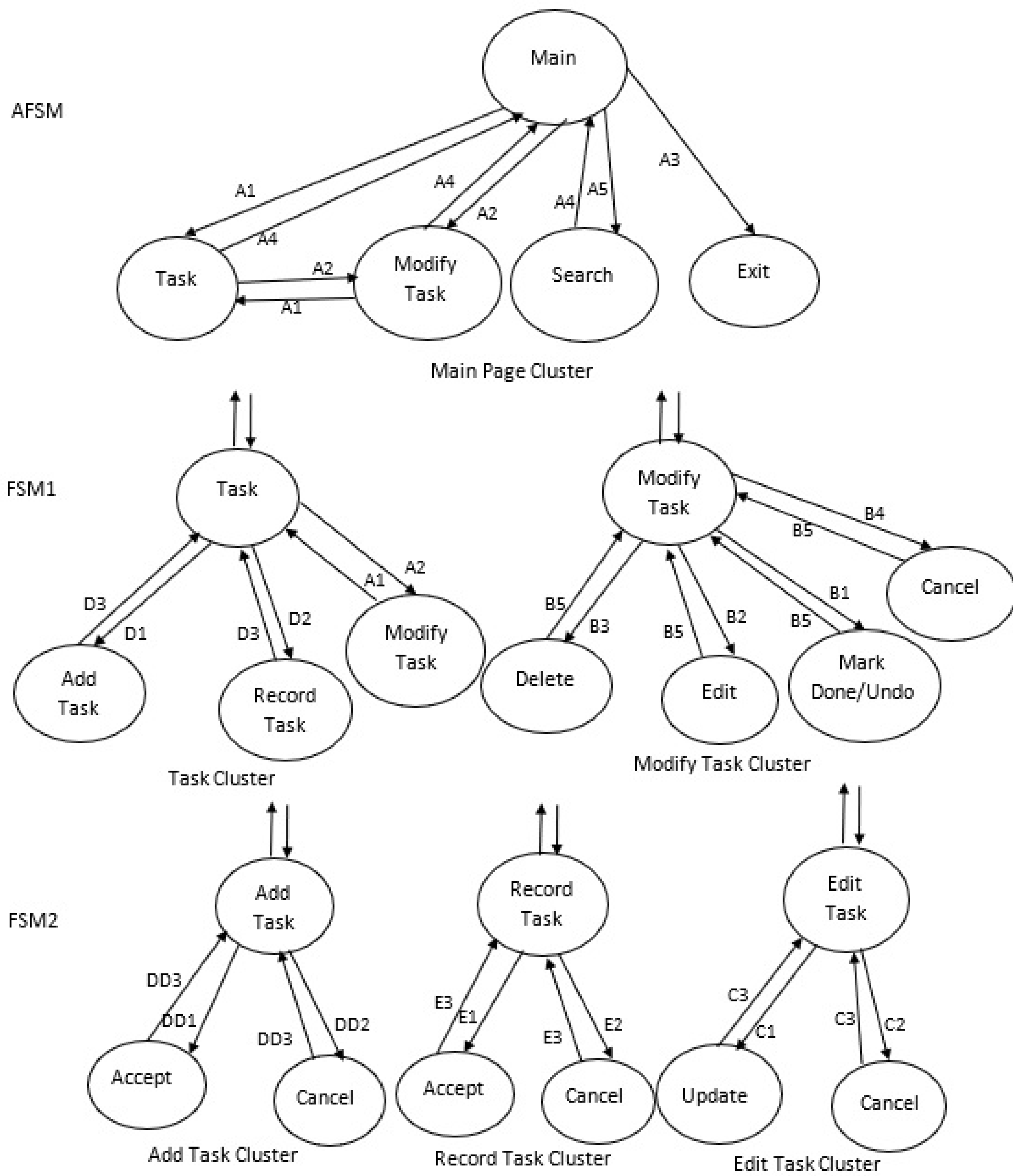


Figure (4.3) Modified Model $HFSM'$ for ToDo App

path coverage (it is not a strong coverage criterion). At the end of this step, we got a test suite (retestable and new tests) that meet all requirements but may have redundant test cases. Table 4.3 shows the selected test suite.

Table (4.3) The selected and new test paths of the To Do app

ID	Test Sequence	Length
1	[Main, Task, Add Task, *Accept*, Add Task, *Cancel*, Add Task, Task, Main, *Exit*]	10
2	[Main, Task, Record Task, *Accept*, Record Task, *Cancel*, Record Task, Task, Main, *Exit*]	10
3	[Main, Task, Modify Task, Edit, *Update*, Edit, *Cancel*, Edit, Modify Task, Task, Main, *Exit*]	12
4	[Main, Task, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Task, Cancel, Modify Task, Main, Exit]	13
5	[Main, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Main, *Exit*]	8
6	[Main, Modify Task, Edit, *Update*, Edit, *Cancel*, Edit, Modify Task, Main, *Exit*]	10

Step 2: We applied test case minimization of FSMApp [3] to minimize the selected test suite. We determined the updated test requirements and test cases based on selective regression testing for FSMApp in step 1 and used them to build the context table (Coverage Information Table). Then we applied the concept analysis to minimize test cases. In the concept analysis, we considered the test cases as objects O and the test requirements as attributes A.

$T = \{t_1, t_2, ..., t_n\}$, $TR = \{r_1, r_2, ..., r_m\}$

Let $T = O$, $TR = A$. Then $A_i = \{r | r \in TR : t_i \in T \wedge t_i \text{ meet } r\}$.

Next, we build a table of the concepts which are objects (test cases of the ToDo App) and attributes (test requirements of the ToDo App) and construct the concept

lattice. The object implication and attribute implication is applied to reduce the context table.

Object Implication: Given two tests $t_i, t_j \in T, t_i \Rightarrow t_j$ if and only if $\forall r \in TR$, (t_j R $r) \Rightarrow (t_i$ R $r)$.

Attribute Implication: Given two test requirements $r_i, r_j \in TR, r_i \Rightarrow r_j$ if and only if $\forall t \in T, (t$ R $r_i) \Rightarrow (t$ R $r_j)$.

The process of reducing the context table continues until no further implications are found. The minimized test suite is then selected from the final reduced context table. Table 4.4 shows the minimized test suite.

Table (4.4) The minimized Test Paths

No	**Test Paths**	**Length**
1	[Main, Task, Add Task, Accept, Add Task, Cancel, Add Task, Task, Main, Exit]	10
2	[Main, Task, Record Task, Accept, Record Task, Cancel, Record Task, Task, Main, Exit]	10
3	[Main, Task, Modify Task, Edit, Update, Edit, Cancel, Edit, Modify Task, Task, Main, Exit]	12
4	[Main, Task, Modify Task, Mark task done/ undo, Modify Task, Delete, Modify Task, Task, Cancel, Modify Task, Main, Exit]	13

Step 3: We applied test case prioritization of FSMApp approach. Here we prioritized the minimized test suite based on the complexity of each test case. We determined the complexity of each test case by determining the number of input values and actions. For example, Test Path1 has six inputs and five actions. The inputs are Task name (parname), Task Date and Time (ParD), (ParT) which occurs twice in test1. The actions are to click Add New Task button (b-ANT), Click Accept button(b-ANTA), Click Cancel (b-Cancel), and click the back arrow to exit the Mobile App (buttonBack). Table 4.5 shows the complexity of the test path1.

Table (4.5) The Complexity of the Test Path 1

Edge	**Constraint**	**Input Value**	**Actions**	**Complexity**
A1	R(b-Task)	0	1	1
D1	C1(SelectAddInfo, SelectRecodInfo)	0	1	1
DD1	R(Par(name,D,T),b-ANTA) S(Par(name, D,T), b-ANTA)	3	1	4
DD2	O(Par(name, D, T), R(b-Cancel) S(Par(name, D, T) b- Cancel)	3	1	4
A3	R(buttonBack)	0	1	1
Total Complexity				11

Then we assigned priority to each test case based on its complexity. Assigned high priority to test path that has the biggest number of inputs (values and actions). Table 4.6 shows the complexity and the priority for each test path.

Table (4.6) The priority of the Test Paths

No	**Complexity**	**Priority**
1	11	1
2	7	2
3	11	1
4	6	3

Lastly, we ordered the test cases in descending order (high priority to low priority). The test case that has high priority will be executed first. The prioritized test suite is: T = $\{t_1, t_3, t_2, t_4\}$.

Chapter 5

Validation via Case Study

5.1 Rationale for the Case Study

So far we explained our work using a simple example of a Mobile App. As a next step, we need to use Test-driven development (TDD) to evaluate our work via a larger case study. The DiscoverU Mobile App was chosen for validation of the regression testing approaches for Mobile Apps. DiscoverU App is a Mobile Application for mental health education. It was chosen as it was under ongoing development as a collaboration between a psychology and computer science team. This means there were several design and code changes as the app development moved from the initial prototype to a deployed Mobile Application. As an ongoing student project at the University of Denver, this Mobile App also allowed me to have access to all versions and full code without any restrictions that may be present for commercial or proprietary Mobile Applications.

Test-driven development (TDD):

TDD is a process of software development that implements test cases before the software is fully developed and, thus, repeatedly tests the software up until

completion. This requires access to each software version and with DiscoverU, I had the necessary access to employ Test-driven development (TDD).

5.2 Case Study Objective

The objective of this case study was to show how to effectively and efficiently apply the developed testing techniques in the different situations that can occur in the Mobile App development process. It was important to test and show how our guidelines (see Chapter 4) can help to efficiently choose which combination approach to use. The proposed recommendations and guidelines from Chapter 4 are used in a case study and we employ a regression test on a Mobile App that was in progress and not yet released at the time of working on this dissertation. This enabled us to use Test-driven development and evaluate our guidelines, and explore different situations that can occur in a real development process. With time often being a factor, the Case Study also highlighted the different situations that influence which combinations of regression testing approaches can be used efficiently and effectively to test software while in the ongoing development process. This is a challenging approach. Often testing is performed on the final version of a code only there might be limited access to software in development or developers who are willing to share all of their code in the process. However, testing on the final code only has a huge disadvantage: it does not highlight the situations where combined regression testing could discover and alleviate problems early on in the development process. We postulate that discovering errors in a Mobile Application early on can lead to overall more efficient and effective coding if our regression testing approach is used for each version, and save cost in the long run.

5.3 Preparation for Case Study

We need to analyze how the selected Mobile App for the case study works before applying the regression testing approaches. We need to study the functions of the DiscoverU Mobile App to understand all components and connections between the Mobile screens before applying the regression testing approaches. This was done so that the time for learning how the functions of the DiscoverU Mobile App work would not confound or increase the testing time. This was achieved by connecting with the Mobile App developer team. The App functions were discussed and an understanding of how the App will work was supported by using the prototype that the developing team designed in Figma [38] [65]. The Collaborative Interface Design Tool Figma is a web-based graphics editing and user interface design app. Figma is free in its basic version and works as an online user Interface (UI) prototyping tool. It enables designers to create and collaborate on prototypes and facilitates the hand-off to developers. Figma states that the users stay in the flow by editing, creating, and collaborating in one end-to-end tool. While the prototype in Figma is for a Mobile App, Figma itself is a web-based tool and works either in a browser or a desktop version.

5.4 Case Study Research Questions

The research questions derived from the case study objectives are as follows:

- RQ1: Can we show that regression test for each of the versions of the in-progress Mobile App using all of the variants described in Chapter 3?
- RQ2: Which of the situations described in the guidelines will occur and can we successfully use the guidelines for testing?

- RQ3: Are the options for the testing criteria described in Chapter 3 effective?
- RQ4: How do test case minimization and test prioritization help to make testing more efficient?

5.5 Units of Analysis

Table 5.1 shows the measurement units at every step of the regression testing process. The first column presents the steps of regression testing of FSMApp that need to be measured. The second column presents the measurements. It also describes the measurement of the percentage of reducing test cases by using selective regression testing, the Percentage of minimizing test cases, and how many test cases need to be applied to detect faults if test prioritization is not applied.

Table (5.1) Units of Analysis

Step	**Measurement**
Determine Types of Changes	Time to determine the changed parts of the App under test (add, delete, modify)
Build Model	Size (Num of Nodes, Num of Edge, Num of Clusters), time to build the model
Generate Test Sequences	Size (Number of test sequences, total test steps), time to generate the test sequences
Input Selection	Num of Inputs, Num of actions, Time to choose input
Execute Test Paths	Time of the execution
Selective Regression Testing	Percentage(%) of reducing test cases
Test Case Minimization	Percentage (%) of minimizing test cases
Test Case Prioritization	How many test cases need to apply to detect faults if we do not apply test prioritization

5.6 Case Study General Descriptions

DiscoverU is a mental health application for mental health education for adolescents. It is developed in collaboration between a psychology and computer science team. The goal of the DiscoverU Mobile App is to provide adolescents a no-cost, scientifically-based mental health support that empowers them to learn about their mental health and what they can do to improve it. The DiscoverU Mobile App development is in progress at the time of working on this dissertation. It is expected that the development of the software continues for at least two more years and that the

content development continues for several years after. The developing team consists of undergraduate students and graduate students with team leads from Computer Science (Dr. Kerstin Haring, University of Denver; Dr. Daniel Pittman, Metropolitan State University) and psychology team lead (Dr. Vicki Tomlin, University of Denver).

DiscoverU addresses various barriers to accessing mental health care in adolescents. One way the Mobile App addresses this is by featuring multiple levels of mental health education and support, while also providing various formats of user activities including writing a journal, and engaging practices to implement in everyday life. To appeal to the range of adolescents (i.e., a wide demographic aged 11–21 from various socioeconomic backgrounds), DiscoverU incorporates gamification to encourage users to remain consistent in the app in a healthy manner. DiscoverU also allows for personalized goals so the user can tailor their experience to their personal goals and the Mobile App meets the user where they are at in their mental health journey. The content of the DiscoverU app is created by the psychology team and can be pushed to the Mobile App through a web-based custom created Content Management System developed by the computer science team. This is expected to facilitate new and updated content in the long run without requiring code changes or updated versions of the Mobile App.

We have four versions of the DiscoverU Mobile App in addition to the original version of the app. (The original version had the same functions as the first version $DiscoverU_{V1}$ but it did not work correctly and had a lot of defects). We executed the test cases for the first two versions of the app ($DiscoverU_{V1}$, and $DiscoverU_{V2}$) on a simulation environment (test web-based), and the test cases for the second two versions of the app ($DiscoverU_{V1}(APK1)$, and $DiscoverU_{V2}(APK2)$) on a Mobile device. Since we are working on a Mobile App in progress (not released

as a Mobile App), we applied test-driven development process to regression test the versions of the DiscoverU Mobile App. We applied the FSMApp approach to testing the original version of the DiscoverU app, then in each time the developer team add any functions or features, we applied our regression testing approaches and by using our guidelines, we choose which type of regression testing technique or the combination of regression testing techniques to apply and then report the evaluation to the developer team so that help to alleviate the problems early in the development process.

It should be highlighted here that by employing TDD with regression testing on a web-based version first and then migrating with the development team to a mobile version, a **novel contribution is to show the portability from a web application to a Mobile App in this case study**.

The next sections describe the versions that we regression tested by applying our regression testing approaches and our guidelines to select the appropriate combination of regression testing to each version of the DiscoverU App.

5.6.1 First Version of the DiscoverU App ($DiscoverU_{V1}$)

The first version of the app ($DiscoverU_{V1}$) has the same functionality as the original version of the DiscoverU App. In the first version of the app ($DiscoverU_{V1}$), after the user creates an account, the user sees the main "DiscoverU" screen. The main screen has four tabs (Basecamp, Explore, Journey, Backpack) represented by an icon and a text description. The first tab (Basecamp) is activated with a click and shows eight buttons, but just three of them are working. When the user clicks on the first button ("How are you feeling"?), it takes the user to the screen for entering the user's feelings with the option to do this now or later in the day. Feelings can

be entered by selecting an emoji that represents the daily feelings best. When the user clicks on the button "free writing", it takes the user to a screen where the user can take a few minutes to free write in the user's journal. Another button is "Demo Theme" where the user can change the color of the screens of the App. The development team designed the buttons that state actions like keep climbing higher and start your next activity on the activity trail, read articles, 10 minutes of meditation, yoga to release tension, and watch videos but they still did not work. When clicking the second tab ("Explore"), the tab is supposed to expand to show all content activities to the user (Read, video, image, listen, etc.) but there was an issue that prevented the user from seeing the content or filter for these activities in the content. The Third tab ("Journey") is supposed to show all journeys, but in this first version the development was not finished and this was not an available action. The fourth tab ("Backpack") expands with a click on the Backpack icon to a screen with five tabs (Favorites, Rewards, Tools, History, and Extra Steps). Finally, in the "Menu", the user can go through all services of the app by clicking on the menu icon (Resources, Favorites, Backpack, Tools, Journal (sub-menu), Daily Check-in, Settings, and Logout). The sub-menu (Journal) expands to several activities (Free Write, Stress Log, Grief Journal, Emotion Reflections, Goals, and Check-ins). The Computer Science team designed the menu and the sub-menu for this first version, but not all services and actions were enabled yet. See Appendix B.

Since the original version did not work correctly and had several bugs, we applied a regression test on it after the developer team made corrective maintenance for the original version of the DiscoverU app without adding any new functions or features. This means that there was no change in the model. We considered this as the first version of the DiscoverU Mobile App ($DiscoverU_{V1}$). Here we applied the first set of regression testing which is a retest of all the original test cases (full prioritization

regression testing using strong coverage criteria) after we prioritized them so we can discover and alleviate problems early.

The situations for this version ($DiscoverU_{V1}$) were as follows: The quality of the app was low, so a strong coverage criterion was required. There was no change in the model, so it was chosen to do selective regression testing in terms of retestable test cases (retesting just the test cases that related to the defects). The app is not a critical system, so it does not have high risk. Since we aimed to develop and manage the quality of the first version of the App ($DiscoverU_{V1}$) in a way to best ensure that the first version meets the necessary developer requirements, we used strong coverage criteria and avoided minimizing the test cases. Therefore, in this version of the App ($DiscoverU_{V1}$), we applied a full prioritization regression testing. We prioritized the original test suite before re-executing it.

Validation Report for the First Version of the App ($DiscoverU_{V1}$) :

In this version ($DiscoverU_{V1}$), we executed the test cases generated using full prioritization regression testing and evaluated the results. The evaluation of the results presented that: The first version of the app ($DiscoverU_{V1}$) has five failed tests and 21 passed tests. Table 5.3 shows the execution results for the four versions of the App and the number of defects. Column 1 shows the versions of the DiscoverU App. Columns 2 and 3 show the number of tests that failed and passed, respectively. The last column shows the number of defects. This version has a defect related to the tab "Basecamp" in the "Free writing" button that allows the user to take a few minutes to free write in his/her journal. The button did not work correctly, the second defect found was the "Explore" tab, which had no content. It was unknown if the filter for all content worked correctly or not. The next defect found was on the path from the "Menu" to Favorites, Basecamp, and then "Free write". The free write function did not work correctly. Another defect found is related to the "sub-

menu" (Journal). When the user goes to the "Menu", "Journal", "Checkins", it displays the Daily Check-ins by the week but when selecting the week that the user already completed check-in for, it did not display the result as expected. The last defect is related to the "Menu", when we do logout, the down tabs is disappeared in the app.

5.6.2 Second Version of the DiscoverU App ($DiscoverU_{V2}$)

The developer team made changes and added new functions to the Second version of the DiscoverU app. In this version of the app ($DiscoverU_{V2}$), we were able to see the content, and we could view the activities, journeys, and trails that were added from the web application by the psychology team. Thus these new services were added to the second version of DiscoverU app ($DiscoverU_{V2}$). Also, there were some new services added to the "menu" such as "Free Write". So, these changes in the second version of the app ($DiscoverU_{V2}$) required us to make changes in the model. These modifications required adding new nodes and new edges in the model because the type of some nodes changed from logical app pages (LAP) nodes into cluster nodes. The change in the model was not an extensive change, as it was isolated to a few partitions of the model.

Based on our guidelines proposed in Chapter 4, in version ($DiscoverU_{V2}$) we selected a combination of regression testing (selective, minimization, and prioritization) to apply. The situations for the version ($DiscoverU_{V2}$) were: the quality of the app was not low. We did not need to use strong coverage criteria, so there was no limitation in using the test case minimization approach. There was a local change in the model. There were some constraints on the time to give feedback to the development team.

We first applied our selective regression testing approach proposed in Chapter 3. We determined the type of changes and classified the test cases into obsolete, retestable, and reusable tests. The new test cases added to test uncovered parts of the app. Then the selected test suite is minimized by applying our test case minimization approach proposed in Chapter 3 to remove the redundant test cases. Next, the minimized test suite is prioritized by applying our test case prioritization approach, proposed in Chapter 3, to execute them.

Table 5.4 shows the classification of the tests based on the type of changes and the number of all tests of each version of the DiscoverU App. In this version ($DiscoverU_{V2}$), there are 11 new test cases, 8 retestable, and 16 reusable tests. In this version, we executed 14 test cases that were generated by applying a combination of regression testing (selective, minimization, and prioritization).

Validation Report for the Second Version of the App ($DiscoverU_{V2}$) :

In version ($DiscoverU_{V2}$) we executed the test cases generated using a combination of regression testing (selective, minimization, and prioritization) approach and evaluated the results. The evaluation of the results presented that the second version of the app ($DiscoverU_{V2}$) has two failed tests and 12 passed tests. Table 5.3 shows the execution results for the four versions of the App and the number of defects. There was a defect related to the content in the "Explore" tab. The video in the content did not work. The other defect is related to the menu, when we do "logout", the down tabs is disappeared in the app.

5.6.3 Third Version of the DiscoverU App $DiscoverU_{V3}(APK1)$

We put this version of the app ($DiscoverU_{V3}(APK1)$) on an android phone. (Device name: moto z4, Android version: 10). The developer team made changes

and added new services to this version of the app ($DiscoverU_{V3}(APK1)$). These changes caused an issue that prevented the end-user to see the content. In the second tab "Explore" there was no content. Because of this change, we removed the nodes and the edges related to this change and which resulted in obsolete tests. Also, in the fourth tab "Backpack", the developers added a function called "Rewards". That change required changing the type of some nodes from the logical app page (LAP) into cluster nodes and adding new nodes and edges. And in the sub-menu ("Journal") the developers added the "Emotion Reflections" service.

Based on our guidelines proposed in Chapter 4, in version ($DiscoverU_{V3}APK1$) we selected a combination of selective regression testing and test case prioritization approach. The situations for this version ($DiscoverU_{V3}APK1$) were that the quality of the app was not low. We did not need to use strong coverage criteria, so there was no limitation to using the test case minimization approach. There also was a local change in the model. In this case, there were no constraints on the time to give feedback to the development team.

We first applied our selective regression testing approach proposed in Chapter 3. We determined the type of changes and classified the test cases into obsolete, retestable, and reusable tests. The new test cases were added to test the uncovered parts of the app. Then the selected test suite is prioritized by applying our test case prioritization approach proposed in Chapter 3. We selected this combination because we did not have constraints on the time the team needed feedback. We also needed to be confident about the adaptive maintenance that was made by the team to transform from the simulation environment to the Mobile device, so it does not affect the existing parts of the app.

Validation Report for the Third Version of the App $DiscoverU_{V3}APK1$

In version ($DiscoverU_{V3}APK1$) we executed the test cases generated using a combination of regression testing (selective and prioritization) approach and evaluated the results. The evaluation of the results presented that the third version of the app ($DiscoverU_{V3}APK1$) has four failed tests and ten passed tests. Table 5.3 shows the execution results for the four versions of the App and the number of defects. The first defect was that we could not create a new account. The app login with the account is created in a different environment. The second defect was that the "Explore" tab had no content. We could not see the activities and trails that we intended to be there; thus, we could not know if they worked correctly or not or if the content filter worked correctly or not. The other defect was related to the sub-menu ("Journal"). When we go to the Menu, Journal, and Checkins, it displays the Daily Check-ins sorted by weeks. However, when we selected the week that we made a Check-in, it did not display the result as expected. These two defects are the same defects that were present in the first version of the app ($DiscoverU_{V1}$). After the development team made modifications to transform the software from web to app, these defects happened again. Also, the same defect from the previous version ($DiscoverU_{V2}$) related to the Menu was still not fixed: when we logged out of the App, the down tabs still disappeared.

5.6.4 Fourth Version of DiscoverU App $DiscoverU_{V4}(APK2)$

We put the fourth version of App ($DiscoverU_{V4}(APK2)$) on a Mobile device (android phone). (Device name: TCL Model: A509DL, Android version: 11). The developer team made changes and added new services to this version of the app ($DiscoverU_{V4}(APK2)$). They fixed the issue that prevented the user from seeing

some of the content. In this version, the second tab ("Explore") shows the intended content. The team also added new services in the sub-menu ("Journal") where they added the "Grief Journal", "Stress Log", and "Goals". That change required changing the type of some nodes from the logical app page (LAP) into cluster nodes and adding new nodes and edges.

Based on guidelines proposed in Chapter 4, in this version $DiscoverU_{V4}(APK2)$ we selected a combination of selective regression testing and test case prioritization. The situation for this version ($DiscoverU_{V4}(APK2)$) was that the quality of the app was medium. We did not need to use strong coverage criteria, so there was no limitation to using the test case minimization approach. There was a local change in the model. In this version, there were no constraints on the time to give feedback.

We first applied our selective regression testing approach as proposed in Chapter 3. We determined the type of changes and classified the test cases into obsolete, retestable, and reusable tests. The new test cases were added to test uncovered parts of the app. Then the selected test suite is minimized by applying our test case minimization approach proposed in Chapter 3 to remove the redundant test cases. Next, the minimized test suite is prioritized by applying our test case prioritization approach proposed in Chapter 3 to execute them.

Validation Report for the Fourth Version of the App $DiscoverU_{V4}APK2$

In version ($DiscoverU_{V4}APK2$) we executed the test cases generated using a combination of regression testing (selective, minimization, and prioritization) approach and evaluated the results. The evaluation of the results presented that the fourth version of the app ($DiscoverU_{V4}APK2$) has six failed tests and 11 passed tests. Table 5.3 shows the execution results for the four versions of the App and the number of defects found. The same defects from the previous version ($DiscoverU_{V3}(APK1)$) were still not fixed. The defects were related to the "Lo-

gout", and to the "Check-ins" in the sub-menu ("Journal"). The other four defects were found in the sub-menu ("Journal") and were related to "Free Write", "Emotion Reflections", "Grief Journal", and "Stress Log".

5.7 Validation Results and Discussion

5.7.1 Applicability

RQ1: Can we show that we can regression test the various in-progress versions using all of the variants described in Chapter 3?

We successfully regression tested the various in-progress versions using all the variants that we described. We applied the three regression testing approaches and the combination of regression testing approaches that we proposed to the various in four progress versions of the DiscoverU App using all of the variants. We successfully applied full prioritization regression testing for the first version of the App ($DiscoverU_{v1}$), a combination of selective regression testing, test case minimization, and test case prioritization for the second version of the App ($DiscoverU_{v2}$), a combination of selective regression testing, and test case prioritization for the third version of the App ($DiscoverU_{v3}(APK1)$), and for the fourth version of the App ($DiscoverU_{v4}(APK2)$), we successfully applied a combination of selective regression testing, test case minimization, and test case prioritization. Table 5.2 shows the types of techniques that we applied and the situations for each version of the DiscoverU App. Tables 5.3 to 5.6 report our results for the four versions of the DiscoverU App.

Table (5.2) Type of used techniques and the situations for each version

App Versions	Techniques	The Situations for Each Version
$DiscoverU_{V1}$	Full Prioritization Regression Testing/ Combination (sel, pri)	Low quality, Strong coverage criteria, Low risk, No constraints, and No model change
$DiscoverU_{V2}$	Combination of (selective, minimization, and prioritization)	High quality, No Strong coverage criteria, Low risk, Some constraints on the time, and Local change in the model
$DiscoverU_{V3}(APK1)$	Combination of selective and prioritization	High quality, Strong coverage criteria, Low risk, No constraints on the time, and Local change in the model
$DiscoverU_{V4}(APK2)$	Combination of selective, minimization, and prioritization	Medium quality, No Strong coverage criteria, Low risk, Some constraints on the time, and Local change in the model

RQ2: Will the various situations described by the guidelines occur and can we successfully use the guidelines for testing?

We successfully used the guidelines for testing that we described in Chapter 4 and we successfully showed the various situations described by the guidelines that may occur.

We used the recommendations and guidelines along a case study and we regression tested software that is in progress and not yet released. The DiscoverU Mobile app enabled us to use test-driven software development and showed the different

situations that can occur. It also showed that the situations influenced which combination of regression testing approaches can be used efficiently and effectively to test software while in the ongoing development process. For example, we applied a combination of selective regression testing, test case minimization, and test case prioritization to the second version of DiscoverU App ($DiscoverU_{v2}$) because the situations in the second version of the DiscoverU App ($DiscoverU_{v2}$) were that we had a good quality version and we did not need to use strong coverage criteria, thus we could use test case minimization. The DiscoverU App is not a critical system and it does not have a high risk, thus we can use the reduction techniques (selective, minimization). We had a tight timeline to give feedback to the developer team so they can move to the next step for developing the app, therefore this combination was preferred to save time by minimizing and prioritizing the test cases to relieve the problems early in the development process for the DiscoverU App. Based on our guidelines, we successfully selected the appropriate combination of regression testing approaches for each version of the DiscoverU App. Table 5.2 shows the situations that guided us to choose the appropriate technique to apply to each of the four versions of the DiscoverU App.

5.7.2 Effectiveness

RQ3: Are the options for testing criteria described in Chapter 3 effective?

The options for testing criteria described in Chapter 3 were effective. To evaluate the effectiveness, we used strong coverage criteria and weak coverage criteria to show that these different coverage criteria were effective to detect the defects in the versions of the DiscoverU App. The case study executes the test cases and captures

the number of defects. We used different coverage criteria to generate abstract test cases for the versions of the DiscoverU App, and they were effective. Applying our proposed approaches and the coverage criteria that we used for the four versions of the DiscoverU App, successfully showed the effectiveness. Table 5.3 shows the execution results for the four versions of the DiscoverU App. Column 1 shows the versions of the DiscoverU App. Columns 2 and 3 show the number of tests that failed and passed, respectively. The last column shows the number of defects.

Table (5.3) Defect Summary

App Versions	**No of Failed**	**No of Passed**	**No of Defects**
$DiscoverU_{V1}$	5	21	5
$DiscoverU_{V2}$	2	12	2
$DiscoverU_{V3}(APK1)$	4	10	4
$DiscoverU_{V4}(APK2)$	6	11	6

5.7.3 Efficiency

RQ4: How does minimization and prioritization help make testing more efficient?

Using the combinations with minimization and prioritization helped make the testing more efficient. To better investigate the efficiency of the combination of our proposed approaches, we conducted a comparison between the versions of the case study (DiscoverU Mobile App).

To evaluate the efficiency, the time to determine changes and classify tests, the test generation process, the length of test sequences, and test execution effort (time) are measured. Table 5.4 presents the classifications of the tests based on the changes that were made for each version of the DiscoverU App during the development pro-

cess, the number of all tests for the full regression test of the four versions of the DiscoverU App, and the time for determining the changes and classifying the tests. Column 1 shows the versions of the App. Columns 2-5 show the test classification as obsolete, retestable, reusable, and new tests, respectively. Column 6 shows the number of full regression tests. The last column shows the time to determine the changes in minutes (the time is measured by the tester). The first version of the DiscoverU App ($DiscoverU_{V1}$) has no changes in the model because the developer team made only corrective maintenance to fix the defects. The time to determine the tests related to these defects was seven minutes. In the second version of the DiscoverU App ($DiscoverU_{V2}$), the time to determine the changes and map them to related tests was 12 minutes. In the third version of the DiscoverU App ($DiscoverU_{V3}(APK1)$), the time to determine the changes and map them to related tests was 13 minutes. In the fourth version of the DiscoverU App ($DiscoverU_{V4}(APK2)$), the time to determine the changes and map them to related tests was 20 minutes.

Table (5.4) Summary of the Changes

App Versions	Obsolete Tests	Retestable Tests	Reusable Tests	New Tests	Full Tests	Time
$DiscoverU_{V1}$	0	16	10	0	26	7
$DiscoverU_{V2}$	3	8	16	11	35	12
$DiscoverU_{V3}(APK1)$	5	8	22	6	36	13
$DiscoverU_{V4}(APK2)$	0	14	22	8	44	20

Table 5.5 presents a summary for each version of the DiscoverU App for model size and time for the model generation phase. Column 1 shows the version number of the DiscoverU app. Columns 2-4 show the number of nodes, edges, and clusters of the model, respectively. The last column shows the model generation time in

minutes. The model generation time is measured by the designer while drawing the model. The first version of the DiscoveU App ($DiscoverU_{V1}$) has the smallest model with 46 nodes and 116 edges and 32 clusters. The model generation time for the first version of the DiscoveU App ($DiscoverU_{V1}$) is zero because there was no change in the model. So, the model for the first version of the App ($DiscoverU_{V1}$) is the same model for the original version of the DiscoverU App. The time for generating the original model was 60 minutes. The fourth version of the DiscoverU App ($DiscoverU_{V4}(APK2)$) has the largest model with 89 nodes, 234 edges, and 84 clusters. The time for generating the model of the fourth version of the DiscoverU App ($DiscoverU_{V4}(APK2)$ was 15 minutes. The model generation time of the three versions of the DiscoverU Apps ($DiscoverU_{V2}$, $DiscoverU_{V3}(APK1)$, ($DiscoverU_{V4}(APK2)$) took between 6 to 15 minutes.

The version that has more clusters takes the maximum model generation time because the clusters require more time to identify and find the connection between them. However, because the model was about the same App and all changes and modifications were local, isolated to some parts of the app, it saved much time to regenerate the model for the versions by editing and adding new parts for the same model instead of completely regenerating the model.

Table (5.5) Model Size Summary

App Versions	**Nodes**	**Edges**	**Clusters**	**Model Time**
$DiscoverU_{V1}$	46	116	32	0
$DiscoverU_{V2}$	68	180	52	10
$DiscoverU_{V3}(APK1)$	72	187	53	6
$DiscoverU_{V4}(APK2)$	89	234	84	15

Table 5.6 shows the results of applying our techniques to the four versions of the DiscoverU Mobile App. Column 1 indicates the versions of the DiscoverU app. Columns 2-3 show the number of test cases, and the time (in minutes) for generating the test sequence. The time is calculated by the tester. This includes the generation of cluster test paths and the aggregation test sequences. Columns 4–6 show the number of input values, number of actions, and time to choose the inputs (in minutes) for the input selection phase. The time is calculated by the tester. The time includes the identification of input boundaries and input selection to execute the test sequences. Column 7 shows the execution time in minutes and Column 8 shows the total time in minutes for all steps: Determine the changes and classify the tests, model generation, test case generation, input selection, and test case execution.

The time for model generation and the time to generate tests for the first version of the App ($DiscoverU_{V1}$) were zero because there was no change in the model. Therefore, the total time for the first version of the App ($DiscoverU_{v1}$) shows the time to choose new inputs and the time to re-execute the original test suit.

The total time range for the three versions of the DiscoverU App ($DiscoverU_{V2}$, $DiscoverU_{V3}(APK1)$, and ($DiscoverU_{V4}(APK2)$) is between 97–126 minutes. We applied a combination of regression testing (selective, minimization, and prioritization) for the second version of the app $DiscoverU_{V2}$, a combination of selective regression testing, and test prioritization for the third version of the app $DiscoverU_{V3}(APK1)$, and a combination of regression testing (selective, minimization, and prioritization) for the fourth version of the app $DiscoverU_{V4}(APK2)$.

Table (5.6) Summary of Test Generation and Execution

App Versions	Tests	Time	Inputs	Action	Choose Time	Exe Time	Total Time
$DiscoverU_{V1}$	26	0	21	146	19	10	29
$DiscoverU_{V2}$	14	30	8	144	25	20	97
$DiscoverU_{V3}(APK1)$	14	25	9	112	33	12	89
$DiscoverU_{V4}(APK2)$	17	32	11	156	40	25	126

In the first version of the app ($DiscoverU_{v1}$), When we applied selective regression testing in terms of rerun retestable test cases, we removed 10 tests from the full regression testing test suite. It means that we executed 16 test cases instead of 26 test cases (execute 62% of all tests).

For the second version of the app ($DiscoverU_{v2}$), when we applied selective regression testing we removed 16 tests from the full regression testing test suite. It means that we will execute 19 test cases instead of 35 test cases (execute 54% of all tests). Then when we minimized the selected test cases, we removed five redundant test cases from the selected test suite. It means that we will execute 14 test cases instead of 35 test cases (execute 40% of all tests). The minimization makes the test more efficient. For the third version of the app ($DiscoverU_{v3}(APK1)$), when we applied selective regression testing we removed 22 tests from the full regression testing test suite. It means that we will execute 14 test cases instead of 36 test cases (execute 39% of all tests). For the fourth version of the app ($DiscoverU_{v4}(APK2)$), when we applied selective regression testing we removed 22 tests from the full regression testing test suite. It means that we will execute 22 test cases instead of 44 test cases (execute 50% of all tests in the test suite). Then when we minimized the selected test cases, we removed five redundant test cases from the selected test

suite. It means that we will execute 17 test cases instead of 44 test cases (execute 39% of all tests). The minimization makes the test more efficient. Table 5.7 shows the summary of reducing and minimizing test cases.

The comparison between applying full regression testing and applying selective regression testing and test minimization for the versions of the DiscovrU app shows the efficiency of the selected combination. Also, applying a test case prioritization helps to make testing more efficient. For example, if we did not prioritize the test suite for the second version of the app ($DiscoverU_{v2}$) we would need to execute seven test cases to find the first defect. But by applying test case prioritization, we found the first defect after executing three test cases already. In the full regression testing of the first version of the app ($DiscoverU_{v1}$), without prioritizing the test suite, we found one defect after executing three tests, and two more defects after executing 12 test cases, and last two defects after executing 21 tests. By applying test case prioritization, we found three defects after executing five test cases, and two more defects after executing nine test cases. The prioritization helps make the testing more efficient in terms of finding the defects earlier. We can conclude that minimization and prioritization help to make testing more efficient.

Table (5.7) Summary of Reducing and Minimizing Test Cases

App Versions	**% Reducing Test Cases**	**% Minimizing Test Cases**
$DiscoverU_{V1}$	62% (16 test cases instead of 26)	-
$DiscoverU_{V2}$	54% (19 test cases instead of 35)	40% (14 test cases instead of 35)
$DiscoverU_{V3}(APK1)$	39% (14 test cases instead of 36)	-
$DiscoverU_{V4}(APK2)$	50% (22 test cases instead of 44)	39% (17 test cases instead of 44)

5.8 Threats to Validity

We performed a case study with four versions to evaluate the applicability, effectiveness, and efficiency of our proposed approaches and guidelines to select an appropriate combination of regression testing approaches. We tested the versions of the DiscoverU Mobile App on two different environments: a simulation environment on a desktop and an Android Mobile device.

Wohlin et al. [115] define external validity as the extent to which it is possible to generalize the findings in a case study. The main threat to external validity we encountered was a generalization. Even though our guidelines cover all possible situations and help to select an appropriate combination of regression testing the finding from the case study are not fully generalizable. While we consider the case study a proof of concept, we cannot generalize the results to other platforms. For example, we applied our proposed approaches to an Android Application only and did not consider IOS where the same Mobile Application remains to be tested on a different operating system. Generalizability is limited as with any case study. We cannot guarantee that a future case performs well under the proposed guidelines and gives the same result for other applications.

In Wohlin et al. [115], construct validity refers to the extent that the operational measures reflect what the researcher had in mind. The internal threat was related to the measurement. For example, as we keep creating the model, we are becoming more familiar with it. It is likely that the time to evolve the model decreases as we go on further. It is important to keep in mind that learning effects might bias the times needed to test each Mobile App. The learning effect in this case describes the time it takes to understand how all functions of the Mobile App work. If the effect is left unchecked it could possibly lead to longer testing times for the first

testing method applied to an app than expected, together with the accompanying effects that the developer might perceive testing as taking up too many resources like time or overpromise on the time it takes to provide feedback back to them from the testing process. To avoid this confounding factor, we studied the functions of the DiscoverU Mobile Apps in detail to understand all components and the connections between the Mobile screens, before applying the regression testing approaches. The learning effects were controlled by carefully analyzing how the app worked before applying the testing.

5.9 Challenges and Successes in our Case Study

This section is dedicated to the challenges, successes, impressions, and takeaways we gained as software testers during the described case study. The impressions here are not technological in nature, but they describe an often underestimated challenge to software testing: The necessity to work in teams and to collaborate and communicate with the developer team. During testing of the DiscoverU Mobile App, we noticed challenges around the time constraints of the developer team, the (miss-) communication with the team, and an initial lack of understanding of what testing does or how it can be helpful. While this dissertation is dedicated to the importance and contribution of testing, it is not always clear how to best manage teams to understand this [12]. Over the course of testing, however, we noticed changes in how the team perceived testing. After running the first test suites and summarizing the results to the developer team, they started to recognize the contributions to their efficiency and effectiveness as coders and started utilizing the report of validity as a guide to uncover defects that were not prevalent in their ad-hoc testing, find the code parts related to a defect and fix them in a more timely manner for the next

version. Offutt et al. [12] discussed the goals of testing in terms of the test process maturity levels, of an organization, where the levels are characterized by the testers' goals. For example, the lowest level is Level 0, where there is no difference between testing and debugging.

The DiscoverU developer team was looking to test at level 0, viewing testing as the same as debugging. This is not unexpected as, this view is naturally adopted by many undergraduate computer sciences (CS) majors or less experienced coders, especially when they encounter their first large-scale, multi-faceted project. In many CS programming classes, students get their program to a state where it compiles, or interprets without syntax errors, and then either debug their code if necessary and then test the programs with a few inputs chosen either arbitrarily or provided by the instructor. However, this model of testing does not distinguish between a program's incorrect behavior and mistake within the program and does very little to help develop software that is reliable or safe. It has to be acknowledged that regression testing often is out of the scope of anyone who is just learning how to code. Similar to this test case, the developer team included undergraduate students and it was only later on in the process that they acknowledge the purpose of testing and its contributions to elevate their software to the next maturity level. In summary, many of the initial human-related challenges we encountered is because there was no plan for testing (Level 0) however the importance of the developer-tester collaboration was recognized and appreciated further into the testing process.

Chapter 6

Future Work

This dissertation describes novel contributions to regression testing (selective, minimization, and prioritization) for Mobile Apps and proposes new guidelines for combining regression testing approaches. In our future work, we plan to continue to extend our research contributions as follows: (1) Minimizing test cases for Mobile Apps with a guarantee of the aggregation test requirements by minimizing FSMs before aggregating test paths. (2) Prioritizing test cases for Mobile Apps by weighting the input types. (3) Applying our proposed approaches of regression testing of FSMApp (selective regression testing, test case minimization, and test case prioritization) and using the guidelines for combining regression testing approaches in different system domains. (4) Building tools to generate FSMApp model and test paths. (5) Execute test paths for the case study DiscoverU App automatically using Appium.

6.1 Minimizing Test Cases with Guarantee of the Aggregation Test Requirements

We proposed a new approach to minimize test cases of Mobile Apps, however, this did not include a guarantee of the aggregation test requirements. We plan to minimize test cases of Mobile Apps with a guarantee of the aggregation test requirements by minimizing $FSMs$ before we aggregate test paths. To achieve this, we will minimize the FSM model to reduce the number of test paths in Mobile Apps. Minimizing the FSM model means removing any equivalent states. The equivalence differs from one technique to another in its meaning. For example, state equivalence as $State\ i = State\ j$ if and only if for every single input X the outputs and the following states are the same. Removing the equivalence FSMs before aggregating the test paths will positively affect the number of test paths. Our plan is to first minimize the FSMs model by removing any equivalent states and examining the effect of minimization on the generated number of test paths.

6.2 Prioritizing Test Cases by weighting the Input Types

We plan to prioritize test cases by weighing the inputs for each test case and then calculating the complexity for each test case. The test path that has the highest weight has the highest complexity, and will be assigned a higher priority. The test path that has the highest priority is executed first. Basically, this idea works by assigning a weight value for each input and calculating the complexity of these weight values for each test case. The most important input is the higher of the

weight values. For example, the importance of a menu-open action might be lower than a button-click action because the former simply calls up a menu list, but the button-click action might trigger one or more functions to change the state of the app. Therefore, the relationships between input types and errors dictate how we will set different weights for different input types. We will assume that higher-weighted events will lead to higher rates of fault detection than lower-weighted inputs due to their complexity.

6.3 Applying our Proposed Approaches and Guidelines in Different System Domains

This dissertation is focused on the Mobile Applications domain. We applied our proposed approaches for regression testing of FSMApps (selective, minimization, and prioritization) and guidelines on Mobile Apps. We plan to apply these proposed approaches and guidelines for combining regression testing approaches in different system domains such as safety-critical systems, robotics devices, and medical systems. We also plan to apply them to Internet of Things (IoT) Applications such as smart homes, smart cities, wearable devices, smart grids, connected health, and connected cars.

6.4 Building Tools

So far, there is no tool to generate FSMApp model or test paths. An important contribution to the field, therefore, is to build tools to automate model and test phases. This would decrease the cost of generating the model, test paths, and choosing the input and action, significantly increasing overall efficiency.

6.5 Finish and Execute Test Paths for the Case Study DiscoverU App Automatically

We plan to finish regression testing for the following versions of the case study DiscoverU and execute the final set of test cases for the app automatically. In our case study, we executed the test paths manually. To achieve the goal of executing test paths automatically, we plan to implement the code for Appium. Appium is an open source test automation framework that is flexible, enabling testers to write test scripts against multiple platforms such as iOS, Windows, and Android [104]. We also plan to use a partial regeneration approach [15] to cover the gaps that were caused by obsolete tests, and when the edges change by changing the input types and input constraints. Because of the hierarchical nature of FSMApp models and the associated stages of test generation, it is possible to use partial regeneration to replace obsolete test cases and make the testing process more efficient.

Chapter 7

Conclusion

This dissertation describes a unified approach to regression testing for Mobile Applications. To achieve this goal, we proposed three novel approaches of regression testing for Mobile Applications: selective regression testing, test case minimization, and test case prioritization. We unified these approaches by proposing new guidelines to combine these regression testing approaches.

In the state of the art literature related to the regression testing of Mobile Applications there are several limitations in the techniques for regression testing. The existing techniques so far did not address all aspects of selective regression testing, and they also did not address test prioritization or test minimization for Mobile Applications. In addition, the literature related to the combination of regression testing approaches lacks the combination of all types of regression testing. There was no systematic approach to how and when to select a particular combination of regression testing approaches. This thesis addresses these limitations and closes those gaps for the regression testing of Mobile Applications.

Mobile Applications tend to change frequently posing challenges for software developers to use regression testing efficiently. This dissertation presents regression

testing approaches for FSMApp, a black box testing technique for Mobile Applications. We addressed all aspects of selective regression testing in our new selective regression testing approach. This did not exist in the state of the art literature yet. So far, FSMApp has not addressed regression testing in general, nor has it addressed test case minimization, or test case prioritization in particular. In this thesis, we developed a test case prioritization of FSMApp and a test case minimization of FSMApp for Mobile Applications. We also developed guidelines on how and when to select a particular combination of regression testing approaches. With those guidelines, we considered all possible combinations of all types of regression testing approaches. The summary of this dissertation is as follows:

First, we proposed selective regression testing. We extended regression testing of FSMWeb from a web-based approach to also test Mobile Applications. Based on the changes to the behavioral model of Mobile Applications, the original set of tests could be classified as obsolete, retestable, and reusable. This extended approach covered the gap resulting from obsolete tests by generating new tests to achieve the coverage. We also determined if any parts of the modified model had not been tested yet, and if not we generated new tests for them. This means that we addressed all aspects of selective regression testing. We also considered the coverage criteria in our work where we used edge coverage criteria for individual FSMs and at least one coverage criteria for aggregating test paths. In the following, we applied our approach to an example of a Mobile Application.

Second, we proposed a test case minimization approach. It minimized the test cases by removing redundant tests based on concept analysis. The test case minimization approach for FSMApp generates a test suite T for Mobile Applications that satisfies the test requirements based on graph coverage criteria and a test aggregation criteria for its hierarchical model. Together, these constitute a set of test

requirements R. This approach solves the following problem: Given a test suite T and a set of test requirements R find a minimal subset of T that covers R. The test case minimization of the FSMApp approach for Mobile Applications is based on concept analysis and removes redundant test cases. It explores under which conditions minimization is not possible and it explores what test coverage criteria are likely to lead to no reduction in tests. In this dissertation, we applied our approach to an example for a Mobile Application. We were able to reduce the set of tests without losing the coverage of all test requirements.

Third, we proposed an approach to prioritize test cases for Mobile Applications based on input complexity. It is most desirable to select test cases that are most likely to reveal defects in the application under test. We applied our proposed approach of prioritizing test cases to prioritize the test cases that we obtained from the test minimization approach on a simple Mobile App as a proof of concept. We prioritized tests based on the input complexity, since a higher complexity of inputs often is associated with a more complex functionality which in turn would make it more fault-prone.

Next, in this dissertation, we proposed guidelines for combining the three types of regression testing (selective, minimization, and prioritization). Regression testing is an important activity in software maintenance and software enhancement as it can help developers to determine if changes made to the system are handled appropriately without compromising efficiency. While not employed often, it is possible to combine several regression testing techniques. Combining them can lead to a more efficient and effective regression test suite. The three types of regression testing can be combined in four different ways: Three ways to combine two of the approaches and one way to combine all three. However, to efficiently and effectively employ regression testing, it is crucial to select the appropriate combination for the soft-

ware that is to be tested. The guidelines in this dissertation address exactly that. For example, the expected quality of the software, the kind of changes made, or the criticality of the software heavily influence which strategies to combine and in what order. The guidelines presented in this dissertation combine regression testing approaches systematically. Additional to these conditions (software maintenance process, software enhancements, and the types of maintenance), we outlined all possible situations that can occur and showed how each of them influences which combination to use. We present recommendations for combining regression testing approaches and describe situations of different coverage criteria, the impact of changes, time constraints, available resources, levels of risk failure, and confidence in the quality of the software. All those factors influence which combination of regression testing approaches can be used to be most effective and efficient. These situations are used as guidelines to employ different combinations of regression testing approaches and can be applied directly by developers and testers.

Finally, we validated the proposed approaches and the guidelines for combining regression testing approaches with a case study. We applied our guidelines to a Mobile Application (DiscoverU). DiscoverU is a mobile application in progress and was not yet released at the time of writing this dissertation. Applying our proposed approach to a mobile application in progress enabled us to use test-driven software development and show the different situations that can occur in a real-world example. It also highlighted the situations that influence which combination of regression testing approaches is used efficiently and effectively to test Mobile Applications while they are in the ongoing software development process. This posed several challenges, that are also present in industry software development and testing. For example, often there is limited access to software in the development process and developers are not always willing to show all their code in the process. This has the consequence

that these approaches are tested on final software, which often has been extensively tested otherwise before release, and in this case, testing a final version does not highlight all the situations that combined regression testing could discover and also problems it could alleviate early in the development process. This dissertation used the case study to investigate the applicability, efficiency, and effectiveness of our proposed regression testing approaches and our guidelines for combining regression testing approaches for ongoing software development. We found that the combination of testing approaches helps to make the testing more efficient as less time is needed to test, and more effective as defects are caught earlier in the development process. In addition, we were able to show in our approach the portability from a web application to a Mobile Application. In the case study, we showed this by testing the software first on a simulation environment (desktop) and then, after the developer team made adaptive maintenance, we tested the software on a mobile device (Android).

We conclude that the use of FSMApp is a contribution to the body of knowledge of Model-based regression testing that has been lacking so far in the literature. FSMApp can be applied effectively and efficiently to minimize and prioritize test cases for Mobile Applications. We were able to show that the use of a combination of regression testing approaches had a positive impact on saving time and that defects can be identified early in the software development process.

Bibliography

[1] Android. https://www.android.com/. Accessed: 2021-10-16.

[2] Zeinab Abdalla, Anneliese Andrews, and Ahmed Alhaddad. Regression testing of mobile apps. In *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1912–1917. IEEE, 2021.

[3] Zeinab Abdalla, Anneliese Andrews, and Ahmed Alhaddad. Test minimization for mobile app testing with fsmapp. In *2022 International Conference on Computer Science, Computer Engineering, & Applied Computing (CSCE'22)*. Springer, 2022.

[4] Zeinab Abdalla, Anneliese Andrews, and Kerstin Haring. Guidelines for combining regression testing approaches. In *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2022.

[5] Zeinab Abdalla, Kerstin Haring, and Anneliese Andrews. Test case prioritization for mobile apps. In *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2022.

[6] Hira Agrawal. Efficient coverage testing using global dominator graphs. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 11–20, 1999.

[7] Hiralal Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–34, 1994.

[8] Ahmed Alhaddad, Anneliese Andrews, and Zeinab Abdalla. Fsmapp: Testing mobile apps. In *Advancing in Computers*. Elsevier, 2021.

[9] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Gui crawling-based technique for android mobile application testing. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 252–261. IEEE, 2011.

[10] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A gui crawling-based technique for android mobile application testing. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 252–261, 2011.

[11] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. IEEE, 2012.

[12] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press., 32 Avenue of the Americas, New York, NY 10013, USA, first edition, 2008.

[13] Jeff Anderson, Saeed Salem, and Hyunsook Do. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 142–151, 2014.

[14] Anneliese Andrews, Seif Azghandi, and Orest Pilskalns. Regression testing of web applications using fsmweb. In *Proceedings of the International Conference on Software Engineering and Applications*, page 14, 2010.

[15] Anneliese Andrews and Hyunsook Do. Trade-off analysis for selective versus brute-force regression testing in fsmweb. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 184–192. IEEE, 2014.

[16] Anneliese A Andrews, Jeff Offutt, and Roger T Alexander. Testing web applications by modeling with fsms. *Software & Systems Modeling*, 4(3):326–345, 2005.

[17] Anneliese A Andrews, Jeff Offutt, Curtis Dyreson, Christopher J Mallery, Kshamta Jerath, and Roger Alexander. Scalability issues with using fsmweb to test web applications. *Information and Software Technology*, 52(1):52–66, 2010.

[18] B Athira and Philip Samuel. Web services regression test case prioritization. In *2010 International Conference on Computer Information Systems and Industrial Management Applications (CISIM)*, pages 438–443. IEEE, 2010.

[19] Young-Min Baek and Doo-Hwan Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 238–249, 2016.

[20] Fevzi Belli, Mubariz Eminov, and Nida Gökçe. Coverage-oriented, prioritized testing–a fuzzy clustering approach and case study. In *Latin-American Symposium on Dependable Computing*, pages 95–110. Springer, 2007.

[21] Fevzi Belli, Mubariz Eminov, and Nida Gokce. Model-based test prioritizing–a comparative soft-computing approach and case studies. In *Annual Conference on Artificial Intelligence*, pages 427–434. Springer, 2009.

[22] Garrett Birkhoff. *Lattice theory*, volume 25. American Mathematical Soc., 1940.

[23] Lionel C Briand, Yvan Labiche, Leeshawn O'Sullivan, and Michal M Sówka. Automated impact analysis of uml models. *Journal of Systems and Software*, 79(3):339–352, 2006.

[24] Lionel C Briand, Yvan Labiche, and George Soccar. Automating impact analysis and regression test selection based on uml designs. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 252–261. IEEE, 2002.

[25] Emanuela G Cartaxo, Patrícia DL Machado, and Francisco G Oliveira Neto. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability*, 21(2):75–100, 2011.

[26] Cagatay Catal and Deepti Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21(3):445–478, 2013.

[27] Nana Chang, Linzhang Wang, Yu Pei, Subrota K. Mondal, and Xuandong Li. Change-based test script maintenance for android apps. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 215–225, 2018.

[28] L. Chen, Z. Wang, L. Xu, H. Lu, and B. Xu. Test case prioritization for web service regression testing. In *2010 Fifth IEEE International Symposium on Service Oriented System Engineering*, pages 173–178, 2010.

[29] Lin Chen, Ziyuan Wang, Lei Xu, Hongmin Lu, and Baowen Xu. Test case prioritization for web service regression testing. In *2010 Fifth IEEE International Symposium on Service Oriented System Engineering*, pages 173–178. IEEE, 2010.

[30] Yanping Chen, Robert L Probert, and D Paul Sims. Specification-based regression test selection with risk analysis. In *Proceedings of the 2002 Conference of the Centre for Advanced Studies on Collaborative research*, page 1, 2002.

[31] Shauvik Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: are we there yet? In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 429–440. IEEE Press, 2015.

[32] Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[33] Ian G Clifton. *Android user interface design: Implementing Material Design for Developers*. Addison-Wesley Professional, 2015.

[34] SES Committee et al. Ieee standard for software maintenance. *IEEE Std*, pages 1219–1998, 1998.

[35] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms second edition. *The Knuth-Morris-Pratt Algorithm*, 2001.

[36] Pedro Costa, Ana CR Paiva, and Miguel Nabuco. Pattern based gui testing for mobile applications. In *2014 9th International Conference on the Quality of Information and Communications Technology*, pages 66–74. IEEE, 2014.

[37] Guilherme de Cleva Farto and Andre Takeshi Endo. Evaluating the model-based testing approach in the context of mobile applications. *Electronic Notes in Theoretical Computer Science*, 314:3–21, 2015.

[38] Figma Design. Figma: the collaborative interface design tool.(2017). *Retrieved September*, 17:2017, 2017.

[39] Arilo C Dias-Neto and Guilherme H Travassos. A picture from the model-based testing area: concepts, techniques, and challenges. In *Advances in Computers*, volume 80, pages 45–120. Elsevier, 2010.

[40] Hyunsook Do and Gregg Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 411–420. IEEE, 2005.

[41] Quan Do, Guowei Yang, Meiru Che, Darren Hui, and Jefferson Ridgeway. Regression test selection for android applications. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 27–28, 2016.

[42] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, 2004.

[43] Juhan Ernits, Rivo Roo, Jonathan Jacky, and Margus Veanes. Model-based testing of web applications using nmodel. In *Testing of Software and Communication Systems*, pages 211–216. Springer, 2009.

[44] Qurat-ul-ann Farooq, Muhammad Zohaib Z Iqbal, Zafar I Malik, and Aamer Nadeem. An approach for selective state machine based regression testing. In *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, pages 44–52, 2007.

[45] Elizabeta Fourneret, Jérôme Cantenot, Fabrice Bouquet, Bruno Legeard, and Julien Botella. Setgam: Generalized technique for regression testing based on uml/ocl models. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 147–156. IEEE, 2014.

[46] Boni García and Juan Carlos Dueñas. Automated functional testing based on the navigation of web applications. *arXiv preprint arXiv:1108.2357*, 2011.

[47] Deepak Garg and Amitava Datta. Parallel execution of prioritized test cases for regression testing of web applications. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference-Volume 135*, pages 61–68, 2013.

[48] Shankar Garg. Getting started with appium. In *Appium Recipes*, pages 1–18. Springer, 2016.

[49] Xiaobo Han, Hongwei Zeng, and Honghao Gao. A heuristic model-based test prioritization method for regression testing. In *2012 International Symposium on Computer, Consumer and Control*, pages 886–889. IEEE, 2012.

[50] M Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.

[51] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Reducing the cost of model-based testing through test case diversity. In *IFIP International Conference on Testing Software and Systems*, pages 63–78. Springer, 2010.

[52] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Empirical investigation of the effects of test suite properties on similarity-based test case selection. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 327–336. IEEE, 2011.

[53] Hadi Hemmati, Lionel Briand, Andrea Arcuri, and Shaukat Ali. An enhanced test case selection approach for model-based testing: an industrial case study. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, 2010.

[54] Shan-Shan Hou, Lu Zhang, Tao Xie, Hong Mei, and Jia-Su Sun. Applying interface-contract mutation in regression testing of component-based software. In *2007 IEEE International Conference on Software Maintenance*, pages 174–183. IEEE, 2007.

[55] Chin-Yu Huang, Jun-Ru Chang, and Yung-Hsin Chang. Design and analysis of gui test-case prioritization using weight-based methods. *Journal of Systems and Software*, 83(4):646–659, 2010.

[56] Sheng Huang, Yang Chen, Jun Zhu, Zhong Jie Li, and Hua Fang Tan. An optimized change-driven regression testing selection strategy for binary java

applications. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 558–565, 2009.

[57] Muhammad Zohaib Z Iqbal, Zafar I Malik, Matthias Riebisch, et al. A model-based regression testing approach for evolving software systems with flexible tool support. In *2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, pages 41–49. IEEE, 2010.

[58] Bo Jiang, Yu Wu, Yongfei Zhang, Zhenyu Zhang, and W.K. Chan. Retestdroid: Towards safer regression test selection for android application. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 235–244, 2018.

[59] Yiming Jing, Gail-Joon Ahn, and Hongxin Hu. Model-based conformance testing for android. In *International Workshop on Security*, pages 1–18. Springer, 2012.

[60] Bogdan Korel, George Koutsogiannakis, and Luay H Tahat. Model-based test prioritization heuristic methods and their evaluation. In *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, pages 34–43, 2007.

[61] Bogdan Korel, George Koutsogiannakis, and Luay H Tahat. Application of system models in regression test suite prioritization. In *2008 IEEE International Conference on Software Maintenance*, pages 247–256. IEEE, 2008.

[62] Bogdan Korel, Luay Ho Tahat, and Mark Harman. Test prioritization using system models. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 559–568. IEEE, 2005.

[63] Debasish Kundu, Monalisa Sarma, Debasis Samanta, and Rajib Mall. System testing for object-oriented systems with test case prioritization. *Software Testing, Verification and Reliability*, 19(4):297–333, 2009.

[64] Remo Lachmann, Sascha Lity, Mustafa Al-Hajjaji, Franz Fürchtegott, and Ina Schaefer. Fine-grained test case prioritization for integration testing of delta-oriented software product lines. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*, pages 1–10, 2016.

[65] Nozdrina Larysa and Savka Marta. Design thinking approaches in it projects. In *CEUR Workshop Proceedings*, pages 45–47, 2019.

[66] Franck Lebeau, Bruno Legeard, Fabien Peureux, and Alexandre Vernotte. Model-based vulnerability testing for web applications. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 445–452. IEEE, 2013.

[67] David Leon and Andy Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pages 442–453. IEEE, 2003.

[68] Hareton KN Leung and Lee J White. A cost model to compare regression test strategies. In *ICSM*, volume 91, pages 201–208, 1991.

[69] Harry R Lewis. Michael r. πgarey and david s. johnson. computers and intractability. a guide to the theory of np-completeness. wh freeman and company, san francisco1979, x+ 338 pp. *The Journal of Symbolic Logic*, 48(2):498–500, 1983.

[70] Lu Lu, Yulong Hong, Ying Huang, Kai Su, and Yuping Yan. Activity page based functional test automation for android application. In *2012 Third World Congress on Software Engineering*, pages 37–40. IEEE, 2012.

[71] Pradeep Macharla. Working with appium. In *Android Continuous Integration*, pages 95–115. Springer, 2017.

[72] Naghmeh Mahmoodian, Rusli Abdullah, and Masrah Azrifah Azim Murad. Text-based classification incoming maintenance requests to maintenance type. In *2010 International Symposium on Information Technology*, volume 2, pages 693–697. IEEE, 2010.

[73] Nashat Mansour, Husam Takkoush, and Ali Nehme. Uml-based regression testing for oo software. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(1):51–68, 2011.

[74] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of ajax web applications. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 121–130. IEEE, 2008.

[75] Leonardo Mariani, Sofia Papagiannakis, and Mauro Pezze. Compatibility and regression testing of cots-component-based software. In *29th International Conference on Software Engineering (ICSE'07)*, pages 85–95. IEEE, 2007.

[76] Martina Marré and Antonia Bertolino. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, 29(11):974–984, 2003.

[77] Abel Méndez Porras, Christian Ulises Quesada López, and Marcelo Jenkins Coronas. Automated testing of mobile applications: A systematic map and review. 2015.

[78] Huai-Kou Miao, Sheng-Bo Chen, and Hong-Wei Zeng. Model-based testing for web applications. *Jisuanji Xuebao(Chinese Journal of Computers)*, 34(6):1012–1028, 2011.

[79] Bernard Monjardet. The presence of lattice theory in discrete problems of mathematical social sciences. why. *Mathematical Social Sciences*, 46(2):103–144, 2003.

[80] Andrey Morozov, Kai Ding, Tao Chen, and Klaus Janschek. Test suite prioritization for efficient regression testing of model-based automotive software. In *2017 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 20–29. IEEE, 2017.

[81] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. Software testing of mobile applications: Challenges and future research directions. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 29–35. IEEE, 2012.

[82] Miguel Nabuco, Ana CR Paiva, Rui Camacho, and João Pascoal Faria. Inferring ui patterns with inductive logic programming. In *2013 8th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–5. IEEE, 2013.

[83] Cu D Nguyen, Alessandro Marchetto, and Paolo Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 100–110, 2012.

[84] Duc Hoai Nguyen, Paul Strooper, and Jörn Guy Süß. Automated functionality testing through guis. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science-Volume 102*, pages 153–162. 2010.

[85] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[86] Alessandro Orso, Hyunsook Do, Gregg Rothermel, Mary Jean Harrold, and David S Rosenblum. Using component metadata to regression test component-based software. *Software Testing, Verification and Reliability*, 17(2):61–94, 2007.

[87] Tuomas Pajunen, Tommi Takala, and Mika Katara. Model-based testing with a general purpose keyword-driven test automation framework. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 242–251. IEEE, 2011.

[88] Anjaneyulu Pasala, YLH Lew Yaw Fung, Fady Akladios, G Appala Raju, and Ravi Prakash Gorthi. Selection of regression test suite to validate software applications upon deployment of upgrades. In *19th Australian Conference on Software Engineering (aswec 2008)*, pages 130–138. IEEE, 2008.

[89] Thomas M Pigoski. *Practical Software Maintenance: Best Practices for Managing your Software Investment.* Wiley Publishing, 1996.

[90] Lihua Ran, Curtis Dyreson, Anneliese Andrews, Renée Bryce, and Christopher Mallery. Building test cases and oracles to automate the testing of web database applications. *Information and Software Technology*, 51(2):460–477, 2009.

[91] Margaret P Rayman. Selenium and human health. *The Lancet*, 379(9822):1256–1268, 2012.

[92] Hassan Reza, Kirk Ogaard, and Amarnath Malge. A model based testing technique to test web applications using statecharts. In *Fifth International Conference on Information Technology: New Generations (itng 2008)*, pages 183–188. IEEE, 2008.

[93] Kenneth H Rosen and Kamala Krithivasan. *Discrete Mathematics and Its Applications: With Combinatorics and Graph Theory*. Tata McGraw-Hill Education, 2012.

[94] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[95] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[96] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 179–188. IEEE, 1999.

[97] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[98] Mehmet Sahinoglu, Koray Incki, and Mehmet S Aktas. Mobile application verification: a systematic mapping study. In *International Conference on Computational Science and Its Applications*, pages 147–163. Springer, 2015.

[99] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 132–141. IEEE, 2004.

[100] PG Sapna and Hrushikesha Mohanty. Prioritization of scenarios based on uml activity diagrams. In *2009 First International Conference on Computational Intelligence, Communication Systems and Networks*, pages 271–276. IEEE, 2009.

[101] PG Sapna and Hrusikesha Mohanty. Prioritizing use cases to aid ordering of scenarios. In *2009 Third UKSim European Symposium on Computer Modeling and Simulation*, pages 136–141. IEEE, 2009.

[102] Michael Siff and Thomas Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25(6):749–768, 1999.

[103] Roberto S Silva Filho, Christof J Budnik, William M Hasling, Monica McKenna, and Rajesh Subramanyan. Supporting concern-based regression testing and prioritization in a model-driven environment. In *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, pages 323–328. IEEE, 2010.

[104] Shiwangi Singh, Rucha Gadgil, and Ayushi Chudgor. Automated testing of mobile applications using scripting technique: A study on appium. *Interna-*

tional Journal of Current Engineering and Technology (IJCET), 4(5):3627–3630, 2014.

[105] Hema Srikanth, Mikaela Cashman, and Myra B Cohen. Test case prioritization of build acceptance tests for an enterprise cloud application: An industrial case study. *Journal of Systems and Software*, 119:122–135, 2016.

[106] Praveen Ranjan Srivastava, Mahesh Ray, Julian Dermoudy, Byeong-Ho Kang, and Tai-hoon Kim. Test case minimization and prioritization using cmimx technique. In *International Conference on Advanced Software Engineering and Its Applications*, pages 25–33. Springer, 2009.

[107] Shiming Sun, Xiuping Hou, Can Gao, and Linlin Sun. Research on optimization scheme of regression testing. In *2013 Ninth International Conference on Natural Computation (ICNC)*, pages 1628–1632. IEEE, 2013.

[108] Tommi Takala, Mika Katara, and Julian Harty. Experiences of system-level model-based gui testing of an android application. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 377–386. IEEE, 2011.

[109] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGSOFT Software Engineering Notes*, 31(1):35–42, 2005.

[110] Wei-Tek Tsai, Xinyu Zhou, Raymond A Paul, Yinong Chen, and Xiaoying Bai. A coverage relationship model for test case selection and ranking for multi-version software. In *High Assurance Services Computing*, pages 285–311. Springer, 2009.

[111] Inayat ur Rehman, Saif Ur Rehman Malik, et al. The impact of test case reduction and prioritization on software testing effectiveness. In *2009 International Conference on Emerging Technologies*, pages 416–421. IEEE, 2009.

[112] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[113] Anneliese von Mayrhauser and Ning Zhang. Automated regression testing using dbt and sleuth. *Journal of Software Maintenance: Research and Practice*, 11(2):93–116, 1999.

[114] Anthony I Wasserman. Software engineering issues for mobile application development. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 397–400, 2010.

[115] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[116] W Eric Wong, Joseph R Horgan, Saul London, and Aditya P Mathur. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience*, 28(4):347–369, 1998.

[117] Samer Zein, Norsaremah Salleh, and John Grundy. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software*, 117:334–356, 2016.

[118] Ke Zhai, Bo Jiang, and WK Chan. Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study. *IEEE Transactions on Services Computing*, 7(1):54–67, 2012.

[119] Lingming Zhang, Ji Zhou, Dan Hao, Lu Zhang, and Hong Mei. Jtop: Managing junit test cases in absence of coverage information. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 677–679. IEEE, 2009.

[120] Lingming Zhang, Ji Zhou, Dan Hao, Lu Zhang, and Hong Mei. Prioritizing junit test cases in absence of coverage information. In *2009 IEEE International Conference on Software Maintenance*, pages 19–28. IEEE, 2009.

Appendix A

Components of Android Apps

Table A.1 summarizes the components for Android Apps [8]. LAPs are at the lowest model level. They can be an input type as defined in Table A.1 or a component. For example, "card" is a component. A component can contain another component. Column 1 shows the number of components. The (c) mark means that the component can contain another component. Column 2 of Table A.1 shows the name of the component. Column 3 shows the interface controls (i.e. the input types for the mobile application or for the mobile component). Column 4 shows the input constraints and transition information. Column 5 shows the effect of executing the component, when entering inputs that satisfy the input constraint. The last column represents the types of interface control: Text and Non-Text.

Table (A.1) Components of Mobile Application (LAPs)

No	Components	Interface Controls	Actions	Effect	Input Type
1	Bottom sheets	A. Button B. Link	A. R(Button, click) B. R(Button = X , click)	Close	Non-Text
2	Buttons	A. Floating action button B. Raised button C. Flat button	A. R(Content, click) B. R(Content, click) C. R(Button, click) D. R(Button, click)	Search Save Show list or select button	Non-Text
3 (c)	Cards	A. Image B. Video C. Textbox D. Text Area E. Button F. Links	A. R(Image, click) B. R(Image, click) C. R(Video, click) D. R(Textbox) E. R(Text Area) F. R(Button, click) G. R(Link, click)	Display Change size Run Display text Display many lines of text Show other website or page	Text
4 (c)	Chips	A. Textbox	A. R(Enter text)	Save	Text
					Continued on next page

Table A.1 – continued from previous page

No	Components	Interface Controls	Actions	Effect	Input Type
		B. Cards	B. R(Display content, choose)	Show details	
			C. R(Click on chip)	Display card	
5 (c)	Data Tables	A. Checkbox	A. R(Select, select dialog, add content)	Save	Text
		B. Link	B. R(Select, click button)	Delete	Non-Text
		C. Textbox	C. R(Click link)	Transfer	
		D. Menu	D. R(Select)	Show card	
		E. Button			
		F. Card			
6 (c)	Dialogs	A. Button	A. R(Show warning, click close)		Text
		B. Textbox	B. R(Select dialog, input content)	Save	Non-Text
		C. Date picker	C. R(Select dialog, click date picker, choose date, close)	Save	
		D. Checkbox	D. R(Select dialog, click Time picker, choose date, close)	Save	
		E. Time picker	E. R(Click menu, choose from list)	Close	
		F. Radio Box			
		G. Menu			

Continued on next page

Table A.1 – continued from previous page

No	Components	Interface Controls	Actions	Effect	Input Type
		H. Bar slide input			
7	Dividers	A. Images	A. R(Show Divider) B. R(Show Divider)	Show images Show content	Text
8	Grid lists	A. Images B. Text	A. R(Select image, zoom in) B. R(Select grid list, scrolling) C. R(Select title, sort)	Show images list Show text list Sort text	Text
9	Lists	A. images B. Text	A. R(Select title, sort)	Show title list Sort title list	Text
10 (c)	Menus	A. Button B. Text C. Combobox D. Checkbox E. Switch F. Reorder G. Expand/collapse H. Leave-behinds	A. R(Select text, copy) B. R(Select combobox, choose content) C. R(Select text, write content)	Show list in a menus Links of button to another pages	Text Non-Text

Continued on next page

Table A.1 – continued from previous page

No	Components	Interface Controls	Actions	Effect	Input Type
11 (c)	Pickers	A. Dialog	A. R(Select dialog, choose info) B. R(Select dialog, cancel)	Save	Non-Text
12	Progress & activity	A. Button	A. R(Click button)	Loading	Non-Text
13	Selection controls	A. Checkbox B. Radio Buttons C. On/Off switches	A. R(Click checkbox, change behavior of page) B. R(Click radio button, change behavior of page) C. R(Change switch, change behavior of page)	Change the behavior of the page	Non-Text
14	Sliders	A Slide bar	A. R(Change slider, effect on page)	Insert input Change behavior of the page	Non-Text
15	Snackbars & toasts	A. Button B. Text	A. R(Click button, display change) B. R(Show text)	Dismiss or cancel the action	Text Non-Text

Continued on next page

Table A.1 – continued from previous page

No	Components	Interface Controls	Actions	Effect	Input Type
16	Subheaders	A. Button B. Text	A. R(Click button, change page)	Filtering or sorting the content	Text
17	Steppers	A. Button	A. R(Click button = next, next step) B. R(Click button = previous, previous step) C. R(Click button = cancel, cancel process)	Show feedback of the process	Non-Text
18	Tabs	A. Dropdown Menu B. Text label	A. R(Select tab) B. R(Select tab)	Show content Show drop menu	Text
19	Toolbars	A. Button	A. R(Click toolbars, display list, click button)	Display the list	Non-Text
20	Tooltips	A. Images	A. R(Hover images)	Show text	Text
21	Text fields	A. Single-line text field B. Floating Label C. Multi-line text field	A. R(Content, click)	Save the content	Text

Continued on next page

Table A.1 – continued from previous page

No	Components	Interface Controls	Actions	Effect	Input Type
		D. Full-width field text field with Character counter E. Multi-line with character counter F. Full-width text field with character counter G. Auto-complete text field H. Inset auto-complete I. Full-width inline auto-complete J. In-line auto-complete			

Appendix B

DiscoverU Case Study Screens

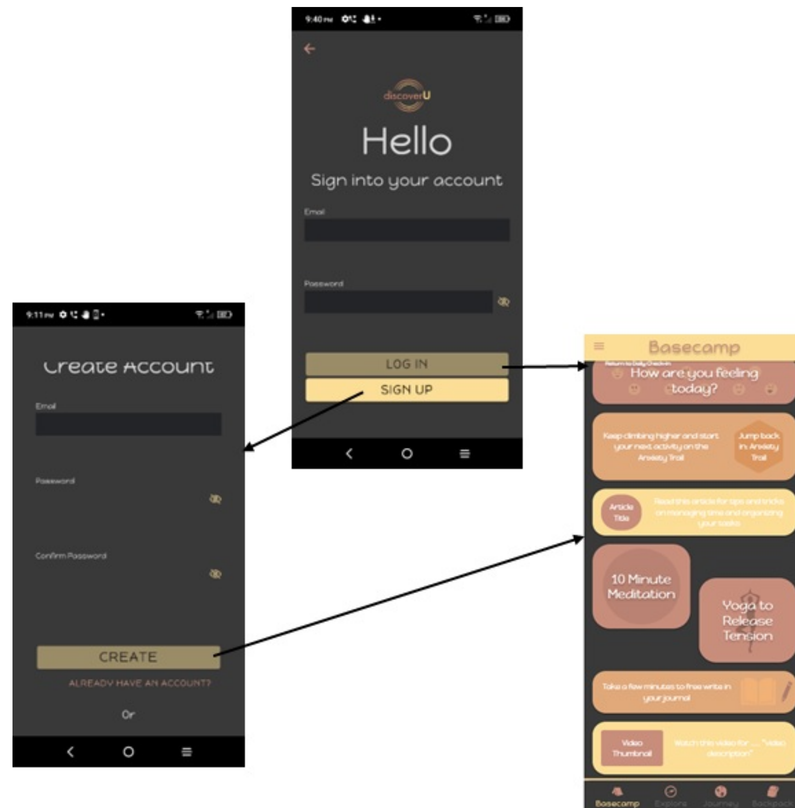


Figure (B.1) Entry Page for DiscoverU App

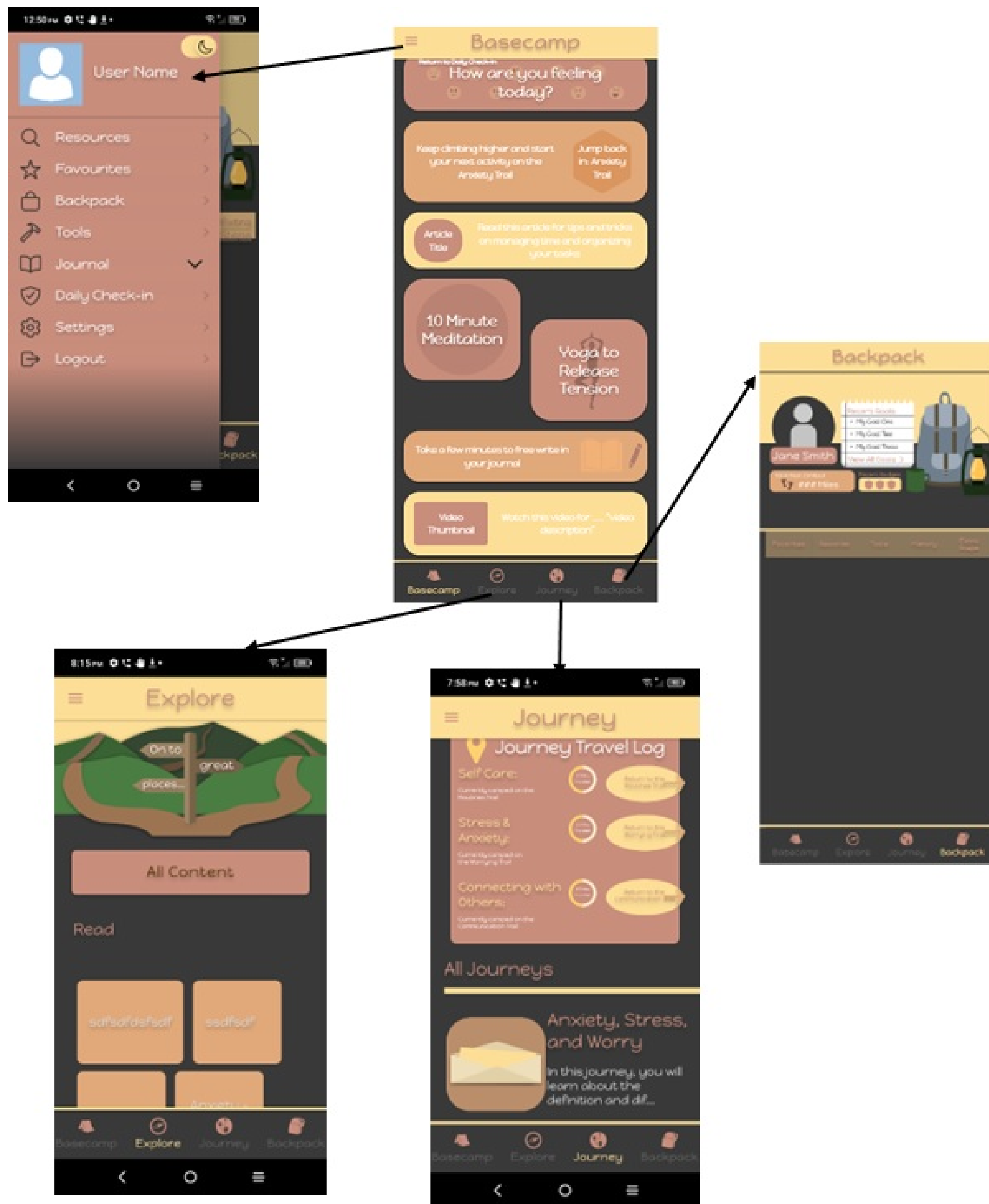


Figure (B.2)   Main Screens for the DiscoverU App

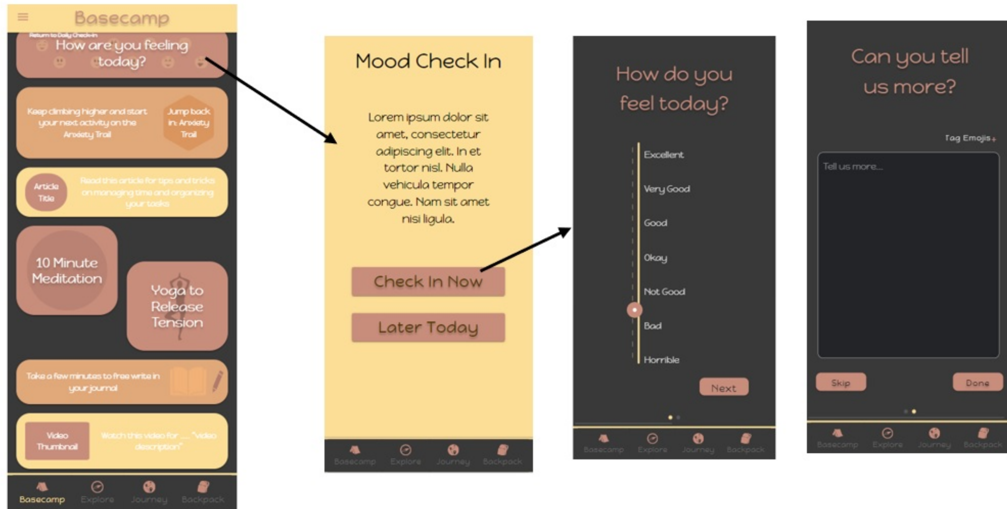


Figure (B.3) How are you feeling Screens for DiscoverU App

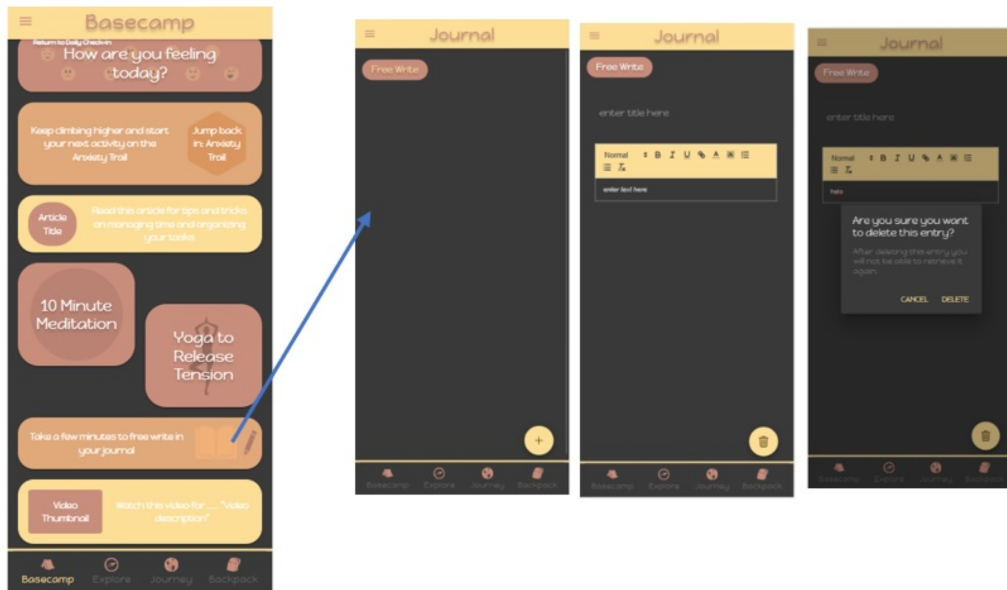


Figure (B.4) Free Write Screens for DiscoverU App

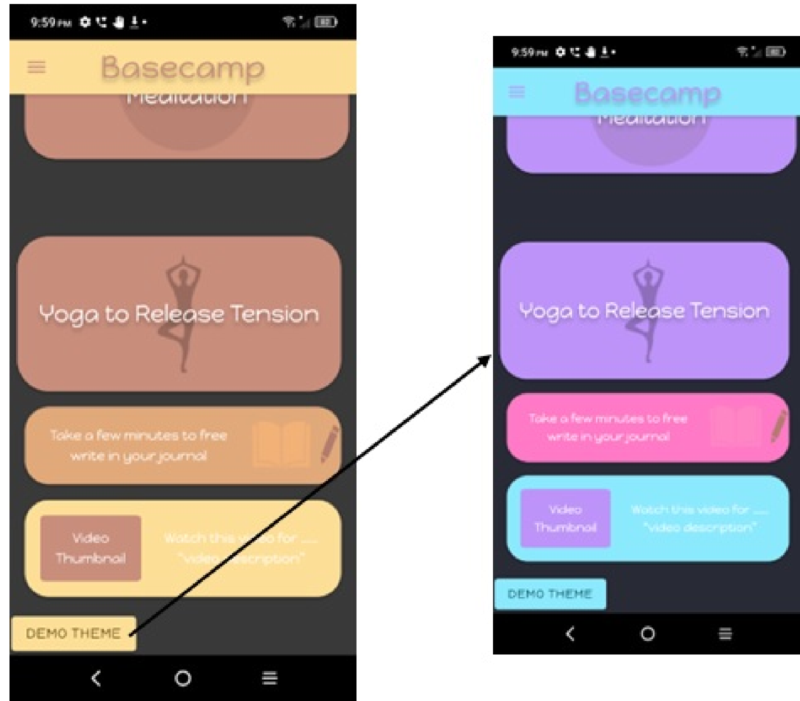


Figure (B.5) Demo Theme for DiscoverU App

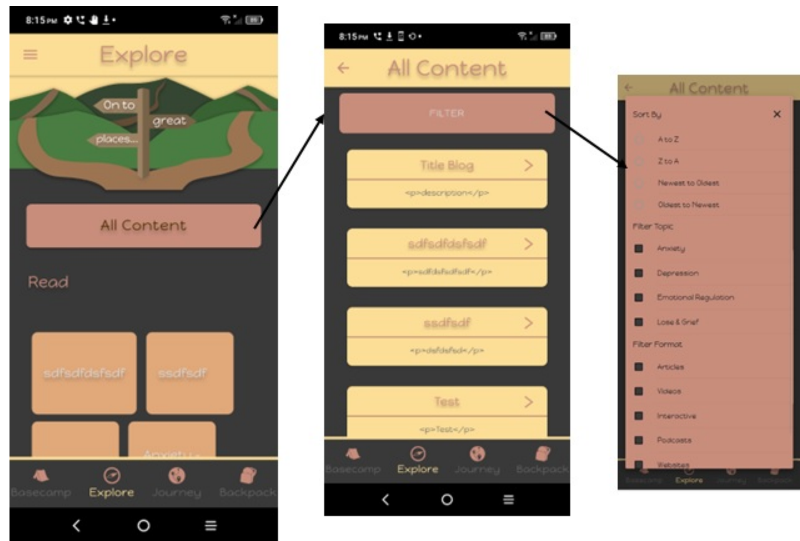


Figure (B.6) Explore Screens for DiscoverU App

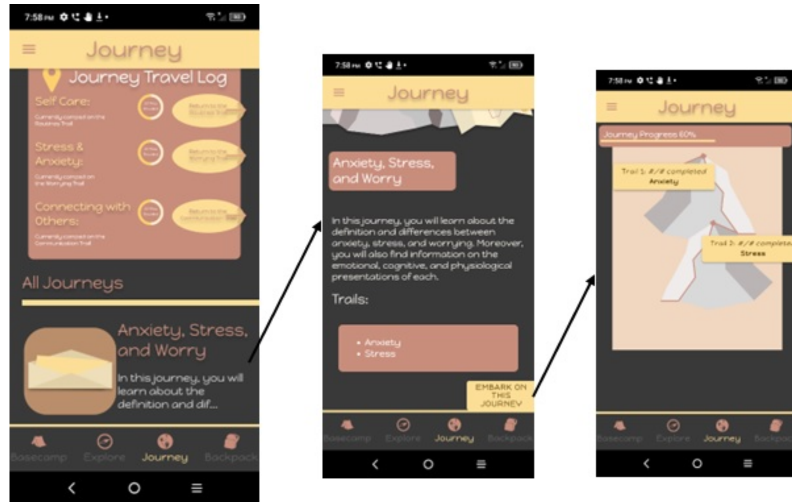


Figure (B.7) Journey Screens for DiscoverU App

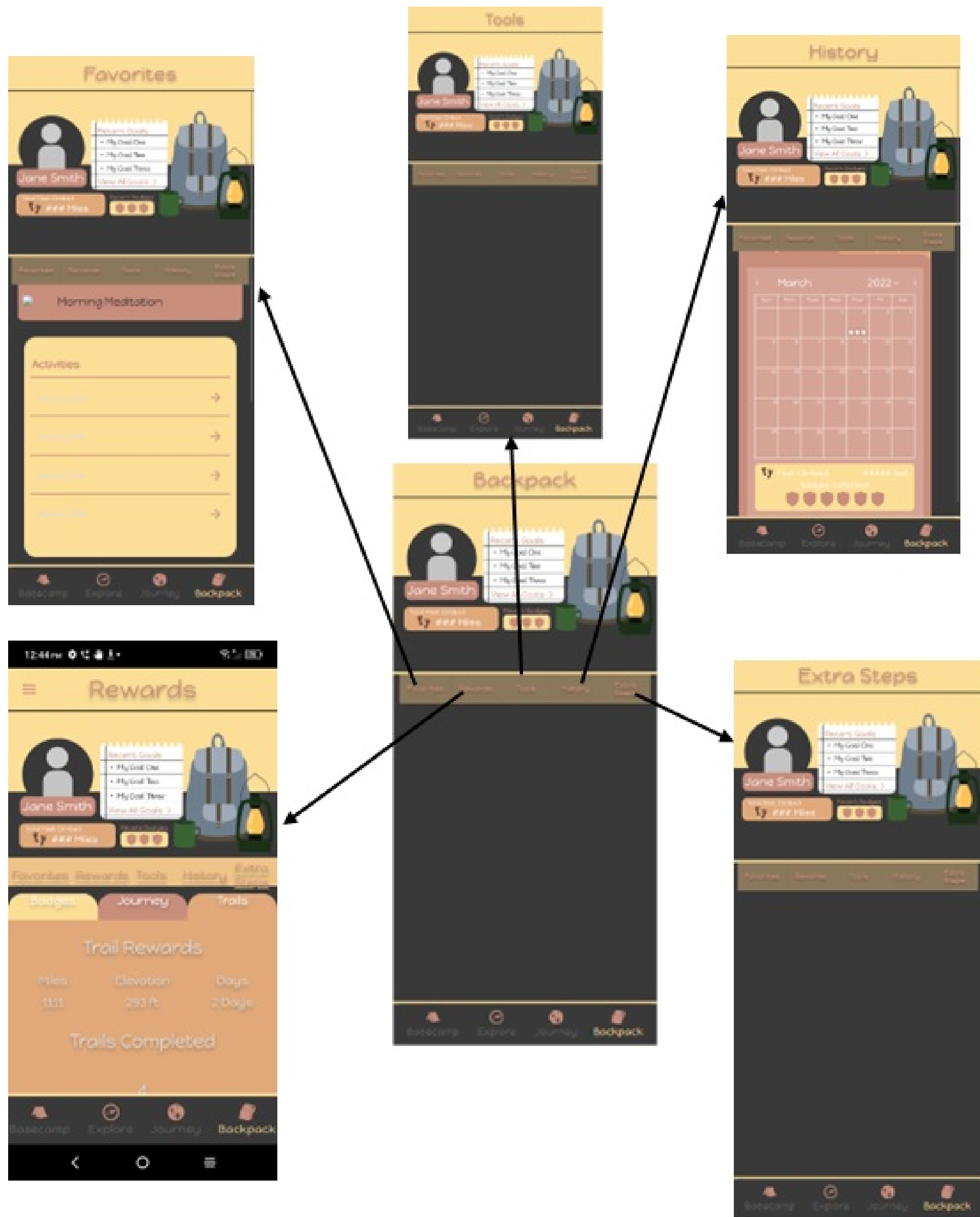


Figure (B.8) Backpack Screens for DiscoverU App

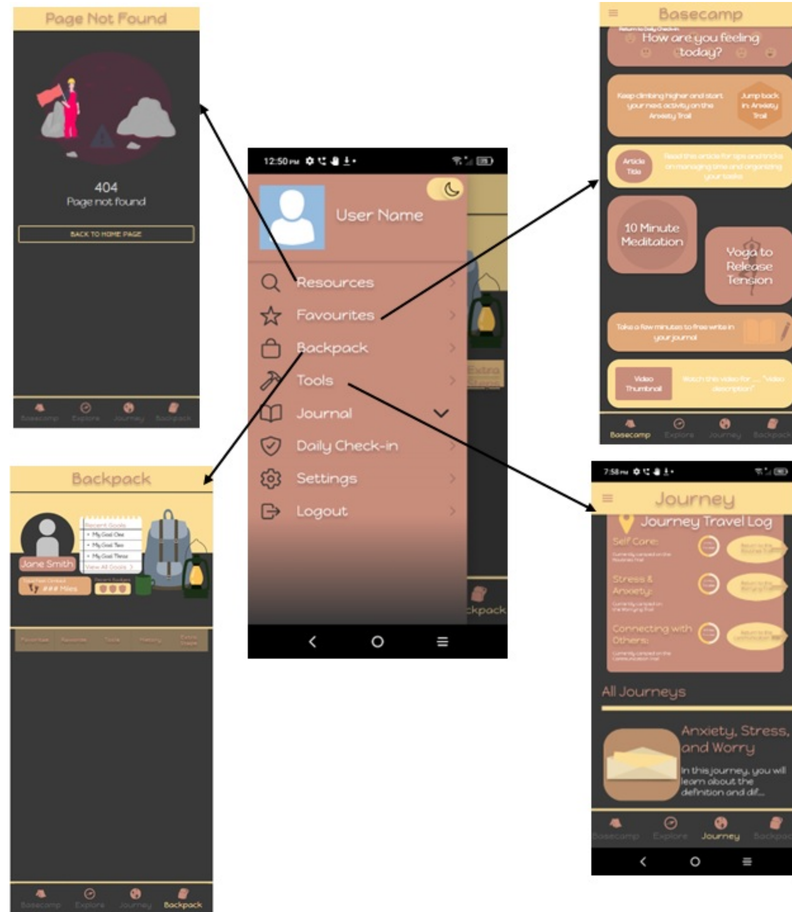


Figure (B.9) Menu Main Screens for DiscoverU App

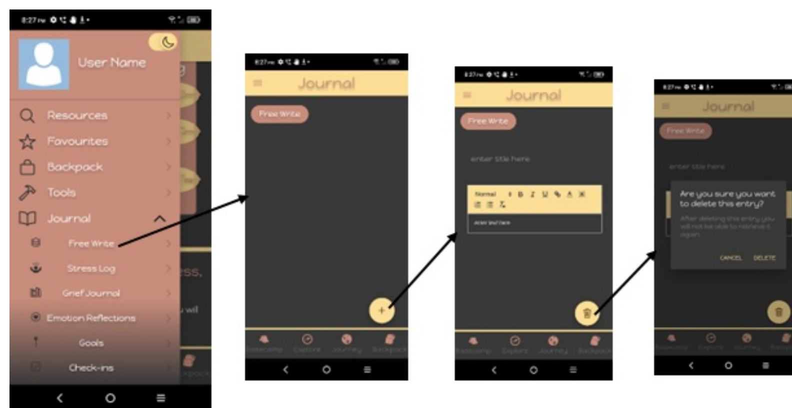


Figure (B.10) Sub-Menu (Free Write) Screens for DiscoverU App

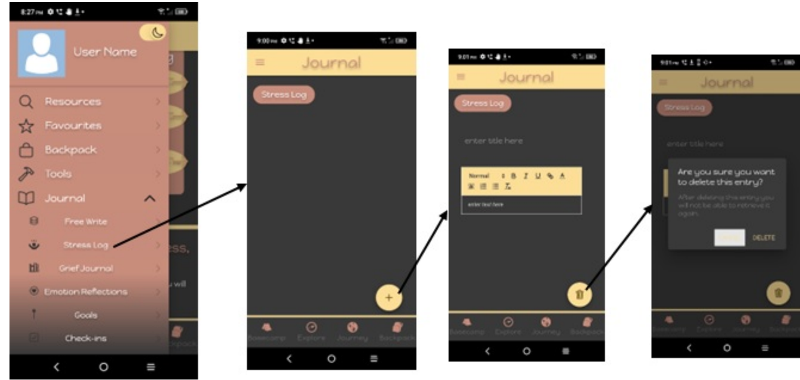


Figure (B.11) Sub-Menu (Stress Log) Screens for DiscoverU App

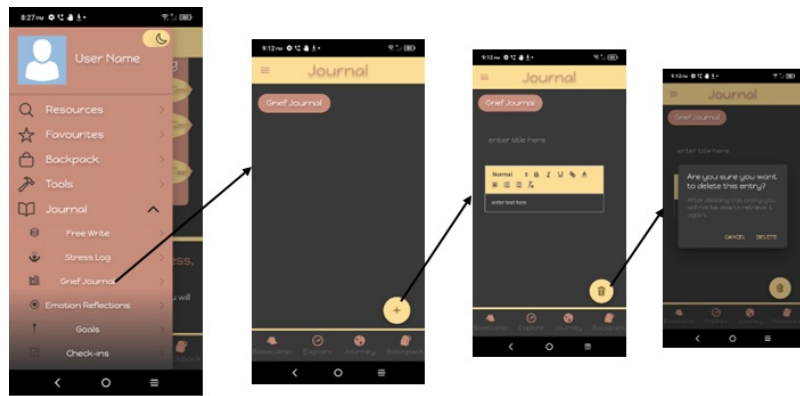


Figure (B.12) Sub-Menu (Grief Journal) Screens for DiscoverU App

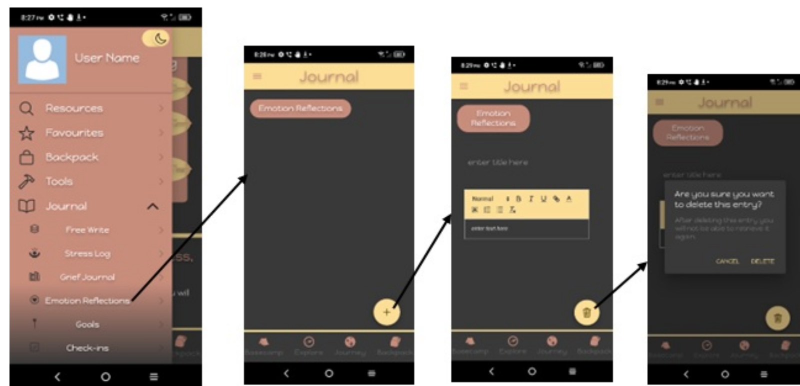


Figure (B.13) Sub-Menu (Emotion Reflections) Screens for DiscoverU App

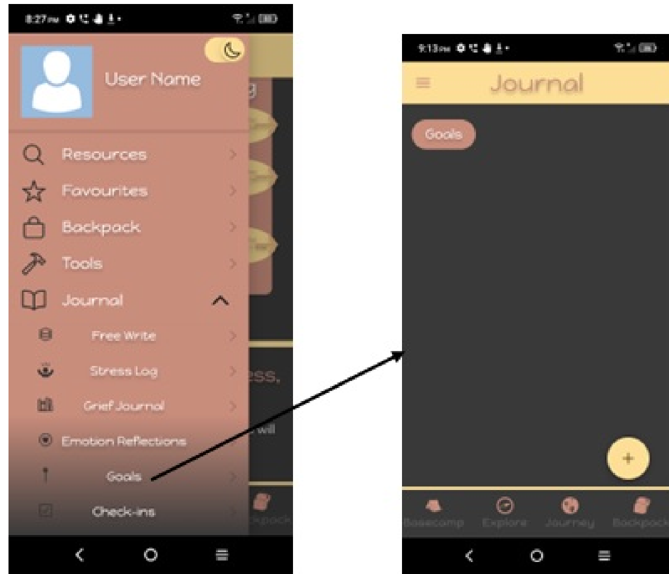


Figure (B.14) Sub-Menu (Goals) Screens for DiscoverU App

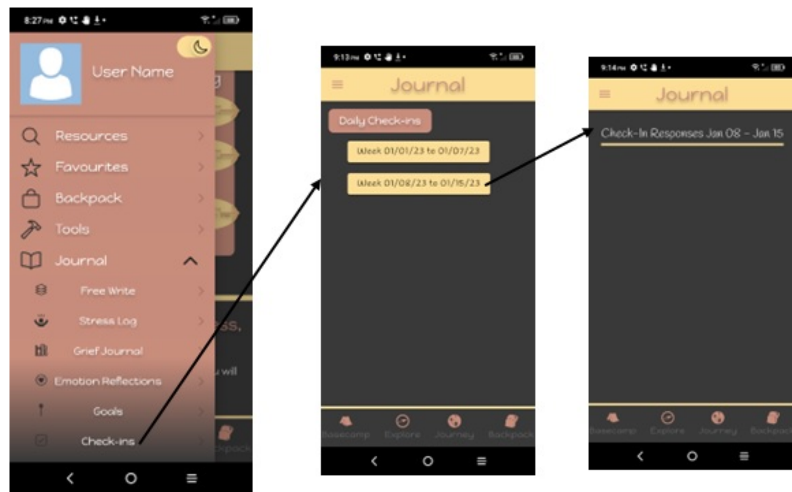


Figure (B.15) Sub-Menu (Check-ins) Screens for DiscoverU App

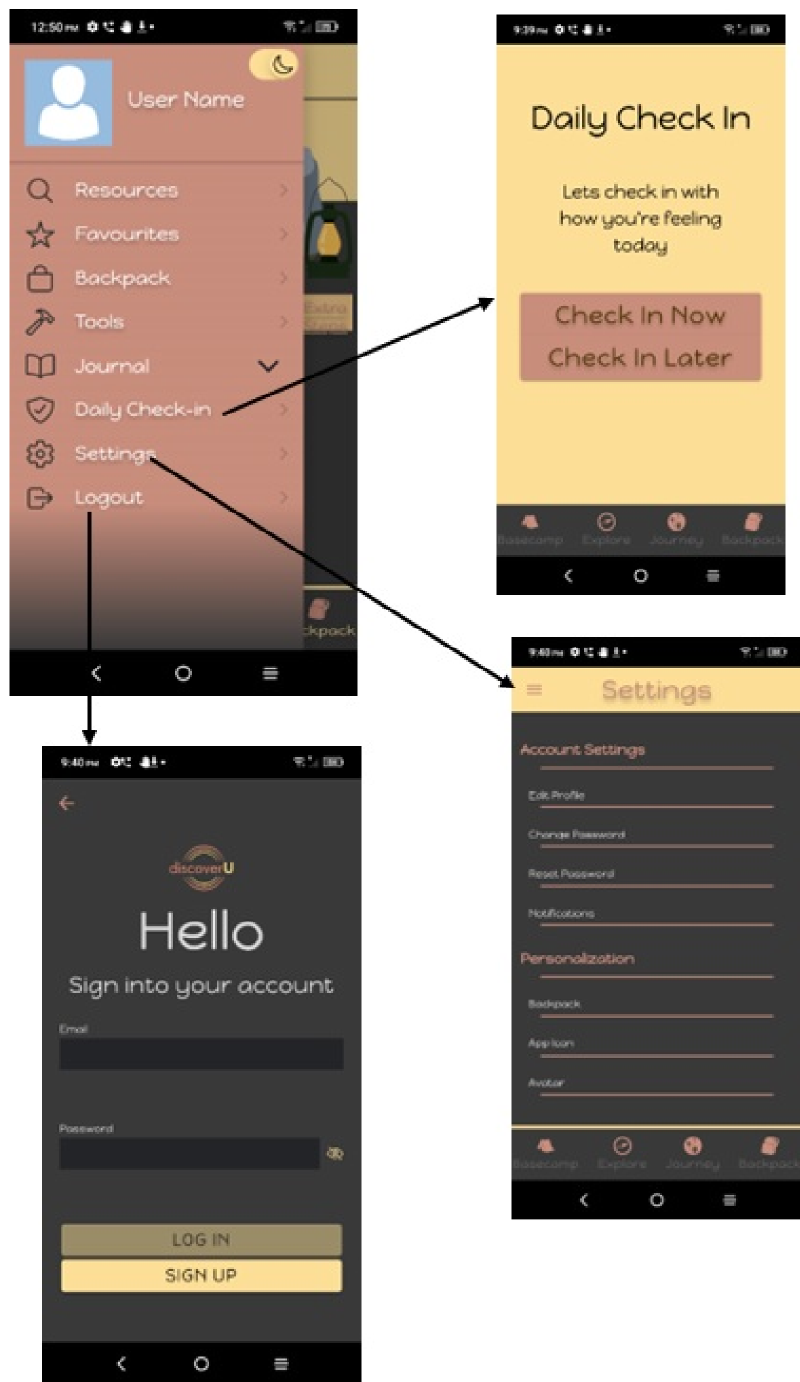


Figure (B.16) Menu (Daily Check-in, Settings, Logout) Screens for DiscoverU App

Appendix C

DiscoverU Mobile App

C.1 Clusters and Nodes

Table (C.1) Clusters and Nodes For **Main**

Node	Cluster/LAP	Explanation
Main	Main Page	Main Screen of the mobile app
Basecamp	Basecamp Page	Check in feeling, free write, demo theme
Explore	Explore Page	Explore the content and filter all content
Journey	Journey Page	Show all journeys to learn from them
Backpack	Backpack Page	Show the personal favorites, trails rewards, tools, history, and extra steps
Menu	Menu List	Show list of services that user can do
Exit	Exit app	Exit the mobile application

Table (C.2) Clusters and Nodes For **Basecamp**

Node	Cluster/LAP	Explanation
Basecamp	Basecamp Page	Connection from main page
How are feeling	Mood check in Information	Page related to feeling information
Free Write	Add Free Write Page	Page related to take minutes to free write in your journal
Demo Theme	Change Demo Theme	Change the app color
Main Exit	To main page	Return to main page of mobile application

Table (C.3) Clusters and Nodes For **Explore**

Node	Cluster/LAP	Explanation
Explore	Explore Page	Connection from main page
All Content	All Content Page	Page related to all activities that pushed by psychology team
Content	Content Page	Page related to types of activities
Cancel	Cancel	Cancel content
Main Exit	To main page	Return to main page of mobile application

Table (C.4) Clusters and Nodes For **Journey**

Node	Cluster/LAP	Explanation
Journey	Journey	Connection from main page
All Journeys	All Journeys Page	Page related to Journeys information
Main Exit	To main page	Return to main page of mobile application

Table (C.5) Clusters and Nodes For **Backpack**

Node	Cluster/LAP	Explanation
Backpack	Backpack Page	Connection from main page
Favorites	Favorites Page	Page related to personal Favorites information
Rewards	Rewards Page	Page related to personal Rewards information
Tools	Tools Page	Page related to personal Tools information
History	History Page	Page related to personal History Information
Extra Steps	Extra Steps Page	Page related to Extra Steps information
Main Exit	To main page	Return to main page of mobile application

Table (C.6) Clusters and Nodes For **Menu**

Node	Cluster/LAP	Explanation
Menu	Menu List	Connection from main page
Resources	Resources Page	Page related to Resource information
Favorites	Favorites Page	Page related to Favorites information
Backpack	Backpack Page	Page related to Backpack
Tools	Tools Page	Page related to Tools Information (Journeys)
Journal	Journal List	Page related to Sub-menu journal
Daily Check-in	Daily Check-in Page	Page related to feeling information
Settings	Settings Page	Page related to settings information
Logout	Logout Page	Logout the app and go to entry page
Main Exit	To main page	Return to main page of mobile application

Table (C.7) Clusters and Nodes For **How are feeling**

Node	Cluster/LAP	Explanation
How are feeling	How are feeling Page	Connection from main page
Check in Now	Check in Now Page	Page related to check in feeling information
Later Today	Check in Later	Return to main page of How are feeling
Main Exit	To main page	Return to main page of mobile application

Table (C.8) Clusters and Nodes For **Check in Now**

Node	Cluster/LAP	Explanation
Check in Now	Check in Now Page	Connection from main page
Next	Feeling information	Page related to check in feeling information
Main Exit	To main page	Return to main page of mobile application

Table (C.9) Clusters and Nodes For **Next**

Node	Cluster/LAP	Explanation
Next	feeling Page	Connection from main page
Done More about feeling	Add more information	Page related to tell more about feeling
Skip	Skip add more information	Skip add more information about feeling
Main Exit	To main page	Return to main page of mobile application

Table (C.10) Clusters and Nodes For **Free Write**

Node	Cluster/LAP	Explanation
Free Write	Free Write Page	Connection from main page
Add FW	Add Free Write Page	Page related to add free write in your journal
Delete FR	Delete Free Write	Page related to delete free write
Main Exit	To main page	Return to main page of mobile application

Table (C.11) Clusters and Nodes For **Add Free Write**

Node	Cluster/LAP	Explanation
Add FW	Add FW Page	Connection from main page
Adding FW	Add FW information	Submit free write information
Delete Adding	Cancel Adding FW information	Cancel free write information
Main Exit	To main page	Return to main page of mobile application

Table (C.12) Clusters and Nodes For **Delete**

Node	Cluster/LAP	Explanation
Delete	Delete Page	Connection from main page
Delete FW	Delete free write	Delete free write information
Cancel	Cancel Delete FW	Cancel delete free write information
Main Exit	To main page	Return to main page of mobile application

Table (C.13) Clusters and Nodes For **All Content**

Node	Cluster/LAP	Explanation
All Content	All Content Page	Connection from main page
Filter	All content to Filter	Page related to filter all content
Back	Back to previous page	Return to the previous page
Main Exit	To main page	Return to main page of mobile application

Table (C.14) Clusters and Nodes For **Filter**

Node	Cluster/LAP	Explanation
Filter	Filter Page	Connection from main page
Filtering	Filter information	Page related to filter all content information
Cancel	Cancel filter	Cancel filtering all content
Main Exit	To main page	Return to main page of mobile application

Table (C.15) Clusters and Nodes For **Journey**

Node	Cluster/LAP	Explanation
Journey	Journey Page	Connection from main page
A Journey	A Journey information	Page related to selected journey information
Main Exit	To main page	Return to main page of mobile application

Table (C.16) Clusters and Nodes For **All Journey**

Node	Cluster/LAP	Explanation
All Journey	All Journey Page	Connection from main page
Embark on this Journey	Trails information	Page related to trails information for this journey
Main Exit	To main page	Return to main page of mobile application

Table (C.17) Clusters and Nodes For **Rewords**

Node	Cluster/LAP	Explanation
Rewords	Rewords Page	Connection from main page
Badges	Badges information	Page related to Badges information
Journey	Journeys information	Page related to Journeys completed information
Trails	Trails information	Page related to Trails rewards and completed information
Main Exit	To main page	Return to main page of mobile application

Table (C.18) Clusters and Nodes For **Resources**

Node	Cluster/LAP	Explanation
Resources	Resources Page	Connection from main page
Back	Back Page	Back to home Page (Basecamp page)
Main Exit	To main page	Return to main page of mobile application

Table (C.19) Clusters and Nodes For **Menu-Favourites**

Node	Cluster/LAP	Explanation
Favourites	Favourites Page	Connection from main page
Basecamp	Basecamp Page	Check in feeling, free write, and demo theme
Main Exit	To main page	Return to main page of mobile application

Table (C.20) Clusters and Nodes For **Menu-favourites-Basecamp**

Node	Cluster/LAP	Explanation
Basecamp	Basecamp Page	Connection from main page
How Are You Feeling	How Are You Feeling Page	Page related to feeling Information
Free Write	Free Write Page	Page related to Free Write Information
Demo Theme	Demo Theme Page	Page related to change color the App
Main Exit	To main page	Return to main page of mobile application

Table (C.21) Clusters and Nodes For **Menu-Backpack**

Node	Cluster/LAP	Explanation
Menu-Backpack	Menu-Backpack Page	Connection from main page
Backpack	Backpack Page	Show the personal favorites, trails rewards, tools, history, and extra steps
Main Exit	To main page	Return to main page of mobile application

Table (C.22) Clusters and Nodes For **Menu-Backpack-Backpack**

Node	Cluster/LAP	Explanation
Backpack	Backpack Page	Connection from main page
Favourites	Favourites Page	Page related to Favourites Information
Rewards	Rewards Page	Page related to Rewards Information
Tools	Tools Page	Page related to tools information
History	History Page	Page related to History information
Extra Steps	Extra Steps Page	Page related to Extra Steps information
Main Exit	To main page	Return to main page of mobile application

Table (C.23) Clusters and Nodes For **Menu-Tools**

Node	Cluster/LAP	Explanation
Menu-Tools	Menu-Tools Page	Connection from main page
Journey	Journey Page	Show all journeys
Main Exit	To main page	Return to main page of mobile application

Table (C.24) Clusters and Nodes For **Menu-Tools-Journey**

Node	Cluster/LAP	Explanation
Journey	Journey Page	Connection from main page
All Journeys	All Journeys Page	Page related to all Journeys Information
Main Exit	To main page	Return to main page of mobile application

Table (C.25) Clusters and Nodes For **Menu-sub-menu**

Node	Cluster/LAP	Explanation
Sub-Menu-Journal	Sub-Menu-Journal Page	Connection from main page
Journal-List	Journal-List	Show list of services
Main Exit	To main page	Return to main page of mobile application

Table (C.26) Clusters and Nodes For **Sub-Menu-Journal-List**

Node	Cluster/LAP	Explanation
Journal-List	Journal-List	Connection from main page
Free Write	Free Write Page	Page related to Free Write Information
Stress Log	Stress Log Page	Page related to Stress Log Information
Grief Journal	Grief Journal Page	Page related to Grief Journal Information
Emotion Reflections	Emotion Reflections Page	Page related to Emotion Reflections Information
Goals	Goals Page	Page related to Goals Information
Check-ins	Check-ins Page	Page related to Check-ins Information
Main Exit	To main page	Return to main page of mobile application

Table (C.27) Clusters and Nodes For **Sub-Free Write**

Node	Cluster/LAP	Explanation
Free Write	Free Write Page	Connection from main page
Add FW	Add Free Write Page	Page related to add free write in your journal
Delete FR	Delete Free Write	Page related to delete free write
Main Exit	To main page	Return to main page of mobile application

Table (C.28) Clusters and Nodes For **sub-AddFreeW**

Node	Cluster/LAP	Explanation
Add FW	Add FW Page	Connection from main page
Adding FW	Add FW information	Submit free write information
Delete Adding	Cancel Adding FW information	Cancel free write information
Main Exit	To main page	Return to main page of mobile application

Table (C.29) Clusters and Nodes For **Delete Free Write**

Node	Cluster/LAP	Explanation
Delete	Delete Page	Connection from main page
Delete FW	Delete free write	Delete free write information
Cancel	Cancel Delete FW	Cancel delete free write information
Main Exit	To main page	Return to main page of mobile application

Table (C.30) Clusters and Nodes For **Stress Log**

Node	Cluster/LAP	Explanation
Stress Log	Stress Log Page	Connection from main page
Add Stress-Log	Add Stress-Log Page	Page related to add stress-log in your journal
Delete Stress-Log	Delete Stress-Log	Page related to delete Stress-log
Main Exit	To main page	Return to main page of mobile application

Table (C.31) Clusters and Nodes For **Add Stress-Log**

Node	Cluster/LAP	Explanation
Add Stress-Log	Add Stress-Log Page	Connection from main page
Adding Stress-Log	Add Stress-Log information	Submit stress-log information
Delete Adding	Cancel Adding stress-log information	Cancel stress-log information
Main Exit	To main page	Return to main page of mobile application

Table (C.32) Clusters and Nodes For **Delete Stress-Log**

Node	Cluster/LAP	Explanation
Delete	Delete Page	Connection from main page
Delete Stress-Log	Delete Stress-Log	Delete stress-log information
Cancel	Cancel Delete Stress-log	Cancel delete stress-log information
Main Exit	To main page	Return to main page of mobile application

Table (C.33) Clusters and Nodes For **Grief Journal**

Node	Cluster/LAP	Explanation
Grief Journal	Grief Journal Page	Connection from main page
Add Grief-Journal	Add Grief-Journal Page	Page related to add Grief-Journal in your journal
Delete Grief-Journal	Delete Grief-Journal	Page related to delete Grief-Journal
Main Exit	To main page	Return to main page of mobile application

Table (C.34) Clusters and Nodes For **Add Grief-Journal**

Node	Cluster/LAP	Explanation
Add Grief-Journal	Add Grief-Journal Page	Connection from main page
Adding Grief-Journal	Add Grief-Journal information	Submit Grief-Journal information
Delete Adding	Cancel Adding Grief-Journal information	Cancel Grief-Journal information
Main Exit	To main page	Return to main page of mobile application

Table (C.35) Clusters and Nodes For **Delete Grief-Journal**

Node	Cluster/LAP	Explanation
Delete	Delete Page	Connection from main page
Delete Grief-Journal	Delete Grief-Journal	Delete Grief-Journal information
Cancel	Cancel Delete Grief-Journal	Cancel delete Grief-Journal information
Main Exit	To main page	Return to main page of mobile application

Table (C.36) Clusters and Nodes For **Emotion Reflections**

Node	Cluster/LAP	Explanation
Emotion Reflections	Emotion Reflections Page	Connection from main page
Add Emotion-Reflections	Add Emotion-Reflection Page	Page related to add Emotion-Reflection in your journal
Delete Emotion-Reflection	Delete Emotion-Reflection	Page related to delete Emotion-Reflection
Main Exit	To main page	Return to main page of mobile application

Table (C.37) Clusters and Nodes For Add **Emotion-Reflection**

Node	Cluster/LAP	Explanation
Add Emotion-Reflection	Add Emotion-Reflection Page	Connection from main page
Adding Emotion-Reflection	Add Emotion-Reflection information	Submit Emotion-Reflection information
Delete Adding	Cancel Adding Emotion-Reflection information	Cancel Emotion-Reflection information
Main Exit	To main page	Return to main page of mobile application

Table (C.38) Clusters and Nodes For **Delete Emotion-Reflection**

Node	Cluster/LAP	Explanation
Delete	Delete Page	Connection from main page
Delete Emotion-Reflection	Delete Emotion-Reflection	Delete Emotion-Reflection information
Cancel	Cancel Delete Emotion-Reflection	Cancel delete Emotion-Reflection information
Main Exit	To main page	Return to main page of mobile application

Table (C.39) Clusters and Nodes For **Check-ins**

Node	Cluster/LAP	Explanation
Check-ins	Check-ins Page	Connection from main page
Daily-Check-ins	Daily-Check-ins Page	Page related to weekly Daily-Check-ins in your journal
Main Exit	To main page	Return to main page of mobile application

Table (C.40) Clusters and Nodes For **Week-Check-ins**

Node	Cluster/LAP	Explanation
Daily-Check-ins	Daily-Check-ins Page	Connection from main page
Week-Check-ins	Week-Check-ins Information	Page related to Check-ins responses to selected rang week in your journal
Main Exit	To main page	Return to main page of mobile application

Table (C.41) Clusters and Nodes For **Menu-Daily Check-in**

Node	Cluster/LAP	Explanation
Daily Check-in	Daily Check-in Page	Connection from main page
Mood Check-in	Mood Check-in Page	Page related to feeling Information
Main Exit	To main page	Return to main page of mobile application

Table (C.42) Clusters and Nodes For **Menu-Mood Check-in**

Node	Cluster/LAP	Explanation
Mood Check-in	Mood Check-in Page	Connection from main page
Check-in Now	Check-in Now Page	Page related to Check-in Now feeling Information
Later Today	Check-in Later Today	Return to Home page (Basecamp page)
Main Exit	To main page	Return to main page of mobile application

Table (C.43) Clusters and Nodes For **Menu-Settings**

Node	Cluster/LAP	Explanation
Menu-Settings	Menu-Settings Page	Connection from main page
Settings	Settings Page	Page related to Settings Information
Main Exit	To main page	Return to main page of mobile application

Table (C.44) Clusters and Nodes For **Menu-Logout**

Node	Cluster/LAP	Explanation
Menu-Logout	Menu-Logout Page	Connection from main page
Logout	Logout Page	Return to entry page of the app
Main Exit	To main page	Return to main page of mobile application

Appendix D

DiscoverU Mobile App FSM Model

D.1 DiscoverU Mobile App HFSM Model

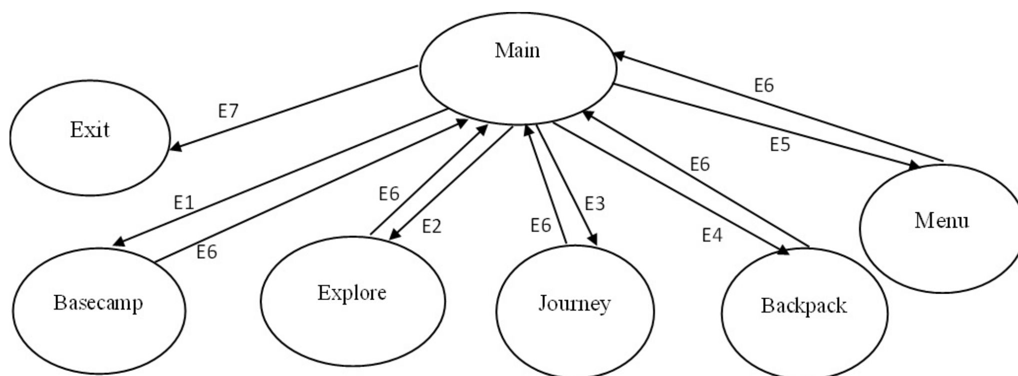


Figure (D.1) Main Cluster (AFSM) for DiscoverU App

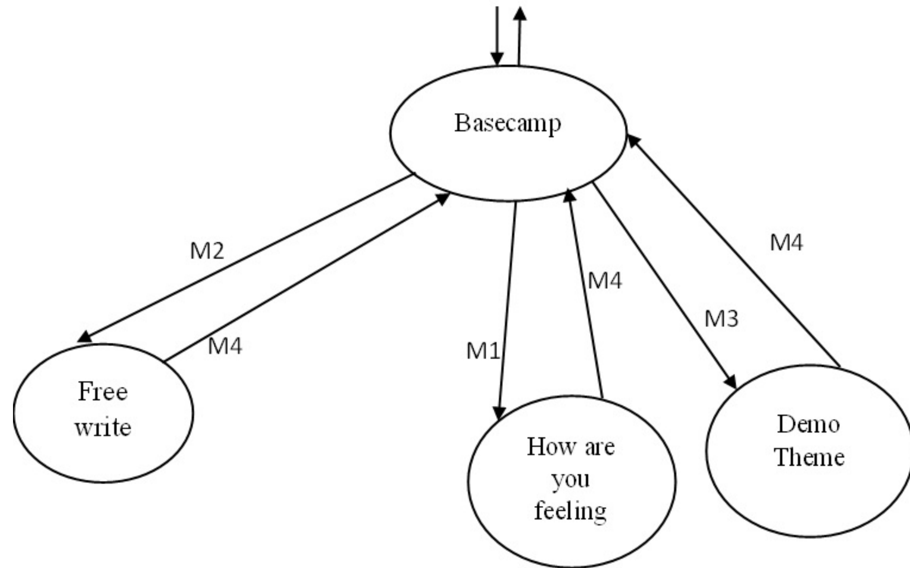


Figure (D.2) Basecamp Cluster for DiscoverU App

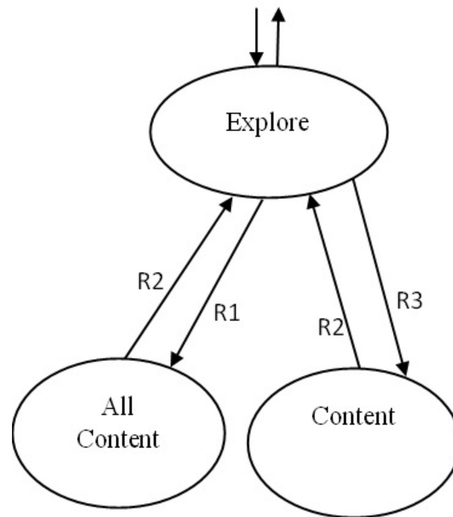


Figure (D.3) Explore Cluster for DiscoverU App

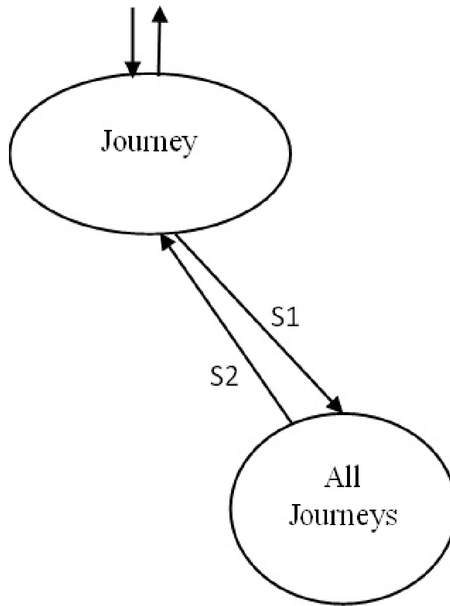


Figure (D.4) Journey Cluster for DiscoverU App

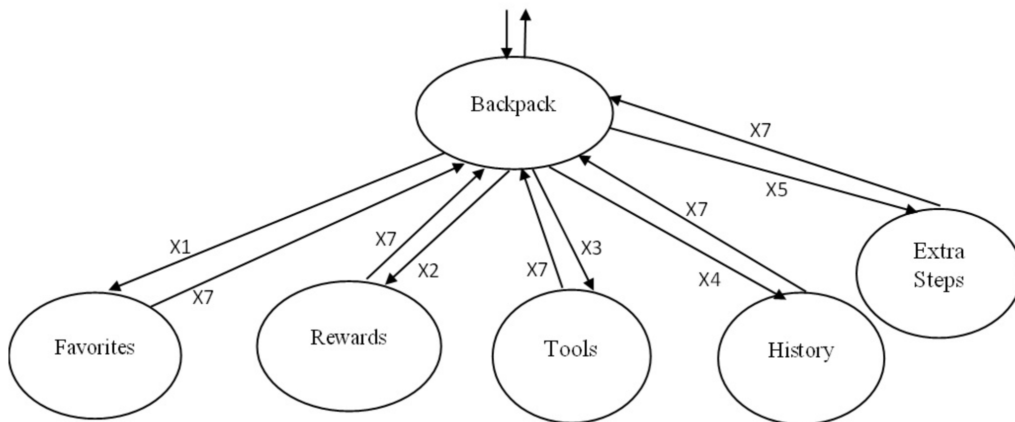


Figure (D.5) Backpack Cluster for DiscoverU App

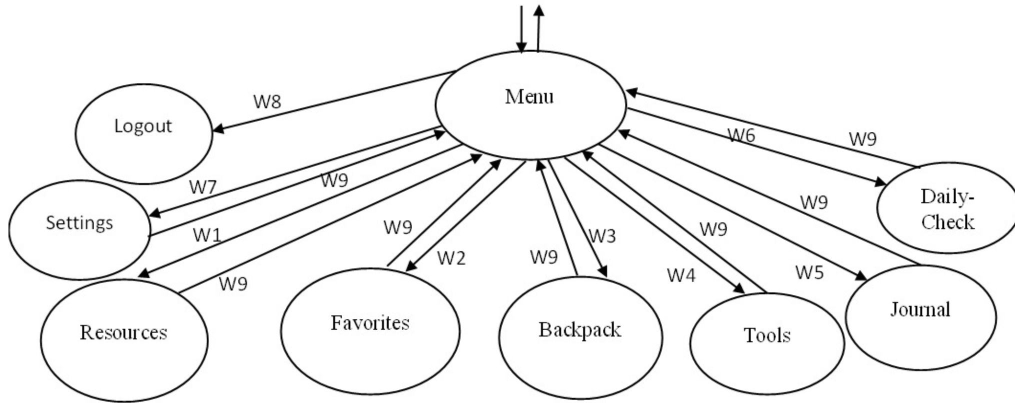


Figure (D.6) Menu Cluster for DiscoverU App

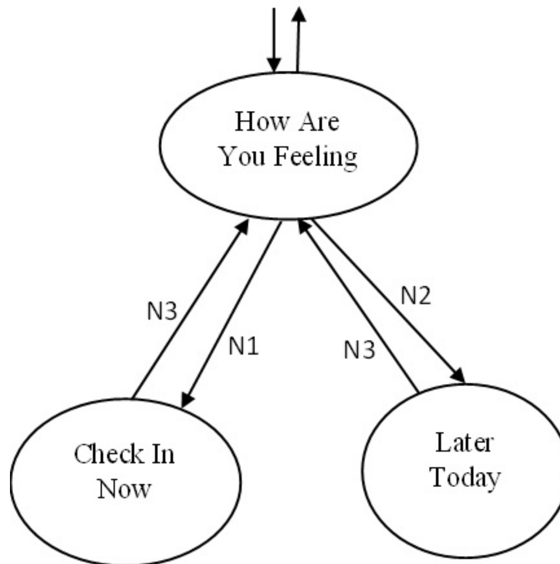


Figure (D.7) How Are You Feeling? Cluster for DiscoverU App

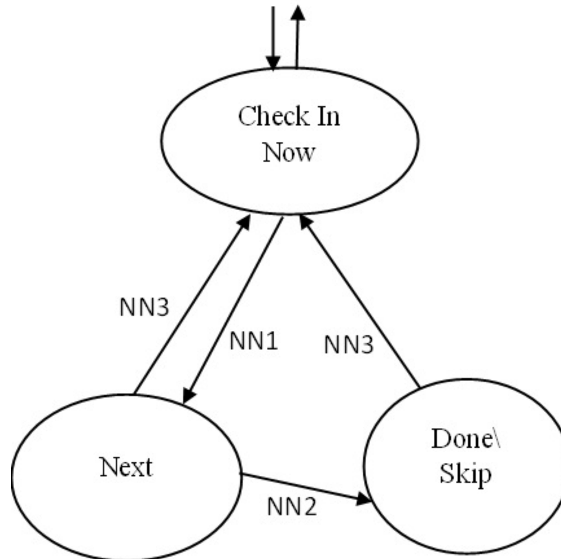


Figure (D.8) Check-in Now Cluster for DiscoverU App

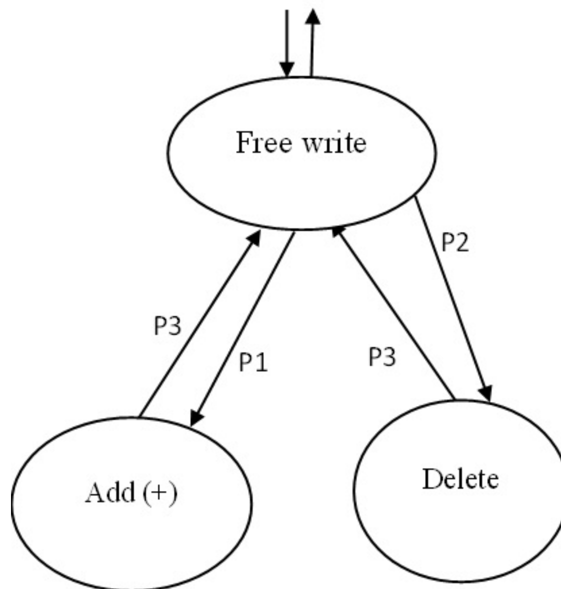


Figure (D.9) Free Write Cluster for DiscoverU App

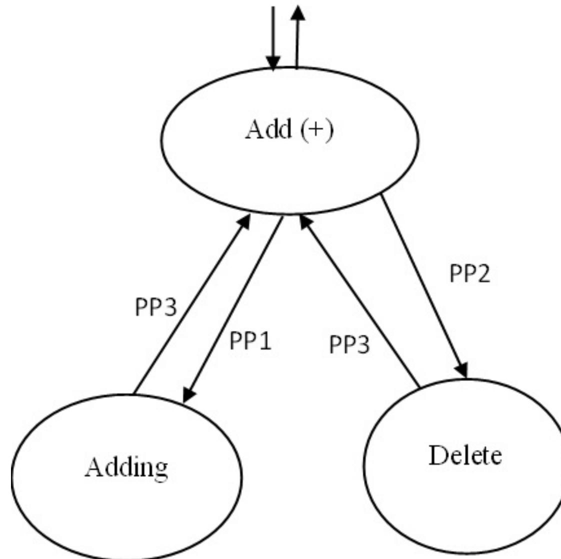


Figure (D.10) Add Free Write Cluster for DiscoverU App

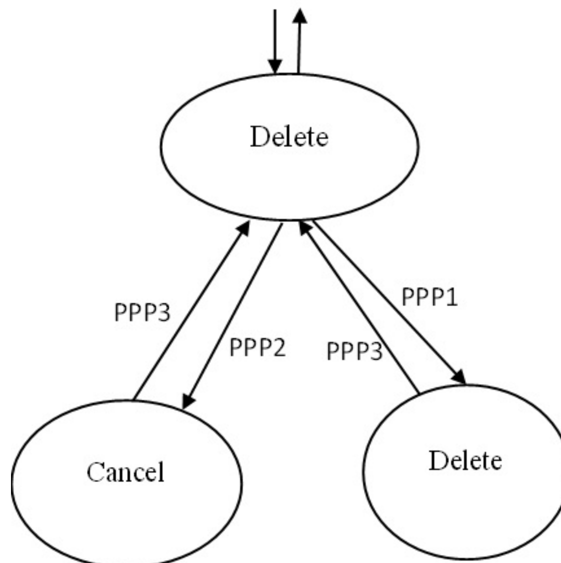


Figure (D.11) Delete Cluster for DiscoverU App

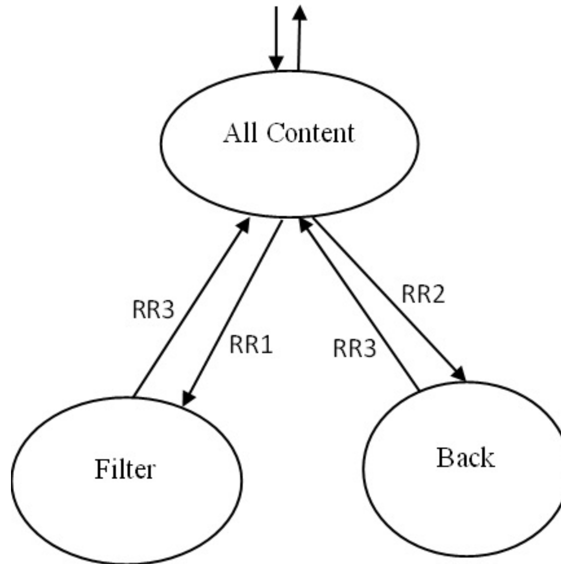


Figure (D.12) All Content Cluster for DiscoverU App

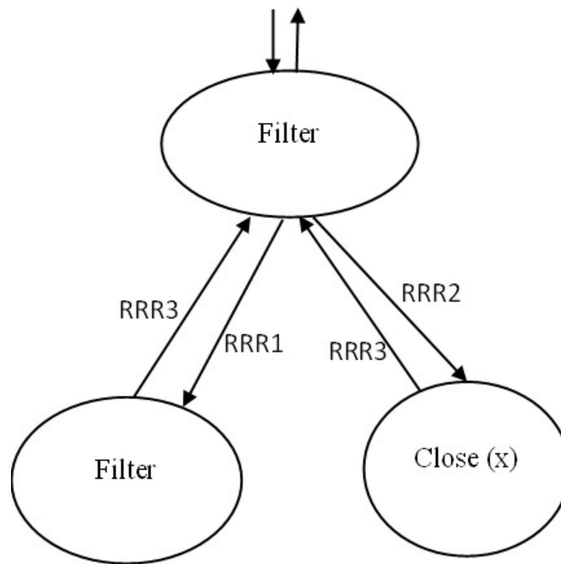


Figure (D.13) Filter Cluster for DiscoverU App

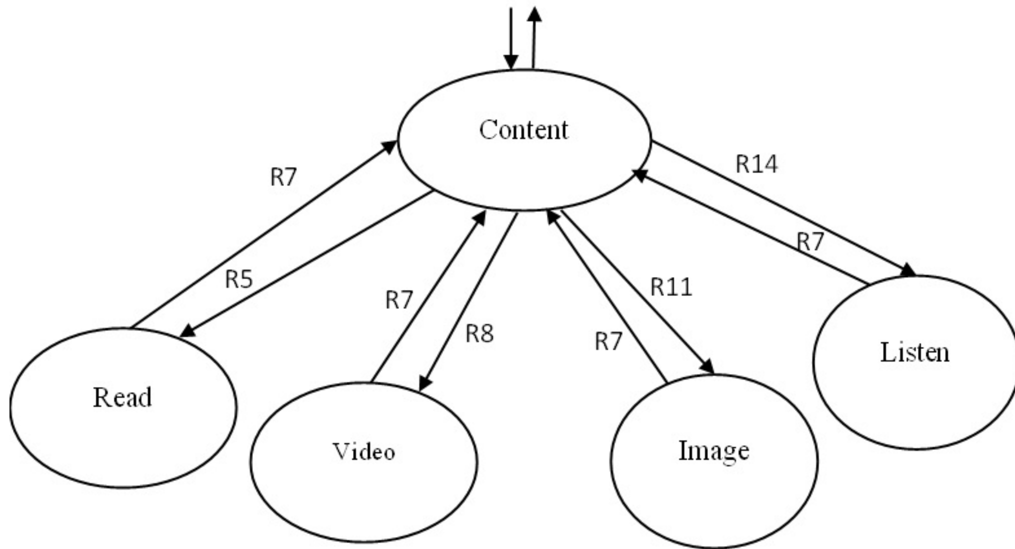


Figure (D.14) Content Cluster for DiscoverU App

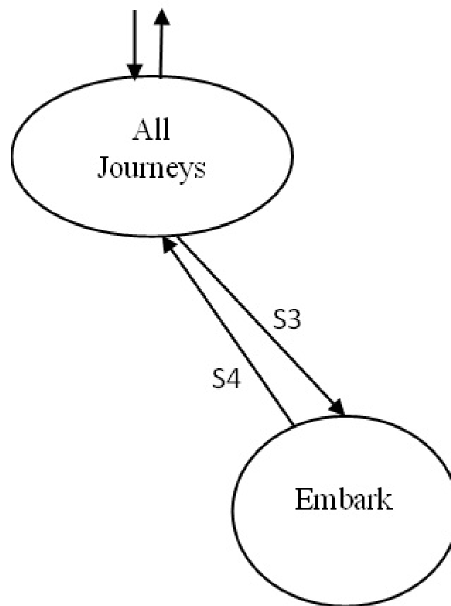


Figure (D.15) All Journeys Cluster for DiscoverU App

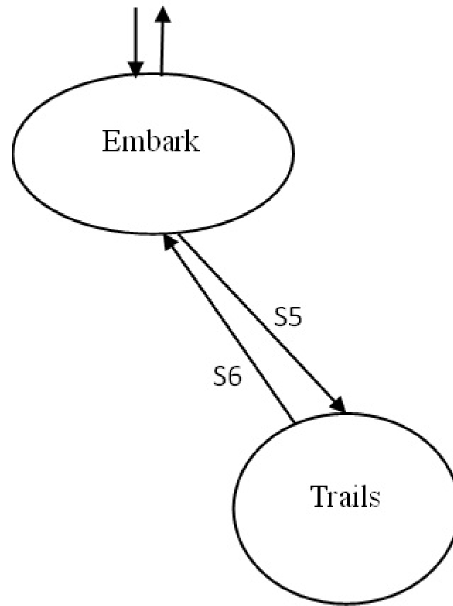


Figure (D.16) Embark Journeys Cluster for DiscoverU App

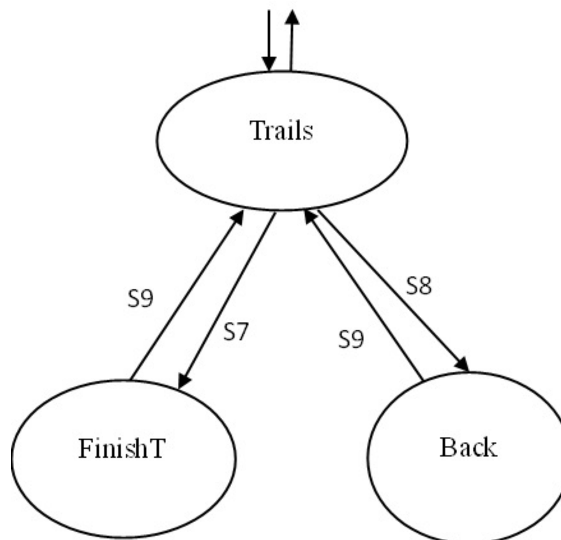


Figure (D.17) Trails Cluster for DiscoverU App

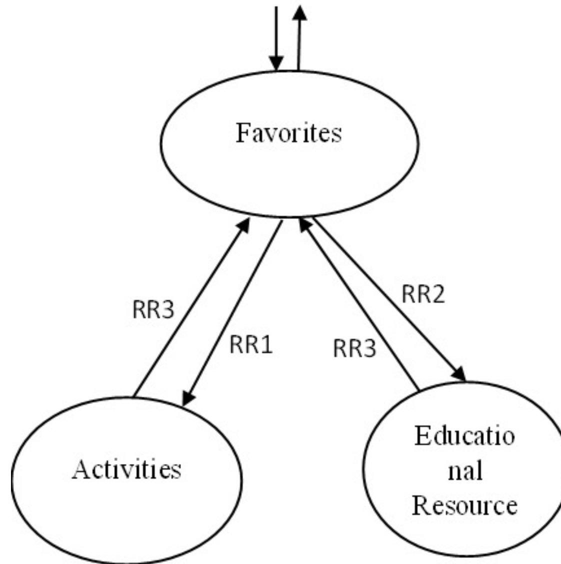


Figure (D.18) Favorites-Backpack Cluster for DiscoverU App

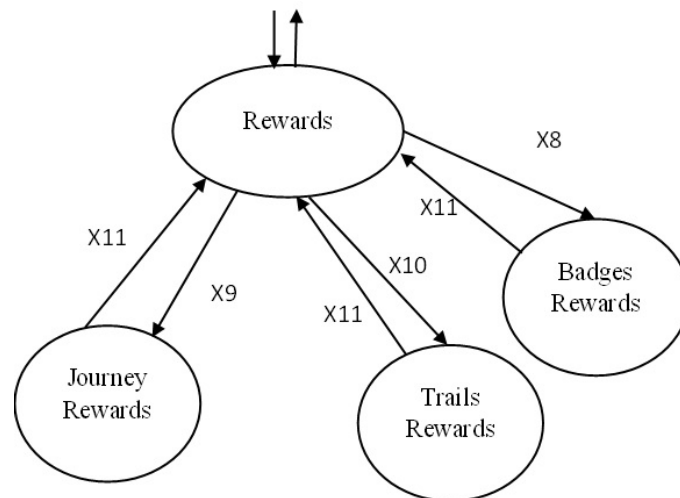


Figure (D.19) Rewards-Backpack Cluster for DiscoverU App

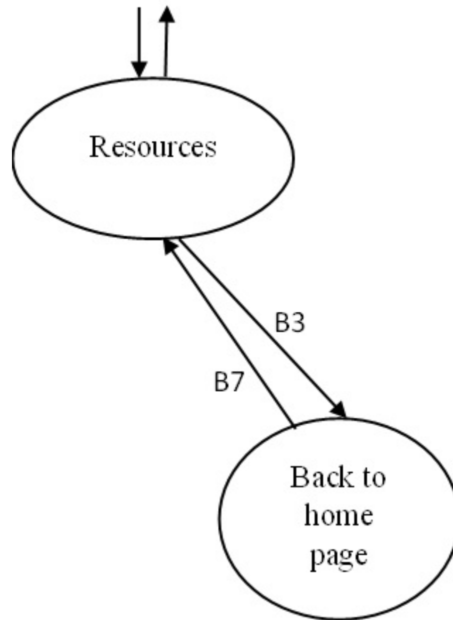


Figure (D.20) Resoutces-Menu Cluster for DiscoverU App

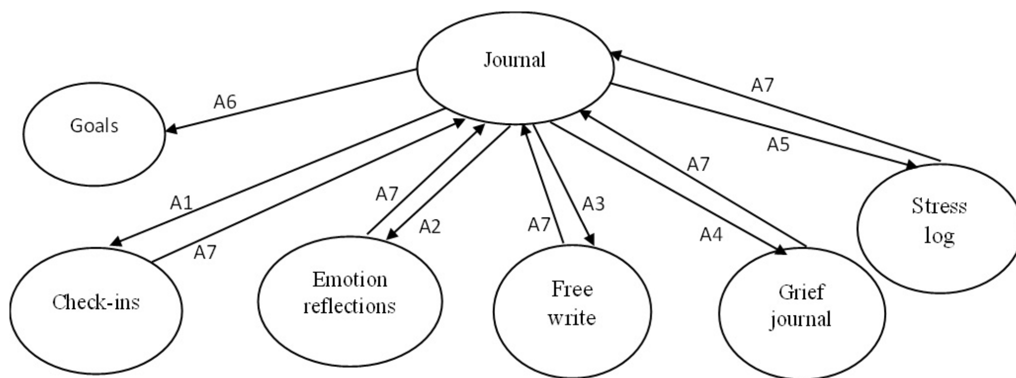


Figure (D.21) Journal-SubMenu Cluster for DiscoverU App

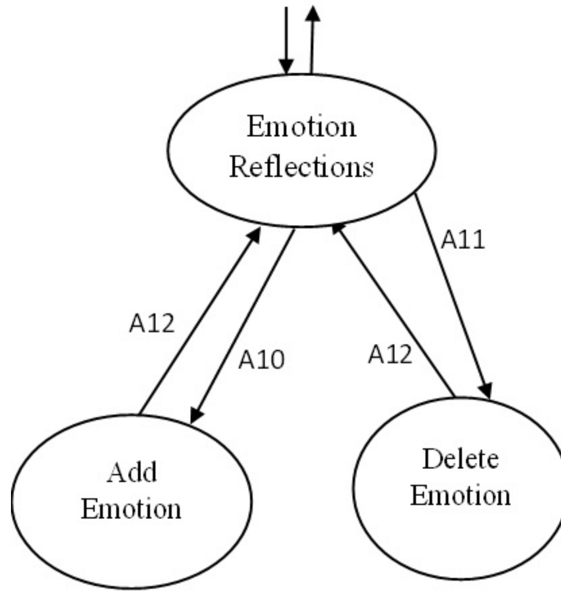


Figure (D.22) Emotion Reflections Cluster for DiscoverU App

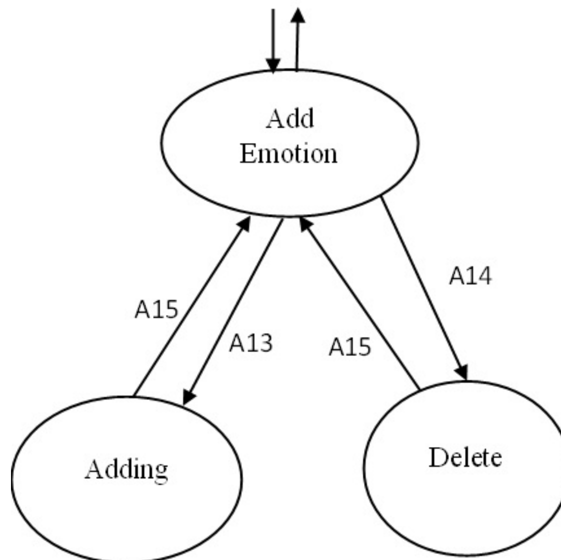


Figure (D.23) Add Emotion Reflections Cluster for DiscoverU App

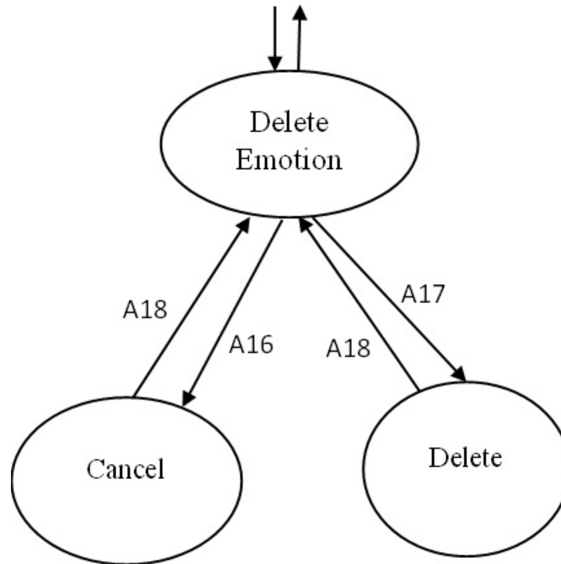


Figure (D.24) Delete Emotion Reflections for DiscoverU App

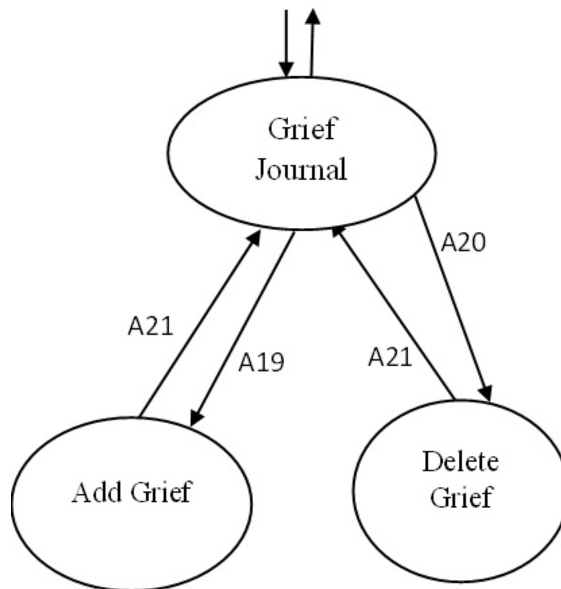


Figure (D.25) Grief Journal Cluster for DiscoverU App

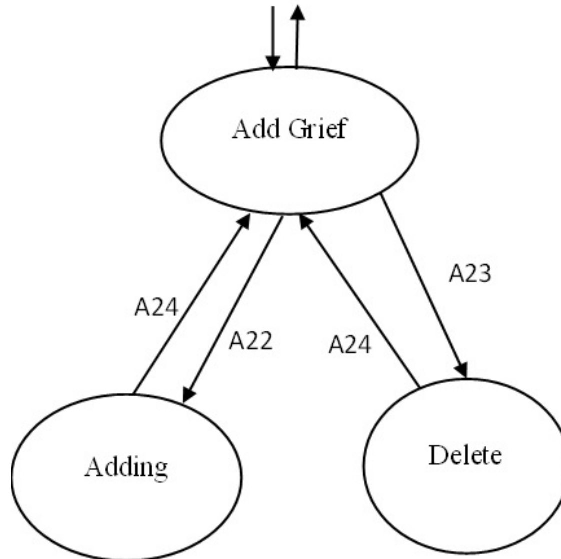


Figure (D.26) Add Grief Journal Cluster for DiscoverU App

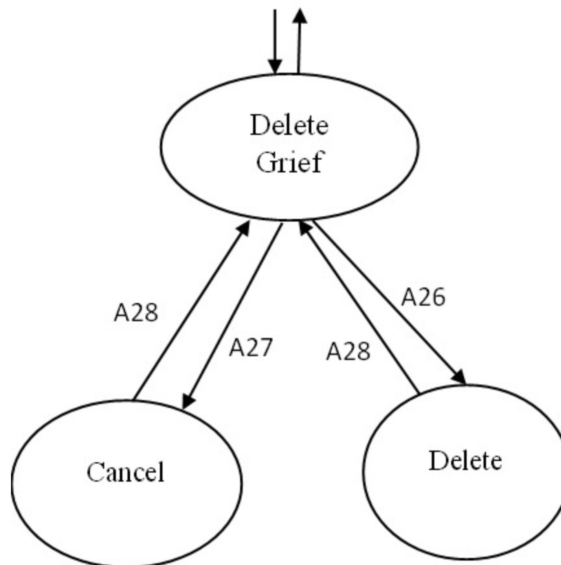


Figure (D.27) Delete Grief Journal for DiscoverU App

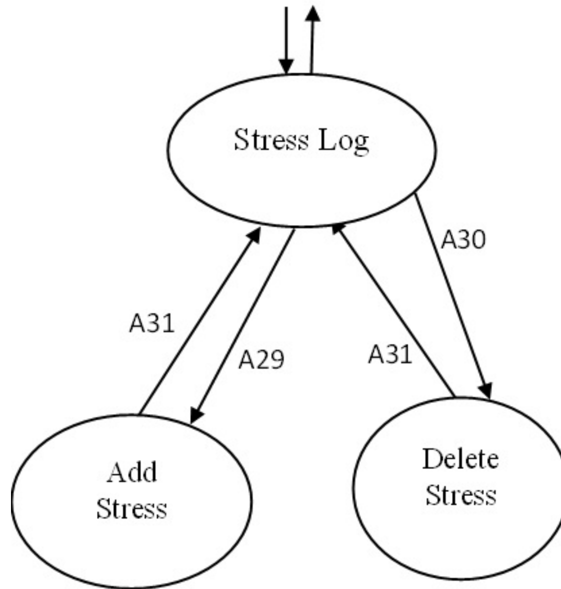


Figure (D.28) Stress Log Cluster for DiscoverU App

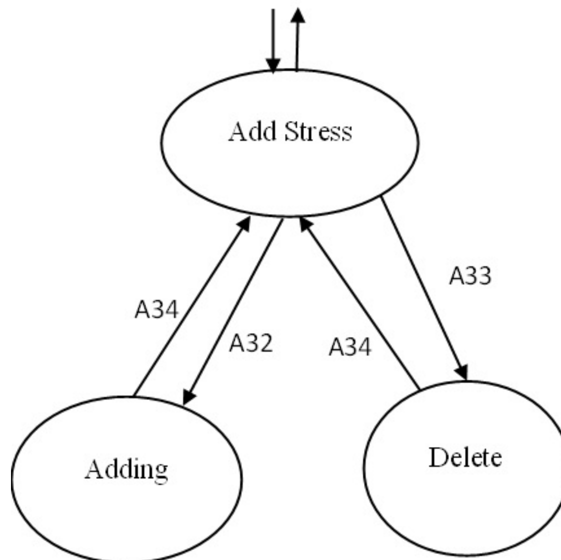


Figure (D.29) Add Stress Log Cluster for DiscoverU App

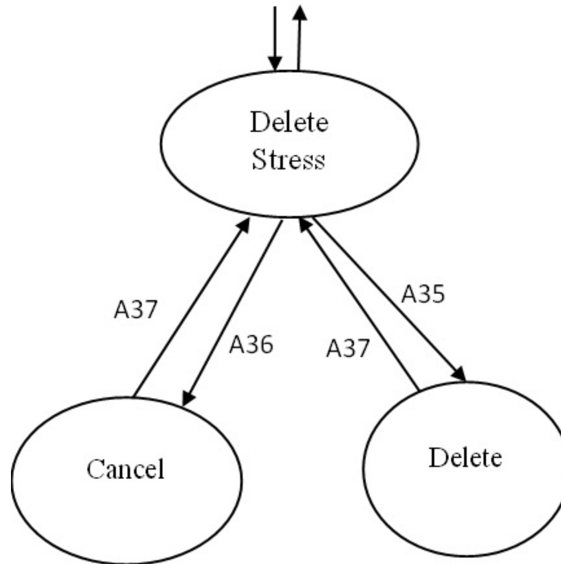


Figure (D.30) Delete Stress Log for DiscoverU App

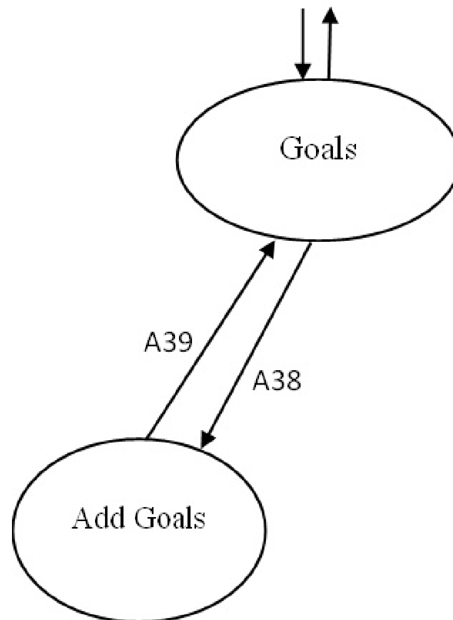


Figure (D.31) Goals Cluster for DiscoverU App

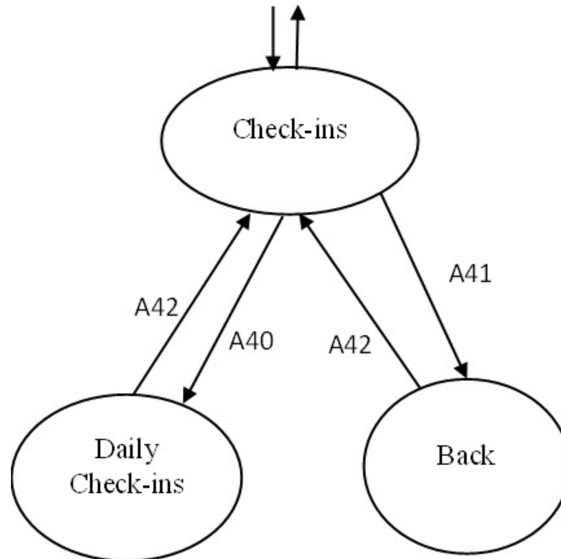


Figure (D.32) Check-ins Cluster for DiscoverU App

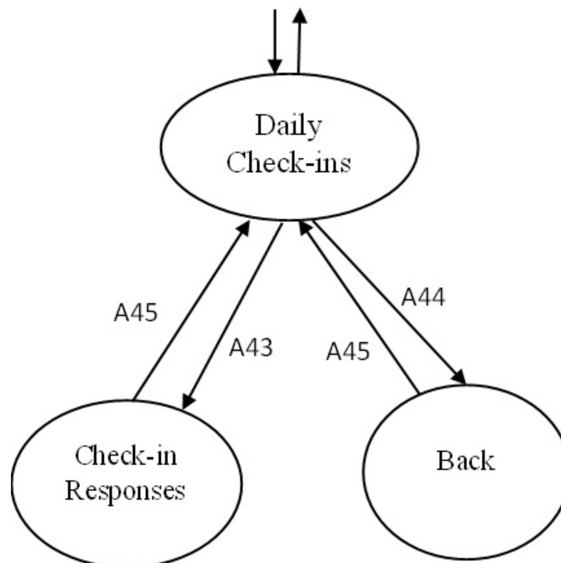


Figure (D.33) Check-ins Responses Cluster for DiscoverU App

D.2 DiscoverU App Case Study Test Paths

Table (D.1) Test Cases for the Versions of the DiscoverU App.

Begin of Table		
ID	**Test Sequence**	**Length**
1	[Entry Page, Sign In, Entry Page, Cancel, Entry Page, Exit]	6
2	[Entry Page, Login, Entry Page, Cancel, Entry Page, Exit]	6
3	[Main, Basecamp, How are you feeling, Check in Now, Next, Check in Now, Done, Check in Now, Skip, Check in Now, How are you feeling, Later Today, How are you feeling, Basecamp, Main, Exit]	16
4	[Main, Basecamp, Free Write, Add, Adding Free Write, Add, Delete, deleting, Delete, Cancel, Delete, Add, Free Write, Basecamp, Main, Exit]	16
5	[Main, Basecamp, Free Write, Delete, deleting, Delete, Cancel, Delete, Free Write, Basecamp, Demo Theme, Basecamp, Main, Exit]	12
6	[Main, Basecamp, Demo Theme, Basecamp, Main, Exit]	6
7	[Main, Explore, All Content, Filter, Filtering Contents, Filter, Cancel Filter, Filter, All Content, Back, All Content, Explore, Main, Exit]	14
8	[Main, Explore, Read, SelectReading, Read, Cancel, Read, Explore, Main, Exit]	10

Continuation of Table D.1		
ID	**Test Sequence**	**Length**
9	[Main, Explore, Video, SelectVideo, Video, Cancel, Video, Explore, Main, Exit]	10
10	[Main, Explore, Image, SelectImage, Image, Cancel, Image, Explore, Main, Exit]	10
11	[Main, Explore, Listen, SelectListen, Listen, Cancel, Listen, Explore, Main, Exit]	10
12	[Main, Journey, a journey, Embark on this Journey, Trails, Finish Trails, Trails, Embark on this Journey, a journey, Journey, Main, Exit]	12
13	[Main, Backpack, Favorites, Explore, All Content, Filter, Filtering Contents, Filter, Cancel Filter, Filter, All Content, Back, All Content, Explore, Favorites, Backpack, Main, Exit]	18
14	[Main, Backpack, Favorites, Explore, Content, SelectContent, Content, Explore, Favorites, Backpack, Main, Exit]	12
15	[Main, Backpack, Rewords, Badges, Rewords, Journey, Rewords, Trails, Rewords, Backpack, Main, Exit]	12
16	[Main, Backpack, Tools, Backpack, Main, Exit]	6
17	[Main, Backpack, History, Backpack, Main, Exit]	6
18	[Main, Backpack, Extra Steps, Backpack, Main, Exit]	6
19	[Main, Menu, Resources, Menu, Main, Exit]	6

Continuation of Table D.1		
ID	**Test Sequence**	**Length**
20	[Main, Menu, Favorites, Basecamp, How are you feeling, Check in Now, Next, Check in Now, Done, Check in Now, Skip, Check in Now, How are you feeling, Later Today, How are you feeling, Basecamp, Favorites, Menu, Main, Exit]	20
21	[Main, Menu, Favorites, Basecamp, Free write, Add, Adding, Add, Delete, Del, Delete, Cancel, Delete, Add, Free write, Basecamp, Favorites, Menu, Main, Exit]	20
22	[Main, Menu, Favorites, Basecamp, Free write, Delete, Del, Delete, Cancel, Delete, Free write, Basecamp, Demo Theme, Basecamp, Favorites, Menu, Main, Exit]	18
23	[Main, Menu, Backpack, BackpackPage, Favorites, Explore, Content, Explore, Favorites, BackpackPage, Backpack, Menu, Main, Exit]	14
24	[Main, Menu, Backpack, Backpack, Favorites, Explore, All Content, Filter, Filtering, Filter, Cancel Filter, Filter, All Content, Back, All Content, Explore, Favorites, Backpack, Backpack, Menu, Main, Exit]	22
25	[Main, Menu, Backpack, Backpack, Rewords, Badges, Rewords, Journey, Rewords, Trails, Rewords, Backpack, Backpack, Menu, Main, Exit]	16
26	[Main, Menu, Backpack, Backpack, Tools, Backpack, Backpack, Menu, Main, Exit]	10

Continuation of Table D.1		
ID	**Test Sequence**	**Length**
27	[Main, Menu, Backpack, Backpack, History, Backpack, Backpack, Menu, Main, Exit]	10
28	[Main, Menu, Backpack, Backpack, Extra Steps, Backpack, Backpack, Menu, Main, Exit]	10
29	[Main, Menu, Tools, Journey, a journey, Embark on this Journey, Trails, Finish Trails, Trails, Embark on this Journey, a journey, Journey, Tools, Menu, Main, Exit]	16
30	[Main, Menu, Journal, Check-ins, Daily Check-ins, Check-ins, Journal, Menu, Main, Exit]	10
31	[Main, Menu, Journal, Free Write, Add, Add FreeW, Add, Delete, Del FreeW, Delete, Cancel, Delete, Add, Free Write, Journal, Menu, Main, Exit]	18
32	[Main, Menu, Journal, Free Write, Delete, Del FreeW, Delete, Cancel, Delete, Free Write, Journal, Menu, Main, Exit]	14
33	[Main, Menu, Journal, Emotion Reflections, Add, Add Emotion, Add, Delete, Del Emotion, Delete, Cancel, Delete, Add, Emotion Reflections, Journal, Menu, Main, Exit]	18
34	[Main, Menu, Journal, Emotion Reflections, Delete, Del Emotion, Delete, Cancel, Delete, Emotion Reflections, Journal, Menu, Main, Exit]	14

Continuation of Table D.1		
ID	**Test Sequence**	**Length**
35	[Main, Menu, Journal, Grief Journal, Add, Add Grief, Add, Delete, Del Grife, Delete, Cancel, Delete, Add, Grief Journal, Journal, Menu, Main, Exit]	18
36	[Main, Menu, Journal, Grief Journal, Delete, Del Grife, Delete, Cancel, Delete, Grief Journal, Journal, Menu, Main, Exit]	14
37	[Main, Menu, Journal, Stress Log, Add, Add StressL, Add, Delete, Del StressL, Delete, Cancel, Delete, Add, Stress Log, Journal, Menu, Main, Exit]	18
38	[Main, Menu, Journal, Stress Log, Delete, Del StressL, Delete, Cancel, Delete, Stress Log, Journal, Menu, Main, Exit]	14
39	[Main, Menu, Journal, Goals, Add, Goals, Journal, Menu, Main]	9
40	[Main, Menu, Check-in, Check in Now, Next, Check in Now, Done, Check in Now, Skip, Check in Now, Check-in, Later Today, Check-in, Menu, Main, Exit]	16
41	[Main, Menu, Settings, Account Settings, Save Change, Account Settings, Back, Account Settings, Settings, Menu, Main, Exit]	12
42	[Main, Menu, Settings, Personalization, Save Change, Personalization, Back, Personalization, Settings, Menu, Main, Exit]	12

Continuation of Table D.1		
ID	**Test Sequence**	**Length**
43	[Main, Menu, Settings, Information, Save Change, Information, Back, Information, Settings, Menu, Main, Exit]	12
44	[Main, Menu, Logout, EntryPage, Exit]	5
End of Table		

D.3 DiscoverU App Inputs and Complexity of each Test path

Table (D.2) The Inputs and the Complexity of the Test Paths of the DiscoverU App.

Begin of Table				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
1	I1	R(Button-Entrypage)	Button-Entrypage = click	1
	L1	R(Parusername, ParPassword, Button-Login)	Parusername= "zeinab@gmail.com", ParPassword=" 1234", Button-Login = click	3
	I1	R(Button-Entrypage)	Button-Entrypage = click	1
	Total Complexity			5
2	I1	R(Button-Entrypage)	Button-Entrypage = click	1
	L2	R(Button-Signin)	Button-Signin = click	1
	LL1	R(Parusername, ParPassword, ParConfirmPW, Button-CreateAccount)	Parusername= "zeinab@gmail.com", ParPassword=" 1234", ParConfirmPW= " 1234", Button-CreateAccount = click	4
	I1	R(Button-Entrypage)	Button-Entrypage = click	1
	Total Complexity			7

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
3	E1	R(Tab-Basecamp)	Tab-Basecamp = click	1
	M1	R(Button-HowAre)	Button-HowAre=click	1
	N1	R(Button-ChNow)	Button-ChNow=click	1
	NN1	R(SelectFeel, Button-Next)	SelectFeel= "Good" Button-Next=click	2
	NN2	R(ParTell, Button-Done) S(ParTell, Button-Done)	ParTell= "I Feel Good Now" Button-Done=Click	2
	NN4	O(ParTell) R (Button-Skip) S(ParTell, Button-Skip)	ParTell= "I Feel Good Now" Button-Skip=Click	2
	M1	R(Button-HowAre)	Button-HowAre=click	1
	N2	R(Button-ChLater)	Button-ChLater=click	1
	E1	R(Tab-Basecamp)	Tab-Basecamp = click	1
	Total Complexity			12
4	E1	R(Tab-Basecamp)	Tab-Basecamp = click	1
	M2	R(Button-FreeW)	Button-FreeW=click	1
	P1	R(Button-Add)	Button-Add=click	1
	PP1	R(ParTitleW,ParFW, Button-FW)	ParTitleW = "FirstFW"	3

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
		S(ParTitleW,ParFW, Button-FW)	ParFW = " I have concerns...." Button-FW=click	
	PP2	O(ParTitleW,ParFW) R(Button-DeleteFW) S(ParTitleW,ParFW, Button-DeleteFW)	ParTitleW = "FirstFW" ParFW = " I have concerns...." Button-DeleteFW=click	3
	PPP1	R (Button-Delete)	Button-Delete= click	1
	PPP2	R (Button-Cancel)	Button-Cancel= click	1
	M2	R(Button-FreeW)	Button-FreeW=click	1
	E1	R(Tab-Basecamp)	Tab-Basecamp = click	1
	Total Complexity			13
5	E1	R(Tab-Basecamp)	Tab-Basecamp = click	1
	M2	R(Button-FreeW)	Button-FreeW=click	1
	P2	R(Button-Delete)	Button-Delete=click	1
	PPP1	R (Button-Delete)	Button-Delete= click	1
	PPP2	R (Button-Cancel)	Button-Cancel= click	1
	M2	R(Button-FreeW)	Button-FreeW=click	1
	E1	R(Tab-Basecamp)	Tab-Basecamp = click	1
	Total Complexity			7
6	E1	R(Tab-Basecamp)	Tab-Basecamp = click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	M3	R(Swap, Button-Demo)	Swap up, Button-Demo=click	2
	E1	R(Tab-Basecamp)	Tab-Basecamp = click	1
	Total Complexity			4
7	E2	R(Tab-Explore)	Tab-Explore = click	1
	R1	R(Button-AllContent)	Button-AllContent=click	1
	RR1	R(Button-Filter)	Button-Filter = click	1
	RRR1	R(SelectSortby, Select-Filtertopic, SelectFilte-format, Button-Close)	SelectSortby = "A to Z", SelectFiltertopic = "Depression", SelectFiltefor-mat= "Articles" , Button-close=click	4
	R1	R(Button-AllContent)	Button-AllContent=click	1
	RR2	R (Button-Back)	Button-Back= click	1
	E2	R(Tab-Explore)	Tab-Explore = click	1
	Total Complexity			10
8	E2	R(Tab-Explore)	Tab-Explore = click	1
	R3	R(Button-Content)	Button-Content=click	1
	R5	R(Button-Read)	Button-Read=click	1
	R6	R(Button-Back)	Button-Back=click	1
	R3	R(Button-Content)	Button-Content=click	1
	E2	R(Tab-Explore)	Tab-Explore = click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	Total Complexity			6
9	E2	R(Tab-Explore)	Tab-Explore = click	1
	R3	R(Button-Content)	Button-Content=click	1
	R8	R(Button-Video)	Button-Video=click	1
	R9	R(Button-Back)	Button-Back=click	1
	R3	R(Button-Content)	Button-Content=click	1
	E2	R(Tab-Explore)	Tab-Explore = click	1
	Total Complexity			6
10	E2	R(Tab-Explore)	Tab-Explore = click	1
	R3	R(Button-Content)	Button-Content=click	1
	R11	R(Button-Image)	Button-Video=Image	1
	R12	R(Button-Back)	Button-Back=click	1
	R3	R(Button-Content)	Button-Content=click	1
	E2	R(Tab-Explore)	Tab-Explore = click	1
	Total Complexity			6
11	E2	R(Tab-Explore)	Tab-Explore = click	1
	R3	R(Button-Content)	Button-Content=click	1
	R14	R(Button-listen)	Button-Listen=click	1
	R15	R(Button-Back)	Button-Back=click	1
	R3	R(Button-Content)	Button-Content=click	1
	E2	R(Tab-Explore)	Tab-Explore = click	1
	Total Complexity			6

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
12	E3	R(Tab-Journey)	Tab-Journey = click	1
	S1	R(Button-AllJourney)	Button-AllJourney=click	1
	S3	R(Button-Embark)	Button-Embark=click	1
	S5	R(Button-Trail)	Button-Trail=click	1
	R7	R(Button-FinishT)	Button-FinishT=click	1
	R8	R(Button-BackT)	Button-BackT=click	1
	S1	R(Button-AllJourney)	Button-AllJourney=click	1
	E2	R(Tab-Journey)	Tab-Journey = click	1
	Total Complexity			8
13	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	X1	R(Tab-Favorites)	Tab-Favorites=click	1
	XX1	R(Button-Activities)	Button-Activities =click	1
	R1	R(Button-AllContent)	Button-AllContent=click	1
	RR1	R(Button-Filter)	Button-Filter = click	1
	RRR1	R(SelectSortby, Select-Filtertopic, SelectFilte-format, Button-Close)	SelectSortby = "A to Z", SelectFiltertopic = "Depression", SelectFilteformat= "Articles" , Button-close=click	4
	R1	R(Button-AllContent)	Button-AllContent=click	1
	RR2	R (Button-Back)	Button-Back= click	1
	E2	R(Tab-Explore)	Tab-Explore = click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	Total Complexity			13
14	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	X1	R(Tab-Favorites)	Tab-Favorites=click	1
	XX1	R(Button-Activities)	Button-Activities =click	1
	R3	R(Button-Content)	Button-Content=click	1
	R5	R(Button-Read)	Button-Read=click	1
	R6	R(Button-Back)	Button-Back=click	1
	R3	R(Button-Content)	Button-Content=click	1
	E2	R(Tab-Explore)	Tab-Explore = click	1
	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	Total Complexity			9
15	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	X2	R(Tab-Rewords)	Tab-Rewords=click	1
	X8	R(Button-BadgesR)	Button-BadgesR =click	1
	X9	R(Button-JourneyR)	Button-JourneyR=click	1
	X10	R(Button-TrailsR)	Button-ReTrailsRad=click	1
	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	Total Complexity			6
16	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	X3	R(Tab-Tools)	Tab-Tools=click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	Total Complexity			3
17	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	X4	R(Tab-History)	Tab-History=click	1
	XX4	R(Button-ActivityCalendar)	Button-ActivityCalendar =click	1
	X9	R(Button-JourneyProgress)	Button-JourneyR=click	1
	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	Total Complexity			5
18	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	X5	R(Button-ExtraSteps)	Button-ExtraSteps=click	1
	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	Total Complexity			3
19	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	X5	R(Tab-ExtraSteps)	Tab-ExtraSteps=click	1
	X8	R(Button-BadgesR)	Button-BadgesR =click	1
	X9	R(Button-JourneyR)	Button-JourneyR=click	1
	X10	R(Button-TrailsR)	Button-ReTrailsRad=click	1
	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	Total Complexity			6

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
20	E5	R(Button-Menu)	Button-Menu = click	1
	W1	R(Button-Resources)	Button-Resources=click	1
	B3	R(Button-BackHome)	Button-BackHome =click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			4
21	E5	R(Button-Menu)	Button-Menu = click	1
	W2	R(Button-FavoritesM)	Button-FavoritesM=click	1
	M1	R(Button-HowAre)	Button-HowAre=click	1
	N1	R(Button-ChNow)	Button-ChNow=click	1
	NN1	R(SelectFeel, Button-Next)	SelectFeel= "Good" Button-Next=click	2
	NN2	R(ParTell, Button-Done) S(ParTell, Button-Done)	ParTell= "I Feel Good Now" Button-Done=Click	2
	NN2	O(ParTell) R (Button-Skip) S(ParTell, Button-Skip)	ParTell= "I Feel Good Now" Button-Skip=Click	2
	M1	R(Button-HowAre)	Button-HowAre=click	1
	N2	R(Button-ChLater)	Button-ChLater=click	1
	E1	R(Tab-Basecamp)	Tab-Basecamp = click	1
	E4	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			14

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
22	E5	R(Button-Menu)	Button-Menu = click	1
	W2	R(Button-FavoritesM)	Button-FavoritesM=click	1
	M2	R(Button-FreeW)	Button-FreeW=click	1
	P1	R(Button-Add)	Button-Add=click	1
	PP1	R(ParTitleW,ParFW, Button-FW) S(ParTitleW,ParFW, Button-FW)	ParTitleW = "FirstFW" ParFW = " I have concerns...." Button-FW=click	3
	PP2	O(ParTitleW,ParFW) R(Button-DeleteFW) S(ParTitleW,ParFW, Button-DeleteFW)	ParTitleW = "FirstFW" ParFW = " I have concerns...." Button-DeleteFW=click	3
	PPP1	R (Button-Delete)	Button-Delete= click	1
	PPP2	R (Button-Cancel)	Button-Cancel= click	1
	M2	R(Button-FreeW)	Button-FreeW=click	1
	E1	R(Tab-Basecamp)	Tab-Basecamp = click	1
	E4	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			15
23	E5	R(Button-Menu)	Button-Menu = click	1
	W2	R(Button-FavoritesM)	Button-FavoritesM=click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	M2	R(Button-FreeW)	Button-FreeW=click	1
	P2	R(Button-Delete)	Button-Delete=click	1
	PPP1	R (Button-Delete)	Button-Delete= click	1
	PPP2	R (Button-Cancel)	Button-Cancel= click	1
	M2	R(Button-FreeW)	Button-FreeW=click	1
	E1	R(Tab-Basecamp)	Tab-Basecamp = click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			9
24	E5	R(Button-Menu)	Button-Menu = click	1
	W3	R(Button-BackpackM)	Button-BackpackM=click	1
	X1	R(Tab-Favorites)	Tab-Favorites=click	1
	XX1	R(Button-Activities)	Button-Activities =click	1
	R1	R(Button-AllContent)	Button-AllContent=click	1
	P1	R(Button-Add)	Button-Add=click	1
	RR1	R(Button-Filter)	Button-Filter = click	1
	RRR1	R(SelectSortby, Select-Filtertopic, SelectFilte-format, Button-Close)	SelectSortby = "A to Z", SelectFiltertopic = "Depression", SelectFilteformat= "Articles" , Button-close=click	4
	R1	R(Button-AllContent)	Button-AllContent=click	1
	RR2	R (Button-Back)	Button-Back= click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	E2	R(Tab-Explore)	Tab-Explore = click	1
	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			16
25	E5	R(Button-Menu)	Button-Menu = click	1
	W3	R(Button-BackpackM)	Button-BackpackM=click	1
	X1	R(Tab-Favorites)	Tab-Favorites=click	1
	XX1	R(Button-Activities)	Button-Activities =click	1
	R3	R(Button-Content)	Button-Content=click	1
	R5	R(Button-Read)	Button-Read=click	1
	R6	R(Button-Back)	Button-Back=click	1
	R3	R(Button-Content)	Button-Content=click	1
	E2	R(Tab-Explore)	Tab-Explore = click	1
	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			11
26	E5	R(Button-Menu)	Button-Menu = click	1
	W3	R(Button-BackpackM)	Button-BackpackM=click	1
	X2	R(Tab-Rewords)	Tab-Rewords=click	1
	X8	R(Button-BadgesR)	Button-BadgesR =click	1
	X9	R(Button-JourneyR)	Button-JourneyR=click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	X10	R(Button-TrailsR)	Button-ReTrailsRad=click	1
	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			8
27	E5	R(Button-Menu)	Button-Menu = click	1
	W3	R(Button-BackpackM)	Button-BackpackM=click	1
	X3	R(Button-Tools)	Button-Tools=click	1
	X5	R(Button-ExtraStep)	Button-ExtraStep =click	1
	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			6
28	E5	R(Button-Menu)	Button-Menu = click	1
	W3	R(Button-BackpackM)	Button-BackpackM=click	1
	X4	R(Button-History)	Button-History=click	1
	XX4	R(Button-Activitycalendar)	Button-Activitycalendar =click	1
	XX5	R(Button-Journeyprogress)	Button-Journeyprogress =click	1
	E4	R(Tab-Backpack)	Tab-Backpack = click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			7

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
29	E5	R(Button-Menu)	Button-Menu = click	1
	W4	R(Button-ToolsM)	Button-ToolsM=click	1
	S1	R(Button-AllJourney)	Button-AllJourney=click	1
	S3	R(Button-Embark)	Button-Embark=click	1
	S5	R(Button-Trail)	Button-Trail=click	1
	R7	R(Button-FinishT)	Button-FinishT=click	1
	R8	R(Button-BackT)	Button-BackT=click	1
	S1	R(Button-AllJourney)	Button-AllJourney=click	1
	E2	R(Tab-Journey)	Tab-Journey = click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			10
30	E5	R(Button-Menu)	Button-Menu = click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	A1	R(Button-check-ins)	Button-check-ins=click	1
	A40	R(Button-DailyWeekR)	Button-DailyWeekR=click	1
	A1	R(Button-check-ins)	Button-check-ins=click	1
	A41	R(Button-Back)	Button-Back = click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			8
31	E5	R(Button-Menu)	Button-Menu = click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	W5	R(Button-JournalM)	Button-JournalM=click	1
	A3	R(Button-FreeW)	Button-FreeW=click	1
	P1	R(Button-Add)	Button-Add=click	1
	PP1	R(ParTitleW,ParFW, Button-FW) S(ParTitleW,ParFW, Button-FW)	ParTitleW = "FirstFW" ParFW = " I have concerns...." Button-FW=click	3
	PP2	O(ParTitleW,ParFW) R(Button-DeleteFW) S(ParTitleW,ParFW, Button-DeleteFW)	ParTitleW = "FirstFW" ParFW = " I have concerns...." Button-DeleteFW=click	3
	PPP1	R (Button-Delete)	Button-Delete= click	1
	PPP2	R (Button-Cancel)	Button-Cancel= click	1
	M2	R(Button-FreeW)	Button-FreeW=click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			15
32	E5	R(Button-Menu)	Button-Menu = click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	A3	R(Button-FreeW)	Button-FreeW=click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	P2	R(Button-Delete)	Button-Delete=click	1
	PPP1	R (Button-Delete)	Button-Delete= click	1
	PPP2	R (Button-Cancel)	Button-Cancel= click	1
	M2	R(Button-FreeW)	Button-FreeW=click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			9
33	E5	R(Button-Menu)	Button-Menu = click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	A2	R(Button-Emotion)	Button-Emotion=click	1
	A10	R(Button-AddEmotion)	Button-AddEmotion=click	1
	A13	R(ParTitleEmotion, ParEmotion, Button-AddEmotion) S(ParTitleEmotion, ParEmotion, Button-AddEmotion)	ParTitleEmotion = "FirstEmotion" ParEmotion = " I have concerns...." Button-AddEmotion=click	3
	A14	O(ParTitleEmotion, ParEmotion) R(Button-DeleteEmotion)	ParTitleEmotion = "FirstEmotion"	3

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
		S(ParTitleEmotion, ParEmotion, Button-DeleteEmotion)	ParEmotion = " I have concerns...." Button-DeleteEmotion=click	
	A17	R (Button-Delete)	Button-Delete= click	1
	A16	R (Button-Cancel)	Button-Cancel= click	1
	A2	R(Button-Emotion)	Button-Emotion=click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			15
34	E5	R(Button-Menu)	Button-Menu = click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	A2	R(Button-Emotion)	Button-Emotion=click	1
	A11	R(Button-DeleteEM)	Button-DeleteEM=click	1
	A17	R (Button-Delete)	Button-Delete= click	1
	A16	R (Button-Cancel)	Button-Cancel= click	1
	A2	R(Button-Emotion)	Button-Emotion=click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			9
35	E5	R(Button-Menu)	Button-Menu = click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	A4	R(Button-Grief)	Button-Grief=click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	A19	R(Button-AddGrief)	Button-AddGrief=click	1
	A22	R(ParTitleGrief,ParGrief, Button-Grief) S(ParTitleGrief,ParGrief, Button-AGrief)	ParTitleGrief = "First-Grief" ParGrief = " I have concerns...." Button-AGrief=click	3
	A23	O(ParTitleGrief,ParGrief) R(Button-DeleteGrief) S(ParTitleGrief,ParGrief, Button-DeleteGrief)	ParTitleGrief = "First-Grief" ParGrief = " I have concerns...." Button-DeleteGrief=click	3
	A26	R (Button-Delete)	Button-Delete= click	1
	A27	R (Button-Cancel)	Button-Cancel= click	1
	A4	R(Button-Grief)	Button-Grief=click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			15
36	E5	R(Button-Menu)	Button-Menu = click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	A4	R(Button-Grief)	Button-Grief=click	1
	A20	R(Button-DeleteG)	Button-DeleteG=click	1
	A26	R (Button-Delete)	Button-Delete= click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	A27	R (Button-Cancel)	Button-Cancel= click	1
	A4	R(Button-Grief)	Button-Grief=click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			9
37	E5	R(Button-Menu)	Button-Menu = click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	A5	R(Button-Stress)	Button-Stress=click	1
	A29	R(Button-AddsStress)	Button-AddsStress=click	1
	A32	R(ParTitleStress, ParStress, Button-AStress) S(ParTitleStress, ParStress, Button-AStress)	ParTitleStress = "First-Stress" ParStress = " I have concerns...." Button-AStress=click	3
	A33	O(ParTitleStress, ParStress) R(Button-DeleteStress) S(ParTitleStress, ParStress, Button-DeleteStress)	ParTitleStress = "First-Stress" ParStress = " I have concerns...." Button-DeleteStress=click	3
	A35	R (Button-Delete)	Button-Delete= click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	A36	R (Button-Cancel)	Button-Cancel= click	1
	A5	R(Button-Stress)	Button-Stress=click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			15
38	E5	R(Button-Menu)	Button-Menu = click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	A5	R(Button-Stress)	Button-Stress=click	1
	A30	R(Button-DeleteStress)	Button-DeleteStress=click	1
	A35	R (Button-Delete)	Button-Delete= click	1
	A36	R (Button-Cancel)	Button-Cancel= click	1
	A5	R(Button-Stress)	Button-Stress=click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			9
39	E5	R(Button-Menu)	Button-Menu = click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1
	A6	R(Button-Goals)	Button-Goals=click	1
	A38	R(Button-AddGoals)	Button-AddGoals=click	1
	A6	R(Button-Goals)	Button-Goals=click	1
	W5	R(Button-JournalM)	Button-JournalM=click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			7
40	E5	R(Button-Menu)	Button-Menu = click	1
	W6	R(Button-Daily)	Button-Daily=click	1
	M1	R(Button-HowAre)	Button-HowAre=click	1
	N1	R(Button-ChNow)	Button-ChNow=click	1
	NN1	R(SelectFeel, Button-Next)	SelectFeel= "Good" Button-Next=click	2
	NN2	R(ParTell, Button-Done) S(ParTell, Button-Done)	ParTell= "I Feel Good Now" Button-Done=Click	2
	NN2	O(ParTell) R (Button-Skip) S(ParTell, Button-Skip)	ParTell= "I Feel Good Now" Button-Skip=Click	2
	M1	R(Button-HowAre)	Button-HowAre=click	1
	N2	R(Button-ChLater)	Button-ChLater=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			13
41	E5	R(Button-Menu)	Button-Menu = click	1
	W6	R(Button-Settings)	Button-Settings=click	1
	y1	R(ParAccInfo, Button-SaveChange)	ParAccInfo= " " Button-SaveChange=click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	y6	O(ParAccInfo) R(Button-Cancel)	ParAccInfo= " " Button-Cancel=click	1
	W6	R(Button-Settings)	Button-Settings=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			6
42	E5	R(Button-Menu)	Button-Menu = click	1
	W6	R(Button-Settings)	Button-Settings=click	1
	y2	R(ParPersonInfo, Button-SaveChangeP)	ParPersonInfo= " " Button-SaveChangeP=click	1
	y9	O(ParPersonInfo) R(Button-Cancel)	ParPersonInfo= " " Button-Cancel=click	1
	W6	R(Button-Settings)	Button-Settings=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			6
43	E5	R(Button-Menu)	Button-Menu = click	1
	W6	R(Button-Settings)	Button-Settings=click	1
	y3	R(ParGeneralInfo, Button-SaveChangeGeneralInfo)	ParGeneralInfo= " " Button-SaveChanGeneralInfo=click	1
	y12	O(ParGeneralInfo) R(Button-Cancel)	ParGeneralInfo= " " Button-Cancel=click	1

Continuation of Table D.2				
ID	**Edge**	**Constraint**	**Input Value and Actions**	Comp
	W6	R(Button-Settings)	Button-Settings=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			6
44	E5	R(Button-Menu)	Button-Menu = click	1
	W7	R(Button-Logout)	Button-Logout=click	1
	E5	R(Button-Menu)	Button-Menu = click	1
	Total Complexity			3
End of Table				